南京林业大学自编讲义

编 译 原 理 与 技 术
实 验 指 导 书

编者：薛联凤

信息科学与技术学院
2024 年 4 月

《编译原理与技术》实验大纲

一．课程的性质和目的

本课程是计算机专业的重要专业课之一，主要介绍程序设计语言编译构造的基本原理和基本实现方法。通过本课程学习，使学生对编译的基本概念、原理和方法有完整的和清楚的理解，并能正确地、熟练地运用。设置本课程的目的是：（1）使学生了解程序语言编译系统的结构及各部分的功能；　（2）使学生掌握设计和构造程序语言编译系统的基本原理和技术.

二、实习目的要求

本专业各类教学实践环节都是计算机专业的必修课程。目的是在学生学完了《C 语言》,《汇编语言》等课程的基础上，在计算机上进行实习，使学生在理论联系实际的过程中：1.了解语言产生环境、界面生成原理等有关方面的知识，获得编译原理的基本构成思想；2.加深对所学理论知识的理解；3.扩大知识面，对将准备在计算机行业的进一步发展学生有一个扩展的空间；4.培养和锻炼学生运用所学知识观察分析实际问题的能力；

三、实习内容

1.词法分析器：了解程序的词法构成和分析过程，了解词法分析单词的识别，掌握词法分析的程序编写。

2. 语法分析：了解预测分析程序与 LL(1)文法构造预测文法分析表求串 a 的 First(a)和 Follow(a)的集合构造预测分析表和状态表。

3.语法分析：了解语法分析构成，LR 分析器，LR(0)项目集族和 LR(0)分析表的构造，SLR 分析表的构造，规范 LR 分析表的构造。

4.递归下降分析程序：下推自动机概念，确定的自顶向下的分析思想。

5. 中间代码生成：各种常见中间语言形式，各种语句到四元式的翻译，自下而上分析制导翻译概述，布尔表达式的翻译，控制语句的翻译

目　　录

第一部分：windows 平台

实验一：词法分析

一、目的和内容

1、 实验目的：通过完成词法分析程序，了解词法分析的过程；

2、 实验内容：用 C 语言实现对 C 的子集程序设计语言的词法识别程序；

3、 实验要求：将该语言的源程序，也就是相应字符流转换成内码，并根据需要是否对于标识符填写相应的符号表供编译程序的以后各阶段使用；

二、程序设计语言的描述

程序设计语言的描述采用扩充的 BNF 表示：

<程序>-> <主程序><子程序>

<主程序>-> main <参数>

<子程序>->

<参数>->

<变量定义>-><类型>标识符{,标识符}

<类型>->int|long|short|float|char

<语句>-><赋值语句>|<条件语句>|<当型循环语句>|<调用子程序语句>|<复合语句>|ε

<赋值语句>->标识符=<表达式>

<条件语句>->IF<条件><语句>ELSE<语句>

<当型循环语句>->WHILE<条件><语句>

<调用子程序语句>->标识符|标识符(<表达式>)

<复合语句>->{<语句>{,<语句>}}

<条件>-><表达式><关系运算符><表达式>|odd<表达式>

<表达式>->[+|-]<项>{<加型运算符><项>}

<项>-><因子>{<乘型运算符><因子>}

<因子>->标识符|无符号整数|(<表达式>)

<加型运算符>->+|-

<乘型运算符>->*|/

<关系运算符>->=|!=|<|>|<=|>=

其中:

<>: 用左右尖括号括起来的字符串表示非终结符

{}: 表示该语句成分可以 0~n 次重复

[]: 表示方括号内为可选项,即 0 或 1 次

三、程序设计语言单词的内部编码

如表 1 为实验----词法分析中的内码单词对照表；

表 1.识别内码表

内码	单词	内码	单词	内码	单词	内码	单词
	auto		else		register		union
	break		enum		return		unsigned
	case		extern		short		void
	char		float		signed		volatile
	const		for		sizeof		while
	continue		goto		static		,
	default		if		struct		(
	do		int		switch		)
	double		long		typedef		[

内码	单词	内码	单词	内码	单词	内码	单词
	]		/		>=		/=
	->		%		==		;
	.		+		!=		{
	!		-		&&		}
	++		<<		‖		#
	--		>>		=		_
	&		<		+=		‘
	~		>		-=		
	*		<=		*=		

四、词法分析程序的设计思想

为了实现的编译程序实用，这里规定源程序可采用自由书写格式，即一行内可书写多个语句，一个语句可占领多行书写；标识符的前 20 个字符有效；整数用 2 个字节表示；长整数用 4 个字节表示。这样词法分析程序的主要工作为：

（1） 从源程序文件中读入字符

（2） 删除空格类字符，包括回车、制表符空格

（3） 按拼写单词，并用（内码，属性）二元式表示

（4） 根据需要是否填写标识符表供以后各阶段使用

这里采用的编译程序的实现方法是一遍扫描，既从左到右扫描一次源程序，也就是词法分析作为语法分析的一个子程序。故在编写词法分析程序时，

用重复调用词法分析子程序取一单词的方法得到整个源程序的内码流。扫描程序流程图，如图所示：

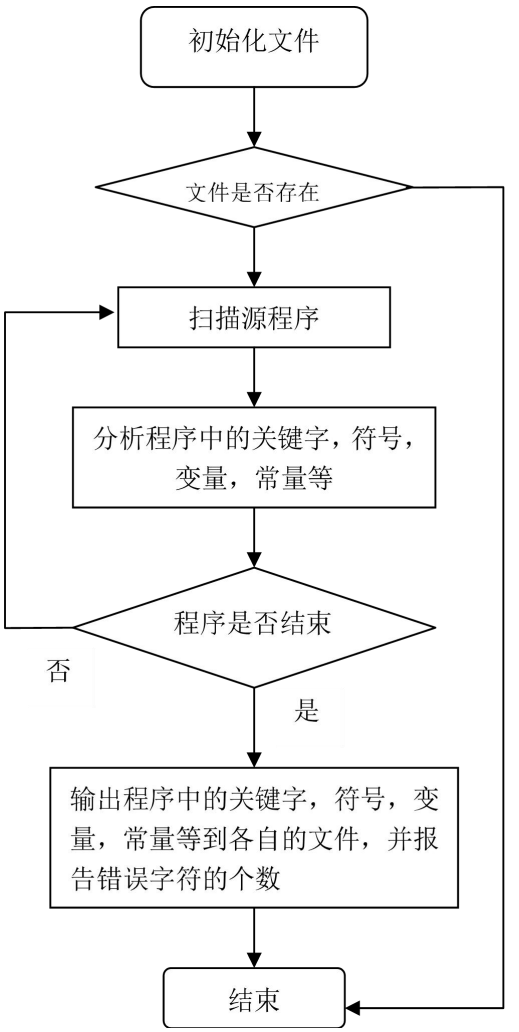


图 1　词法分析程序的流程图

三、实验过程和指导：

（一）准备：

1.阅读课本有关章节，明确语言的语法，写出基本保留字、标识符、常数、运算符、分隔符和程序样例。

2.初步编制好程序。

（二）上课上机：

将源代码拷贝到机上调试，发现错误，再修改完善。

第二次上机调试通过。

（三）程序要求：

程序输入/输出示例：

如源程序为 C 语言。输入如下一段：

```
main()
{
int i,j,m,n;
printf("please input data to i,j”);
i=1;
j=4;
m=i+j;
n=i-j;
printf("m=%d,n=%d",m,n);
```

}要求输出如下图。

（1，”main”）
（5，”（“）
（5，”）“）
（5，”{“）
（1，”int”）
（2，”i”）
（5，”,”）
（2，”j”）
（5，”,”）
（2，”m”）
（5，”,”）
（2，”n”）
（5，”;”）
（2，”m”）
（4，”=”）
（2，”i”）
（4，”+”）
（2，”j”）
（5，”;”）

（2，”n”）
（4，”=”）
（2，”i”）
（4，”+”）
（2，”j”）
（5，”;”）
（1，”printf”）
（5，”（“）
（4，”””）
（4，”%d”）
（5，”,”）
（4，”%d”）
（5，”,“）
（2，”m”）
（4，”+”）

（2，”n”）
(5,”)”)
（5，”;”）

（5，”}“）

要求：

识别保留字：main printf if、int、for、while、do、return、break、continue；单词种别码为 1。其他的都识别为标识符；单词种别码为 2。常数为无符号整形数；单词种别码为 3。运算符包括：+、-、*、/、=、>、<、>=、<=、!= ；单词种别码为 4。分隔符包括：,、;、{、}、(、)； 单词种别码为 5。

gei

（四）程序思路（仅供参考）：

这里以开始定义的 C 语言子集的源程序作为词法分析程序的输入数据。在词法分析中，自文件头开始扫描源程序字符，一旦发现符合“单词”定义的源程序字符串时，将它翻译成固定长度的单词内部表示，并查填适当的信息表。经过词法分析后，源程序字符串（源程序的外部表示）被翻译成具有等长信息的单词串（源程序的内部表示），并产生两个表格：常数表和标识符表，它们分别包含了源程序中的所有常数和所有标识符。

0. 定义部分：定义常量、变量、数据结构。

1. 初始化：从文件将源程序全部输入到字符缓冲区中。

2. 取单词前：去掉多余空白。

3.取单词后：去掉多余空白（可选，看着办）。

4.取单词：利用实验一的成果读出单词的每一个字符，组成单词，分析类型。（关键是如何判断取单词结束？取到的单词是什么类型的单词？）

5.显示结果。

（五）练习该实验的目的和思路：

程序开始变得复杂起来，可能是大家目前编过的程序中最复杂的，但相对于以后的程序来说还是简单的。因此要认真把握这个过渡期的练习。

本实验和以后的实验相关。通过练习，掌握对字符进行灵活处理的方法。

（六）为了能设计好程序，注意以下事情：

1.模块设计：将程序分成合理的多个模块（函数），每个模块做具体的同一事情。

2.写出（画出）设计方案：模块关系简图、流程图、全局变量、函数接口等。

3.编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

(七)基于面向对象的开发界面，供大家参考

一. 以下运行之后的可视界面：

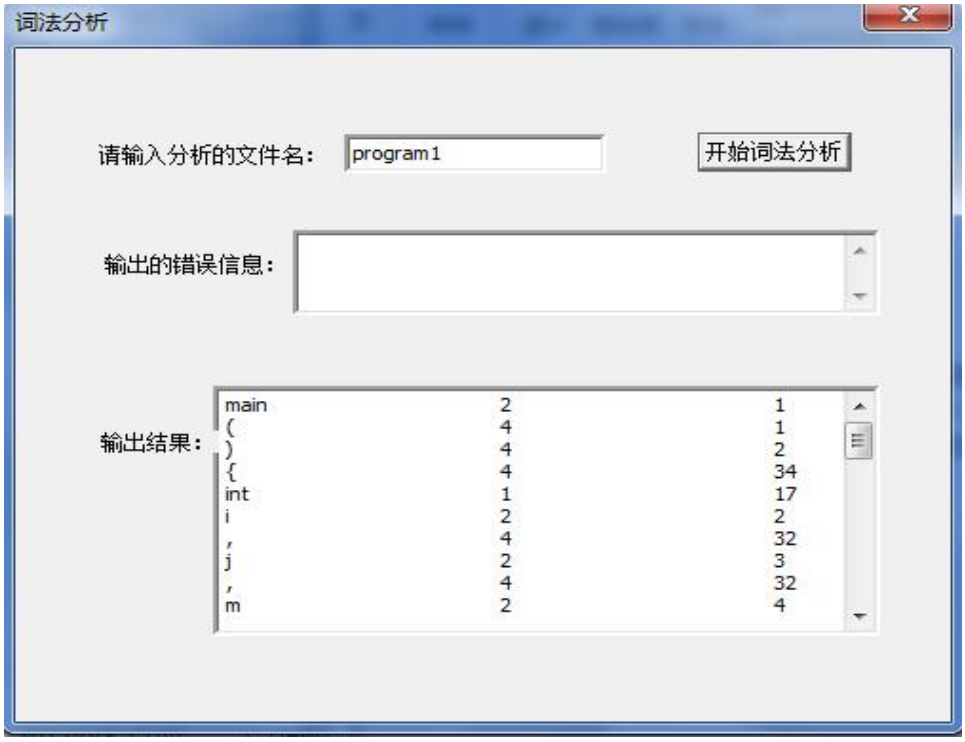
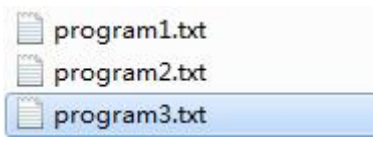


图 1.1 录入

说明：

1. 运行软件是：visual studio 2008
2. 把需要分析的 txt 文件（写了一个 C 语言程序）放到 MFCWordAnalyse 目录下面
3. 如上图一所示，program1 就是需要分析的 C 语言的文件名，点击开始词法分析，输出结果就会显示分析出来的结果（也就是 output.txt 文件中显示结果）。我所做的项目中有如下几个文件：



还可以自己加入一些文件都可以进行分析

二. 运行出错时出现的界面

1. 如果第一次运行成功后，再次输入另外一个不存在的文件名进行分析的话就会出现图二所示界面（输出的结果还是上次分析的结果，但是输出错误信息栏会出现 cannot open file）：

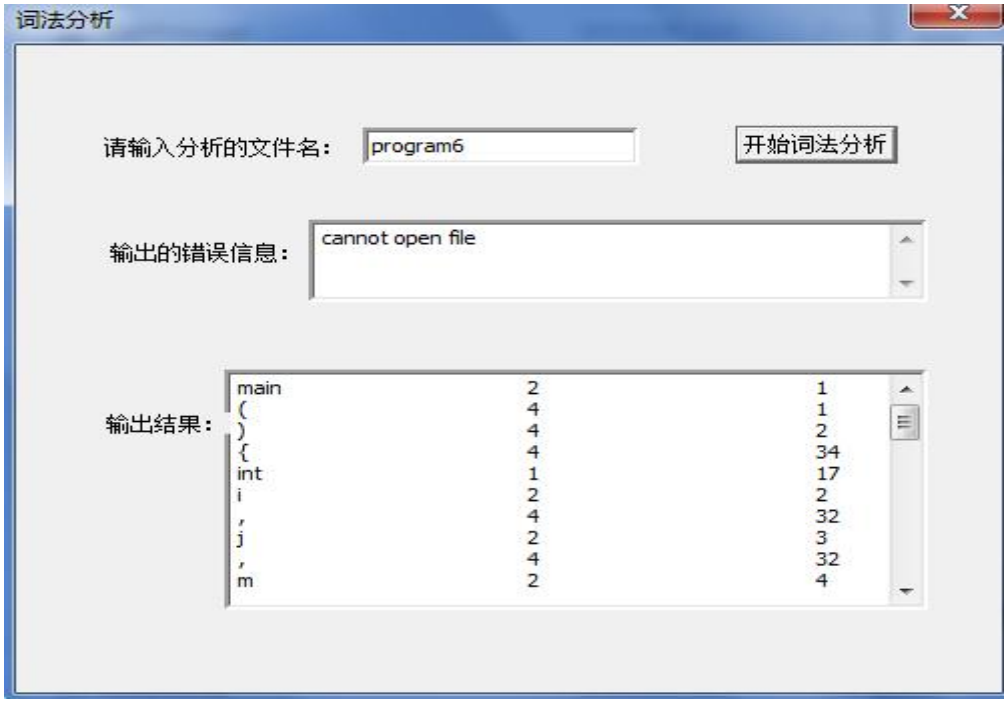


图 1.2 识别结果

2. 重新运行是保存会出现下面图三的结果：

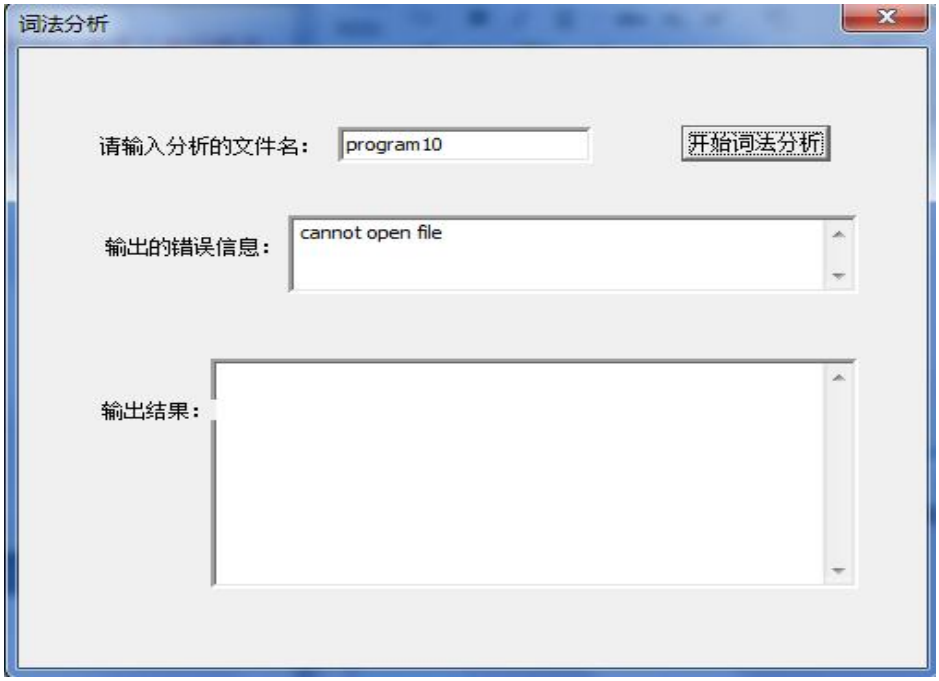


图 1.3 报错信息提示

实验二：语法分析法 1-递归子程序法

一、实验目的和内容：

1．实验目的：通过完成语法分析程序，了解语法分析的过程和作用。

2．实验内容：用递归子程序法实现对 C 的子集程序设计语言的分析程序

3．实验要求：根据某一文法编制调试递归下降分析程序，以便对任意输入的符号串进行分析。

二、实验预习提示

1、递归下降分析法的功能

词法分析器的功能是利用函数之间的递归调用模拟语法树自上而下的构造过程。

2、递归下降分析法的前提

改造文法：消除二义性、消除左递归、提取左因子，判断是否为 LL（1）文法，

3． 递归下降分析法实验设计思想及算法

为每一个非终结符设计一个识别的子程序，寻找该非终结符也就是调用相应的子程序，由于单词在语法分析中作为一个整体，故在语法识别中仅适用期内码。在这里将词法分析作为语法分析的一个子程序，当语法分析需要单词时，就调用相应的词法分析程序获得一个单词。

语法分析的作用是识别输入符号串是否是文法上定义的句子，即判断输入符号串是否满足“程序”的要求。也就是当语法识别程序从正常退出表现输入符号串是正确的“程序”。

例如：为 G 的每个非终结符号 U 构造一个递归过程,不妨命名为 U。

U 的产生式的右边指出这个过程的代码结构：

(1)若是终结符号，则和向前看符号对照，若匹配则向前进一个符号；否则出错。

(2)若是非终结符号，则调用与此非终结符对应的过程。当 A 的右部有多个产生式时,可用选择结构实现。

具体为：

（1）对于每个非终结符号 U->u1|u2|…|un 处理的方法如下：

```
U( )
{
ch=当前符号;
```

if(ch 可能是 u1 字的开头) 处理 u1 的程序部分;
else if(ch 可能是 u2 字的开头)处理 u2 的程序部分;
…
else error()
}
（2）对于每个右部 u1->$x_1x_2…x_n$ 的处理架构如下：
处理 x_1 的程序；
处理 x_2 的程序；
…
处理 x_n 的程序；
（3）如果右部为空，则不处理。
（4）对于右部中的每个符号 x_i
① 如果 xi 为终结符号：
if(xi= = 当前的符号)
{
NextChar()；
return;
}
else
出错处理
② 如果 xi 为非终结符号，直接调用相应的过程 xi()
说明： NextChar 为前进一个字符函数。

三、实验过程和指导：

（一）准备：
1.阅读课本有关章节，
2.考虑好设计方案；
3.设计出模块结构、测试数据，初步编制好程序。
（二）上课上机：
将源代码拷贝到机上调试，发现错误，再修改完善。第二次上机调试通过。
（三）程序要求：
程序输入/输出示例：
对下列文法，用递归下降分析法对任意输入的符号串进行分析：
（1）E->TG

（2）G->+TG|—TG

（3）G-> ε

（4）T->FS

（5）S->*FS|/FS

（6）S-> ε

（7）F->(E)

（8）F->i

输出的格式如下：
(1)递归下降分析程序，编制人：姓名，学号，班级
(2)输入一以#结束的符号串(包括+—*/（）i#)：在此位置输入符号串例如：i+i*i#
(3)输出结果：i+i*i#为合法符号串
备注：输入一符号串如 i+i*#,要求输出为“非法的符号串”。
引用也要改变)。
注意：1.表达式中允许使用运算符（+-*/)、分割符（括号)、字符 I，结束符#;
2.如果遇到错误的表达式，应输出错误提示信息（该信息越详细越好);
3.对学有余力的同学，可以详细的输出推导的过程，即详细列出每一步使用的产生式。

（四）程序思路（仅供参考）：

0.定义部分：定义常量、变量、数据结构。
1.初始化：从文件将输入符号串输入到字符缓冲区中。
2.利用递归下降分析法分析，对每个非终结符编写函数，在主函数中调用文法开始符号的函数。

（五）练习该实验的目的和思路：

程序开始变得复杂起来，需要利用到程序设计语言的知识和大量编程技巧，递归下降分析法是一种较实用的分析法，通过这个练习可大大提高软件开发能力。通过练习，掌握函数间相互调用的方法。
（六）为了能设计好程序，注意以下事情：
1.模块设计：将程序分成合理的多个模块（函数)，每个模块做具体的同一事情。
2.写出（画出）设计方案：模块关系简图、流程图、全局变量、函数接口等。
3.编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

(六)实验样例

输入符号串
- 合法
- 非法
  - 符号不匹配
  - 括号不匹配

1. 测试 i+i#

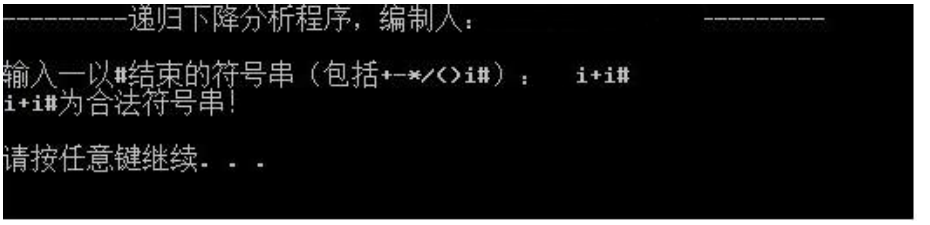


图 2.1

2. 测试 i-i#

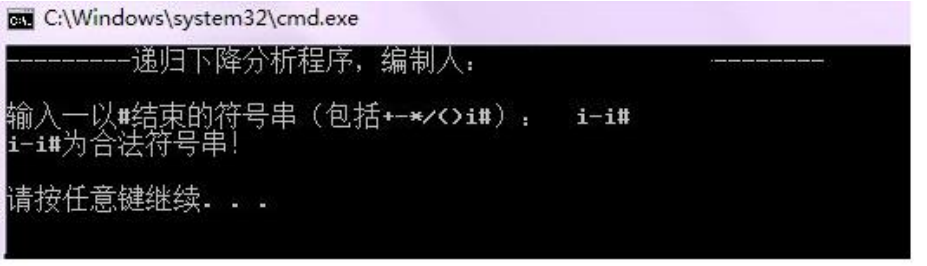


图 2.2

3. 测试 i*i#

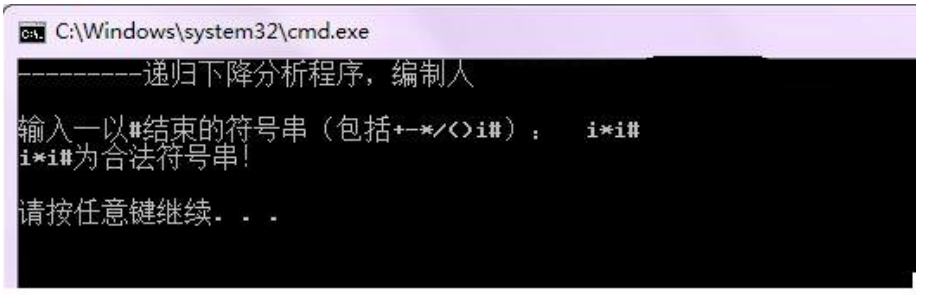


图 2.3

4. 测试 i/i#

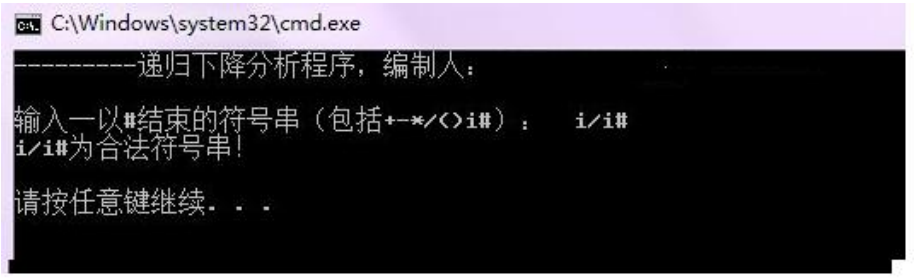


图 2.4

5. 测试 i+i*i#

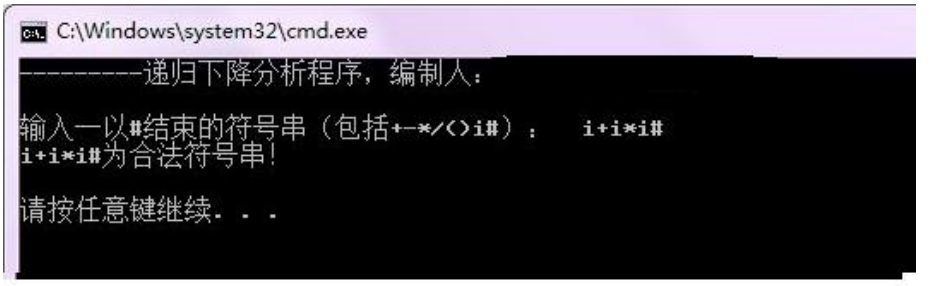


图 2.5

实验三：语法分析法 2-预测分析法

一、　　实验目的和内容：

1. 实验目的：通过完成预测分析的语法分析程序，了解预测分析法和递归子程序法的区别和联系。
2. 实验内容：构造预测分析程序, 并利用分析表和一个栈来实现对上程序设计语言的分析程序。
3. 实验要求：根据某一文法编制调试 LL（1）分析程序，以便对任意输入的符号串进行分析。本次实验的目的主要是加深对预测分析 LL（1）分析法的理解。

二、实验预习提示

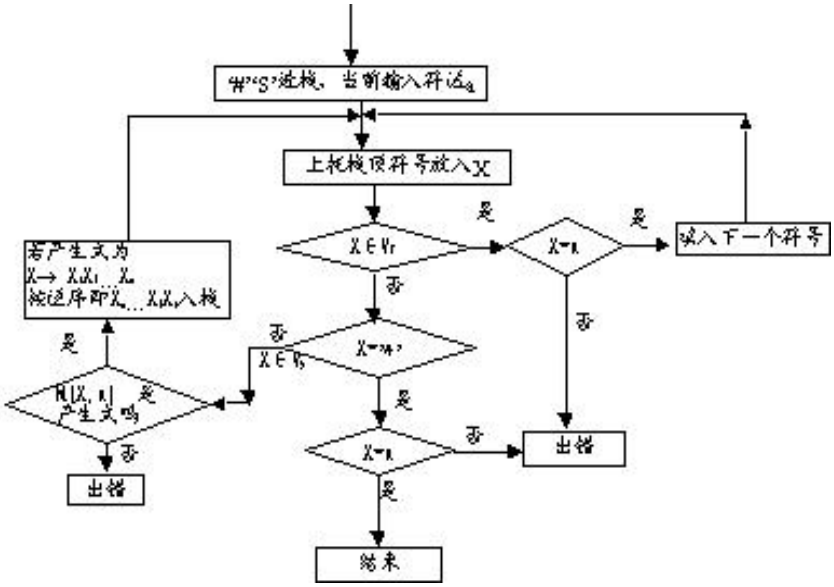
1、LL（1）分析法的功能

LL（1）分析法的功能是利用 LL（1）控制程序根据显示栈栈顶内容、向前看符号以及 LL（1）分析表，对输入符号串自上而下的分析过程。

2、LL（1）分析法的前提

改造文法：消除二义性、消除左递归、提取左因子，判断是否为 LL（1）文法，

3、LL（1）分析法实验设计思想及算法



三、实验过程和指导：

（一）准备：

1.阅读课本有关章节，

2.考虑好设计方案；

3.设计出模块结构、测试数据，初步编制好程序。

（二）上课上机：

将源代码拷贝到机上调试，发现错误，再修改完善。第二次上机调试通过。

（三）程序要求：

程序输入/输出示例：

对下列文法，用 LL（1）分析法对任意输入的符号串进行分析：

（1）E->TG

（2）G->+TG|—TG

（3）G-> ε

（4）T->FS

（5）S->*FS|/FS

（6）S-> ε

（7）F->(E)

（8）F->i

输出的格式如下：

(1)LL（1）分析程序，编制人：姓名，学号，班级

(2)输入一以#结束的符号串(包括+—*/（）i#)：在此位置输入符号串

(3)输出过程如下：

步骤	分析栈	剩余输入串	所用产生式
1	E	i+i*i#	E->TG

(4)输入符号串为非法符号串(或者为合法符号串)

备注：(1)在“**所用产生式**”一列中如果对应有推导则写出所用产生式；如果为匹配终结符则写明匹配的终结符；如分析异常出错则写为“分析出错”；若成功结束则写为“分析成功”。

(2) 在此位置输入符号串为用户自行输入的符号串。

(3)上述描述的输出过程只是其中一部分的。

注意：1.表达式中允许使用运算符（+-*/）、分割符（括号）、字符 i，结束符#；
2.如果遇到错误的表达式，应输出错误提示信息（该信息越详细越好）；
3.对学有余力的同学，测试用的表达式事先放在文本文件中，一行存放一个表达式，同时以分号分割。同时将预期的输出结果写在另一个文本文件中，以便和输出进行对照；

（四）程序思路（仅供参考）：
模块结构：
（1）定义部分：定义常量、变量、数据结构。
（2）初始化：设立 LL(1)分析表、初始化变量空间（包括堆栈、结构体、数组、临时变量等）；
（3）控制部分：从键盘输入一个表达式符号串；
（4）利用 LL(1)分析算法进行表达式处理：根据 LL(1)分析表对表达式符号串进行堆栈（或其他）操作，输出分析结果，如果遇到错误则显示错误信息。

（五）练习该实验的目的和思路：

程序相当复杂，需要利用到大量的编译原理，也用到了大量编程技巧和数据结构，通过这个练习可大大提高软件开发能力。

（六）为了能设计好程序，注意以下事情：
1.模块设计：将程序分成合理的多个模块（函数），每个模块做具体的同一事情。
2.写出（画出）设计方案：模块关系简图、流程图、全局变量、函数接口等。
3.编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

（七）基于面向对象案例

一、预测分析器

1、打开主见面，根据预测语法分析器 LL 分析合法字符串，在请输入字符串位置输入文法，并且根据预测语法分析器 LL 分析输入字符串 i+i*i#的分析过程，在分析过程中，可以看到分析栈中入栈字符，剩余输入串还没有识别的字符，推到产生式或匹配中反映的是每一个入栈字符的状态，并且在分析结果中反映出识别的结果。

图 3.1 预测分析器 LL

2、在这个页面中，在同样的文法中分析不合法字符，如果输入字符串 i+(i*i# 不符合文法规则字符串时，利用预测语法分析器 LL 具体分析此文法的分析过程，并报出分析结果为此字符串为非法字符串。

图 3.2 不符合文法规则的分析结果

3、通过 LR 文法检测语法分析，出现语法分析的界面。

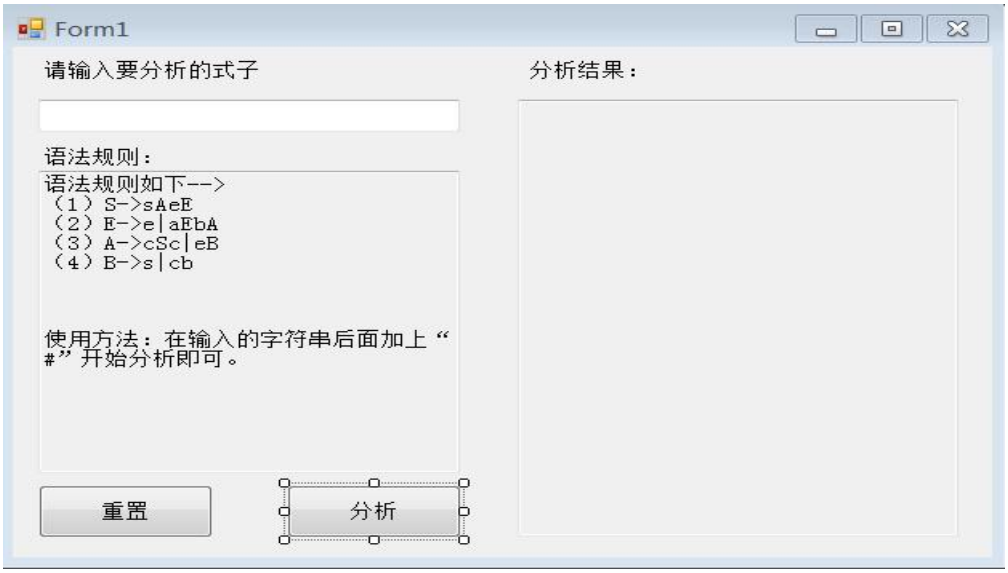


图 3.3 改变文法规则的见面图

4、如果输入字符串 sesee#不符合文法规则，在语法分析器具体分析此文法的分析过程，并报出分析结果，并判断是否是合法字符。

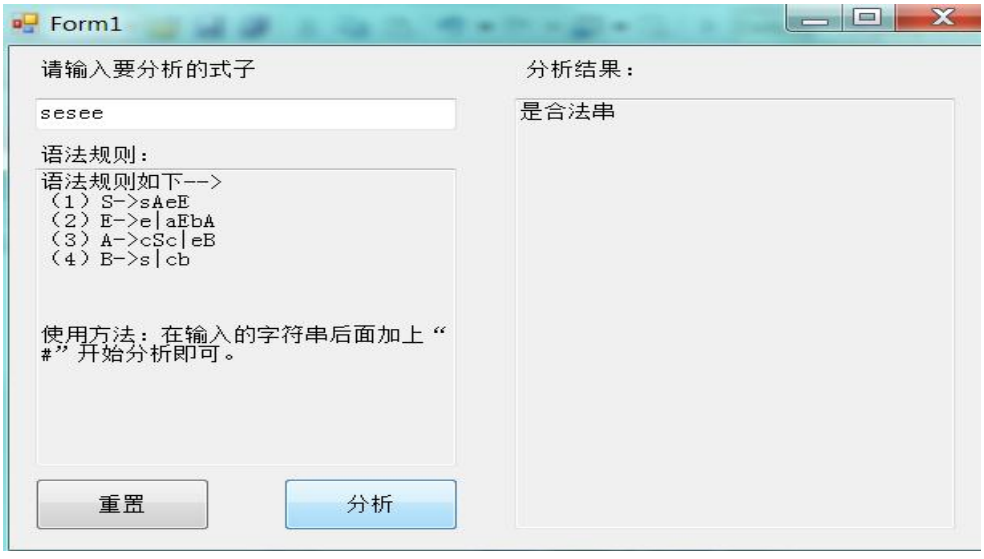


图 3.4 根据不同文法规则识别合法字符串

5、如果输入字符串 seabb#不符合文法规则，在语法分析器具体分析此文法的分析过程，并报出分析结果，并判断是非合法字符。

图 3.5 根据不同文法规则识别不合法字符串

二、操作步骤总结

本实验在开发环境 Visual Studio 2010 中，利用语言 C#开发预测分析器，C#简单、功能强大、类型安全，而且是面向对象的,C#凭借在许多方面的创新，在保持 C 语言风格的表现力和雅致特征的同时，实现了应用程序的快速开发。利用 C#面向对象程序设计语言实现传统软件编译器过程，实现预测分析语法分析器动态实现的动态过程。

1） 用 C#面向对象程序设计语言，搭建界面，实现可视化窗口。

2） 通过定义部分，定义常量、变量、数据结构。

3） 开始初始化建立 LL(1)语法分析表，初始化变量空间，包括堆栈、结构体、数组、临时变量。

4） 在可视化窗口输入界面，读入要识别的合法字符串，根据算法分析，得出合法字符的文法规则，并输出文法规则分析过程。

5） 只需输入一串字符并在输入的式子后加上#号，调用分析函数开始分析，并将分析结果输入窗口中显示。其中 judge 为布尔类型的数据，用来输出被分析的式子是否合法，E 为式子的分析方法

6） 在可视化窗口输入界面，读入要识别的不合法字符串，根据算法分析，得出不合法字符的文法规则，并输出不合文法结论，利用 LL(1)分析算法

进行表达式处理，根据 LL(1)分析表对表达式符号串进行堆栈或其他，操作输出分析结果，如果遇到错误则显示错误信息。

7） 根据语法分析器进一步分析，不同字符识别过程。

实验四：语法分析 3-LR(1)分析程序

一。实验目的和内容：

1. 实验目的：通过完成预测分析的语法分析程序，了解预测分析法和递归子程序法的区别和联系。
2. 实验内容：构造预测分析程序,并利用分析表和一个栈来实现对上程序设计语言的分析程序。
3. 实验要求：根据某一文法编制调试 LR（1）分析程序，以便对任意输入的符号串进行

二、实验预习提示：

1、使用 LR(1)的优点：

(1)LR 分析器能够构造来识别所有能用上下文无关文法写的程序设计语言的结构。

(2)LR 分析方法是已知的最一般的无回溯移进-归约方法,它能够和其他移进-归约方法一样有效地实现。

(3)LR 方法能分析的文法类是预测分析法能分析的文法类的真超集。

(4)LR 分析器能及时察觉语法错误,快到自左向右扫描输入的最大可能。

为了使一个文法是 LR 的，只要保证当句柄出现在栈顶时，自左向右扫描的移进-归约分析器能够及时识别它便足够了。当句柄出现在栈顶时，LR 分析器必须要扫描整个栈就可以知道这一点,栈顶的状态符号包含了所需要的一切信息。如果仅知道栈内的文法符号就能确定栈顶是什么句柄。LR 分析表的转移函数本质上就是这样的有限自动机。不过，这个有限自动机不需要根据每步动作读栈，因为，如果这个识别句柄的有限自动机自底向上读栈中的文法符号的话，它达到的状态正是这时栈顶的状态符号所表示的状态，所以，LR 分析器可以从栈顶的状态确定它需要从栈中了解的一切。

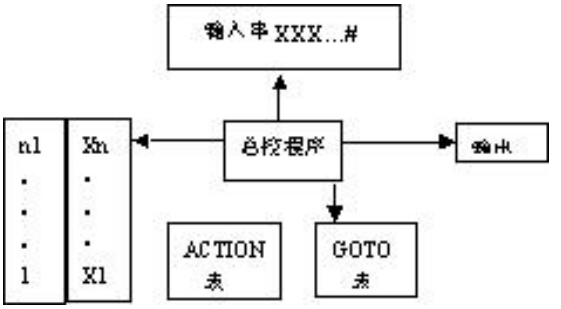
2、LR 分析器由三个部分组成：

(1)总控程序，也可以称为驱动程序。对所有的 LR 分析器总控程序都是相同的。

(2)分析表或分析函数，不同的文法分析表将不同，同一个文法采用的 LR 分析器不同时,分析表将不同,分析表又可以分为动作表（ACTION）和状态转换（GOTO）表两个部分，它们都可用二维数组表示。

(3)分析栈，包括文法符号栈和相应的状态栈，它们均是先进后出栈。分析器的动作就是由栈顶状态和当前输入符号所决定。

LR 分析器结构：



其中:SP 为栈指针，S[i]为状态栈，X[i]为文法符号栈。状态转换表用 GOTO[i，X]=j 表示，规定当栈顶状态为 i，遇到当前文法符号为 X 时应转向状态 j，X 为终结符或非终结符。

ACTION[i，a]规定了栈顶状态为 i 时遇到输入符号 a 应执行。动作有四种可能：

(1)移进：

action[i，a]= Sj：状态 j 移入到状态栈，把 a 移入到文法符号栈，其中 i,j 表示状态号。

(2)归约：

action[i，a]=r_k：当在栈顶形成句柄时，则归约为相应的非终结符 A，即文法中有 A->B 的产生式，若 B 的长度为 R(即|B|=R)，则从状态栈和文法符号栈中自顶向下去掉 R 个符号，即栈指针 SP 减去 R，并把 A 移入文法符号栈内，j=GOTO[i,A]移进状态栈，其中 i 为修改指针后的栈顶状态。

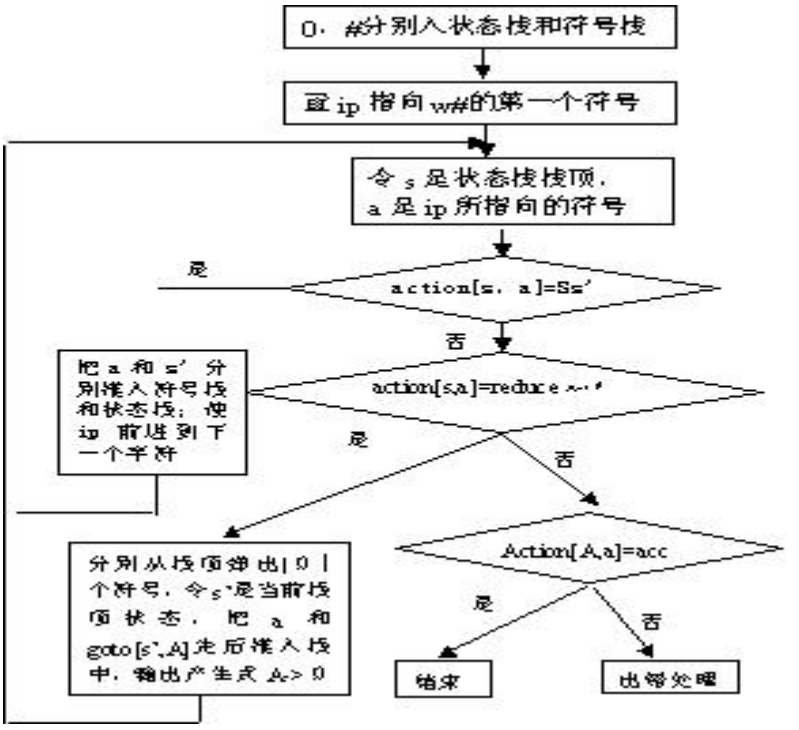
(3)接受 acc:

当归约到文法符号栈中只剩文法的开始符号 S 时，并且输入符号串已结束即当前输入符是'#'，则为分析成功。

(4)报错:

当遇到状态栈顶为某一状态下出现不该遇到的文法符号时，则报错，说明输入端不是该文法能接受的符号串。

3、LR（1）分析法实验设计思想及算法



三、实验过程和指导：

（一）准备：

1.阅读课本有关章节，

2.考虑好设计方案；

3.设计出模块结构、测试数据，初步编制好程序。

（二）上课上机：

将源代码拷贝到机上调试，发现错误，再修改完善。

（三）程序要求：

程序输入/输出示例：

对下列文法，用 LR（1）分析法对任意输入的符号串进行分析：

（1）E->E+T

（2）E->E—T

（3）T->T*F

（4）T->T/F

（5）F->(E)

（6）F->i

输出的格式如下：
(1)LR（1）分析程序，编制人：姓名，学号，班级
(2)输入一以#结束的符号串(包括+—*/（）i#)：在此位置输入符号串
(3)输出过程如下：

步骤	状态栈	符号栈	剩余输入串	动作
1	0	#	i+i*i#	移进

(4)输入符号串为非法符号串(或者为合法符号串)

备注：(1)在“**所用产生式**”一列中如果对应有推导则写出所用产生式；如果为匹配终结符则写明匹配的终结符；如分析异常出错则写为“分析出错”；若成功结束则写为“分析成功”。

(2) 在此位置输入符号串为用户自行输入的符号串。

注意：1.表达式中允许使用运算符（+-*/）、分割符（括号）、字符 i，结束符#；

2.如果遇到错误的表达式，应输出错误提示信息（该信息越详细越好）；

3.对学有余力的同学，测试用的表达式事先放在文本文件中，一行存放一个表达式，同时以分号分割。同时将预期的输出结果写在另一个文本文件中，以便和输出进行对照；

（四）程序思路（仅供参考）：

模块结构：
（1）定义部分：定义常量、变量、数据结构。

（2）初始化：设立 LR(1)分析表、初始化变量空间（包括堆栈、结构体、数组、临时变量等）；
（3）控制部分：从键盘输入一个表达式符号串；
（4）利用 LR(1)分析算法进行表达式处理：根据 LR(1)分析表对表达式符号串进行堆栈（或其他）操作，输出分析结果，如果遇到错误则显示错误信息。
（五）练习该实验的目的和思路：

程序相当复杂，需要利用到大量的编译原理，也用到了大量编程技巧和数据结构，通过这个练习可大大提高软件开发能力。

（六）为了能设计好程序，注意以下事情：
1.模块设计：将程序分成合理的多个模块（函数），每个模块做具体的同一事情。
2.写出（画出）设计方案：模块关系简图、流程图、全局变量、函数接口等。
3.编程时注意编程风格：空行的使用、注释的使用、缩进的使用

第五章：中间代码生成

一、实验目的和内容

1. 实验目的：通过完成逆波兰式分析程序，了解中间代码的生成。
2. 实验内容：翻译程序为逆波兰式形式。
3. 实验要求：将非后缀式用来表示的算术表达式转换为用逆波兰式来表示的算术表达式，并计算用逆波兰式来表示的算术表达式的值。

二、实验预习提示

1、逆波兰式定义

将运算对象写在前面，而把运算符号写在后面。用这种表示法表示的表达式也称做后缀式。逆波兰式的特点在于运算对象顺序不变，运算符号位置反映运算顺序。采用逆波兰式可以很好的表示简单算术表达式，其优点在于易于计算机处理表达式。

2、产生逆波兰式的前提

中缀算术表达式

3、逆波兰式生成的实验设计思想及算法

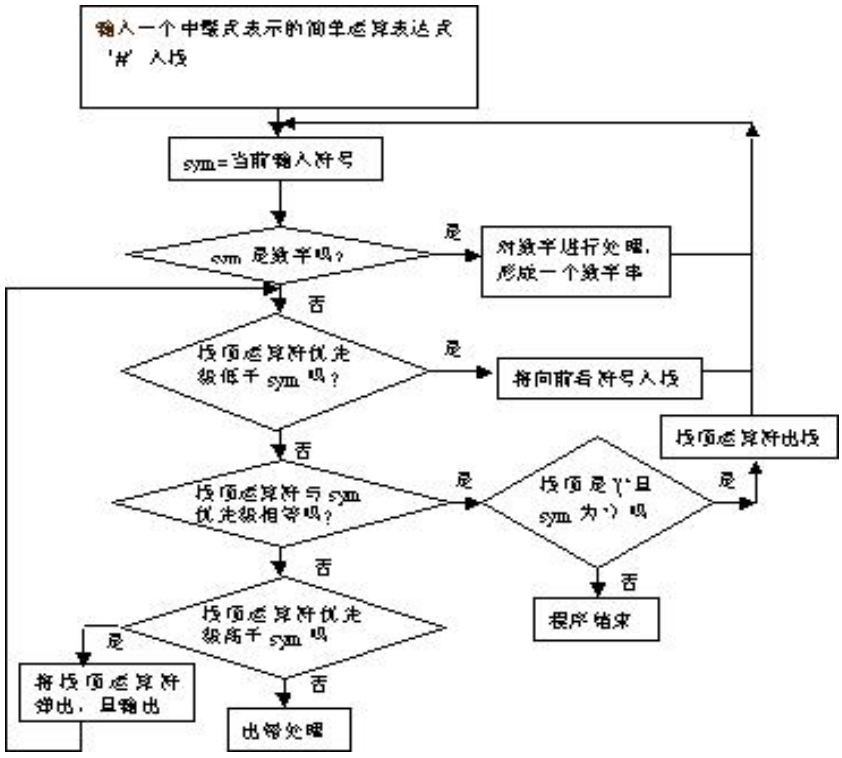


图 5.1 流程图

(1)首先构造一个运算符栈，此运算符在栈内遵循越往栈顶优先级越高的原则。

(2)读入一个用中缀表示的简单算术表达式，为方便起见,设该简单算术表达式的右端多加上了优先级最低的特殊符号“#”。

(3)从左至右扫描该算术表达式，从第一个字符开始判断，如果该字符是数字，则分析到该数字串的结束并将该数字串直接输出。

(4)如果不是数字，该字符则是运算符，此时需比较优先关系。

做法如下：将该字符与运算符栈顶的运算符的优先关系相比较。如果，该字符优先关系高于此运算符栈顶的运算符，则将该运算符入栈。倘若不是的话，则将此运算符栈顶的运算符从栈中弹出，将该字符入栈。

(5)重复上述操作(1)-(2)直至扫描完整个简单算术表达式，确定所有字符都得到正确处理，我们便可以将中缀式表示的简单算术表达式转化为逆波兰表示的简单算术表达式。

3、逆波兰式计算的实验设计思想及算法

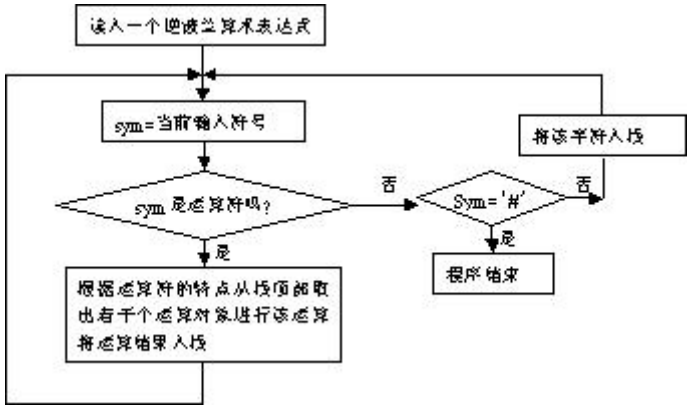


图 5.2　流程图

(1)构造一个栈，存放运算对象。

(2)读入一个用逆波兰式表示的简单算术表达式。

(3)自左至右扫描该简单算术表达式并判断该字符，如果该字符是运算对象，则将该字符入栈。若是运算符，如果此运算符是二目运算符，则将对栈顶部的两个运算对象进行该运算，将运算结果入栈，并且将执行该运算的两个运算对象从栈顶弹出。如果该字符是一目运算符，则对栈顶部的元素实施该运算，将该栈顶部的元素弹出，将运算结果入栈。

(4)重复上述操作直至扫描完整个简单算术表达式的逆波兰式，确定所有字符都得到正确处理，我们便可以求出该简单算术表达式的值。

三、实验过程和指导：

（一）准备：

1.阅读课本有关章节，

2.考虑好设计方案；

3.设计出模块结构、测试数据，初步编制好程序。

（二）上课上机：

将源代码拷贝到机上调试，发现错误，再修改完善。第二次上机调试通过。

（三）程序要求：

程序输入/输出示例：

输出的格式如下：

(1)逆波兰式的生成及计算程序，编制人：姓名，学号，班级

(2)输入一以#结束的中缀表达式(包括+—*/（）数字#)：在此位置输入符号串如(28+68)*2#

(3)逆波兰式为：28&68+2*

(4)逆波兰式 28&68+2*计算结果为 192

备注：(1)在生成的逆波兰式中如果两个数相连则用&分隔，如 28 和 68，中间用&分隔；

(2)在此位置输入符号串为用户自行输入的符号串。

注意：1.表达式中允许使用运算符（+-*/）、分割符（括号）、数字，结束符#；

2.如果遇到错误的表达式，应输出错误提示信息（该信息越详细越好）；

3.对学有余力的同学，测试用的表达式事先放在文本文件中，一行存放一个表达式，同时以分号分割。同时将预期的输出结果写在另一个文本文件中，以便和输出进行对照；

（四）程序思路（仅供参考）：

模块结构：

（1）定义部分：定义常量、变量、数据结构。

（2）初始化：设立算符优先分析表、初始化变量空间（包括堆栈、结构体、数组、临时变量等）；

（3）控制部分：从键盘输入一个表达式符号串；

（4）利用算符优先分析算法进行表达式处理：根据算符优先分析表对表达式符号串进行堆栈（或其他）操作，输出分析结果，如果遇到错误则显示错误信息。

（5）对生成的逆波兰式进行计算。

（五）练习该实验的目的和思路：

程序较复杂，需要利用到程序设计语言的知识和大量编程技巧，逆波兰式的生成是算符优先分析法的应用，是一种较实用的分析法，通过这个练习可大大提高软件开发能力。

（六）为了能设计好程序，注意以下事情：

1.模块设计：将程序分成合理的多个模块（函数），每个模块做具体的同一事情。

2.写出（画出）设计方案：模块关系简图、流程图、全局变量、函数接口等。

3.编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

（六）面向对象开发样例

1. 未输入中缀表达式

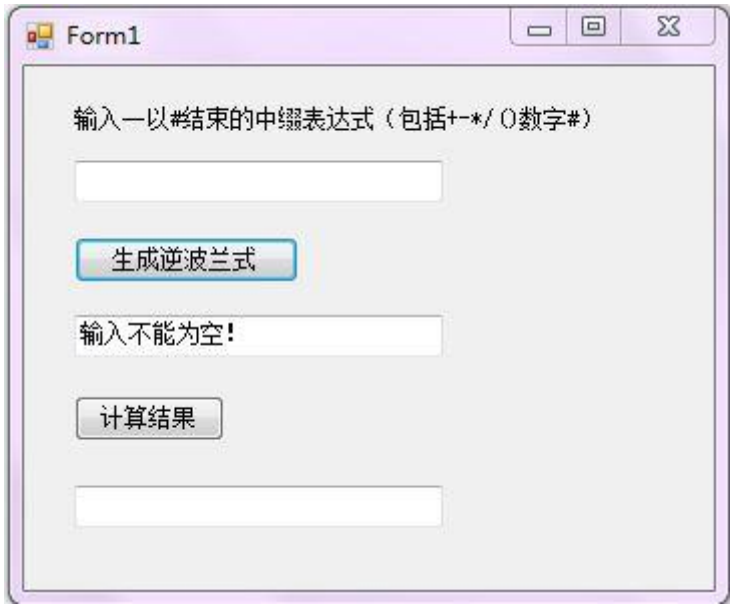


图 5.3 中缀表达式

2. 分析出错时，“计算结果”控件设置为不可用的

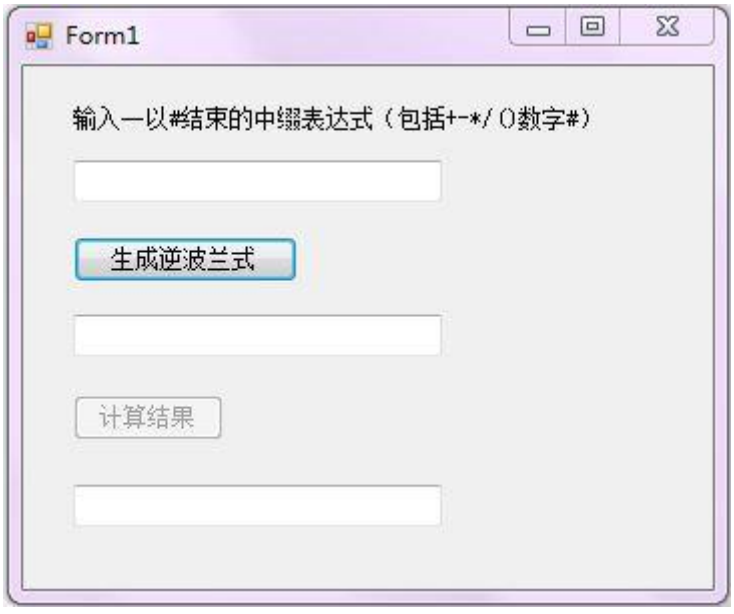


图 5.4 计算结果

3. 中缀式输入错误

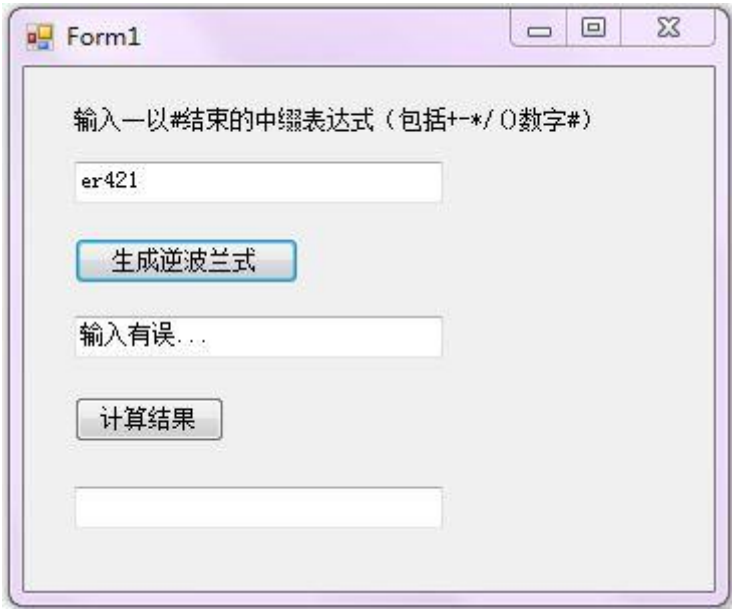


图 5.5 计算结果

4. 正确结果

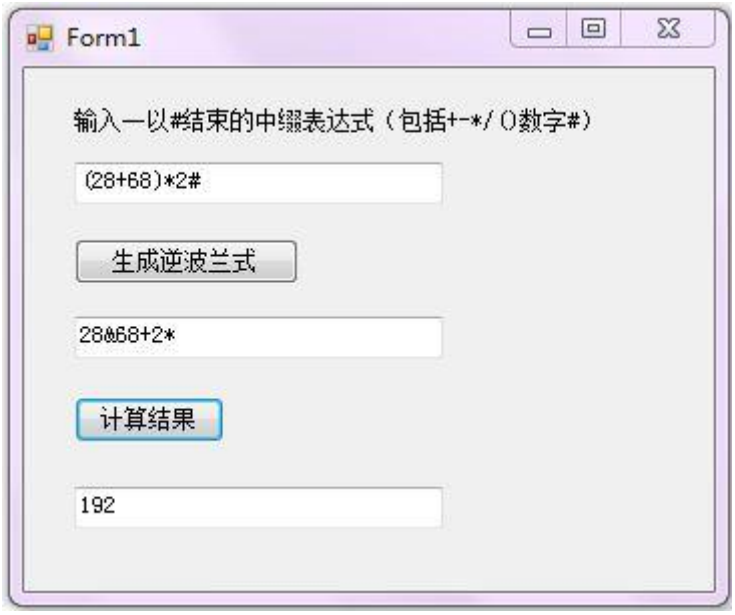


图 5.6 计算结果

第二部分 基于 Linux 系统

试验一： Linux 了解和使用

1.1Linux 启动和简单使用

打开电脑，选择系统：Linux。等待片刻。出现如下图 1_1.

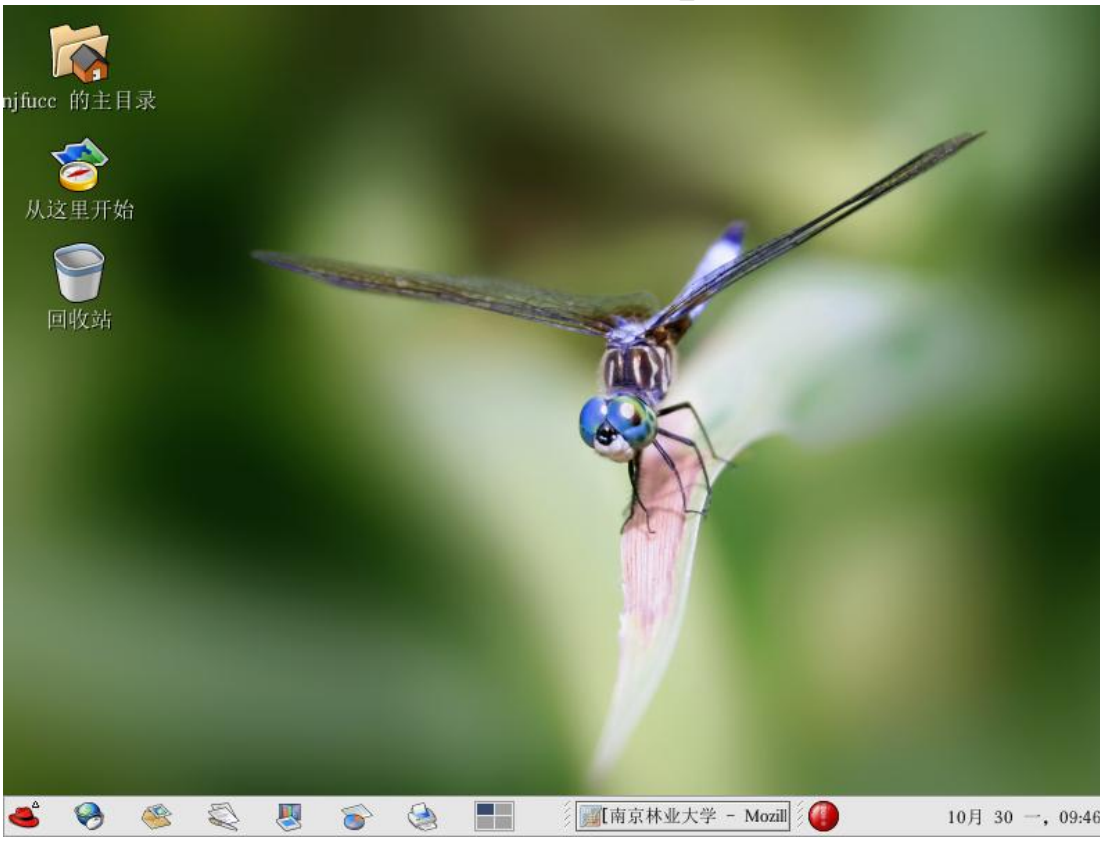


图 1_3 Linux 系统界面

在桌面上单击鼠标邮件，出现快捷菜单上，选择打开终端，出现如下图 1_3，[njfucc@localhost njfucc]$为终端接受符。在$后面输入各种命令。例如 Vi test 出现的界面如图 1_3.

图 1_2. 打开终端

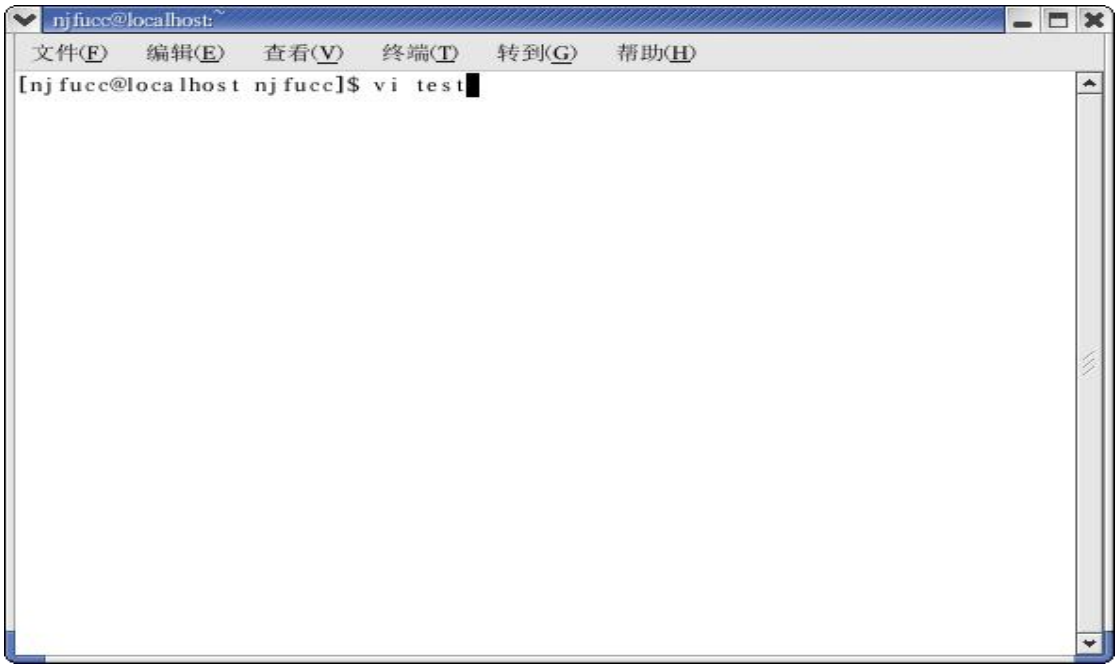


图 1_3. 打开 VI 编辑器，并且输入命令

图 1_4.VI 编辑器界面

接下来程序输入和编辑，可以查看本书的后面章节。

图 1_5.应用程序界面

1.2Linux Shell 简介

1.什么是 linux shell?

linux shell 其实就是核心程序(kernel)之外的一个命令解析器，它是一个程序(说白了就是基于 kernel 的一个进程) ,同时也是一种命令语言和程序设计语言。作为命令语言可以交互式的解析执行用户输入的命令;

作为程序设计语言它定义了各种变量和参数,并且提供了许多在高级语言里才具有的控制结构。他虽然不是 linux 系统核心的一部分,但他调用了系

统核心的大部分功能来执行程序、建立文件,并以并行的方式来协调各个程序运行。

可以说 shell 对用户来讲是最重要的一个使用程序 ,可以说 shell 的使用程度，反映用户对 linux 的使用程度。

2.系统命令的执行:

当我们登陆一个 linux 系统的时候系统初始化程序就为每一个用户运行了一 shell 程序 ,通过这个 shell 来解析用户输入的命令 ,比喻我们输"ls",shell 就会解析"ls"字符为一个命令，同时向内核发送请求这个命令，内核执行这个命令后就会把返回的结果告诉 shell,shell 把结果呈现给用户。

shell 类似于 dos 时代的 command.com(负责解析 dos 下输入的命令),不同的是在 dos 下面 command.com 只有一个，而在 linux 下面这样的解析器可以有很多个。可以通过察看/etc/shells 这个文件来察看所登陆的系统中有哪些 shell:

cat /etc/shell

怎样知道当前运行的是哪一个 shlle,可以运行命令 echo $SHELL.

3.bash shell 简介:

现在这个使用得比较多的 shell，它有哪些好优点呢？

3.1.它有类似 dos 下的 doskey 的功能，可以按上下箭头来找到以前用过

的命令

3.2.还有就是可以查找命令，比喻我们要输入 cat,但是不记得 cat 只知道是以 ca 开头 ,我们就可以输入 ca 然后按两次 Tab 键他就会返回以 ca 开头的命令。这样我们就不用大量记忆命令。

3.3.就是帮助功能,输入 help 就会返回 bash 下内置的 40 几个常用非常重要命令。如果要查询某个命令怎么用 ,比喻要查询 cat ,我们可以 man cat 或者 info cat 查看帮助信息 ,要退出察看的信息按键 q。至于其他的几个 shell 环境的介绍,去查资料,我认为把常用的弄清了就可以了,反正都差不多的东西。

4.shell 脚本

4.1.shell 脚本的基本组成

#!/bin/bash (这是第一行这里是 bash 为主)

(除了第一行，其它的以#开头的行都为注释)

变量什么的

流程控制

4.2.一个简单的脚本例子

这里就来个传说中的 hello world 了。保存为 hello.sh

```
#!/bin/bash

#This my hello world shell program

heihei=”hello world”

echo $heihei
```

这里注意上面的脚本你必须保证有可执行权限，也就是 chmod u+x hello.sh

5.关于 shell 下的别名

比喻我们想让 ls 显示文件时带上颜色，我们可以自己定一个命令

```
alias dir=’ls –color’
```

这样我们用 dir 来查看文件就相当于执行了 ls –color。

6。Linux Shell 的简单使用（在试验二中详细介绍）

1.3 Linux 的有关文件和目录的命令

Linux 文件系统是呈树形结构，了解 Linux 文件系统的目录结构，对于我们驾驭 Linux 还是有必要的。本文对 Linux 下比较重要的目录加以解说，以答初学者所说的“杂乱无章”目录结构，给一个简要的说明。

Linux 的有关文件和目录的命令

/	Linux 系统根目录
/bin	Binary 的缩写，存放用户的可执行程序，例如 ls,cp,也包含其它的 SHELLR 如：bash 等
/boot	包含 vmlinuz,initrd.img 等启动文件，随便改动可能无法正常

	开机哦
/dev	接口设备文件目录，如你的硬盘：hda
/etc	passwd 这样有关系统设置与管理的文件
/etc/x11	X Windows System 的设置目录
/home	一般用户的主目录，如 FTP 目录等
/lib	包含执行/bin 和/sbin 目录的二进制文件时所需的共享函数库 library
/mnt	各项装置的文件系统加载点，例如：/mnt/cdrom 是光驱的加载点
/opt	提供空间，叫较大的且固定的应用程序存储文件之用
/proc	PS 命令查询的信息与这里的相同，都是系统内核与程序执行的信息
/root	管理员的主目录
/sbin	lilo 等系统启动时所需的二进制程序
/tmp	Temporary,存放暂存盘的目录
/usr	存放用户使用系统命令和应用程序等信息
/usr/bin	存放用户可执行程序，如 grep,mdir 等
/usr/doc	存放各式程序文件的目录
/usr/include	保存提供 C 语言加载的 header 文件
/usr/include/X11	保存提供 X Windows 程序加载的 header 文件
/usr/info	GNU 程序文件目录
/usr/lib	函数库

/usr/lib/X11	函数库
/usr/local	提供自行安装的应用程序位置
/usr/man	存放在线说明文件目录
/usr/sbin	存放经常使用的程序，如 showmount
/usr/src	保存程序的原始文件
/usr/X11R6/bin	存放 X Windows System 的执行程序
/var	Variable,具有变动性质的相关程序目录，如 log

1.4Linux 的系统对文件和目录的操作命令

1. 显示文件内容

使用过 DOS 命令的人都应该知道，我们可以使用 type 命令来查看一个文件的内容。在 Linux 下有五个相关的命令，功能各有千秋，不过它们都象 type 命令一样，只能用来查看文本文件。

cat 命令

cat 命令是最象 type 命令的，使用的方法很简单：“cat 文件名”。不过比 type 命令更强大的是，它可以同时查看多个文件：“cat 文件名一 文 件名二”。

(2) more 命令

如果文本文件比较长，一屏无法显示完，那么使用 cat 命令就可能无法看清。这里我们可以简单地使用 more 来代替 cat 命令即可。其效果与 type 文件名/p 类似。使用 more 命令将一次显示一屏文本，显示满后，停下来，并提示出已显示全部内容的百分比，按空格键就可以看到下一屏。

(3) less 命令

less 命令的功能几乎和 more 命令一样，也是按页显示文件，不同之处在于 less 命令在显示文件时允许用户既可以向前又可以向后翻阅文件。

向前翻：按 b 键；向后翻：按 p 键；指定位置：输入百分比；退出：q

(4) head 命令

通过 head 命令可以仅查看某文件的前几行，格式为： head 行数 文件名 如果未指定行数，则使用默认值 10。

(5) tail 命令

与 head 命令相对应的，我们可以使用 tail 命令来查看文件尾部的内容。通常用来实时监测某个文件是否被修改，通常用来观察日志。如： tail -f maillog

2. 编辑文件

在 Red Hat Linux 7 中有许多文字编辑工具，其中最常用的应该是 vi，这是一个广泛应用于所有 UNIX 系统的编辑器。它的使用有些特别：首先，可以使用命令“vi 文件名”打开一个文件。

刚启动的时候，vi 处于命令状态，不能够输入任何字符。在这个状态下，可以使用方向键进行移动，而需要输入内容时，你需要输 入“i”或“a”命令进入编辑状态。编辑完成后，你需要按下“ESC”键回到命令状态。

在命令状态下，你可以输入“:q!”不存盘退出，输入“:wq”存盘退出。

3. 文件的复制、删除与移动

大家都早已熟知在 DOS 下我们可以使用 copy、del、move 命令来实现文件的复制、删除与移动。下面我们说说如何在 Linux 系统中做以上操作。

1) cp 命令：文件/目录复制命令

它的语法格式为：cp [选项] 源文件或目录 目标文件或目录

常用的选项有：

a 该选项常在复制目录时使用，它保留链接、文件属性，并递归地复制目录，

就象 DOS 中的 xcopy /s 一样

f 如果目标文件或目录已存在，就覆盖它，并且不做提示

i 与 f 选项正好相反，它在覆盖时，会让用户回答“Y”来确认

p 使用该选项，复制文件时将保留修改时间和访问权限

r 若给出的源是一个目录，那么 cp 将递归复制该目录下所有的子目录和文件，不过这要求目标也是一个目录名

另外，大家要注意的是，如果源是文件名，目标是目录名的话，那么使用 cp 命令可以指定多个源文件名。如：

$ cp a.txt b.txt /home/user1

该命令将把 a.txt 和 b.txt 文件复制到/home/usr1 目录中。

2) rm 命令：文件/目录删除命令

它的语法格式为：rm [选项] 文件 …

常用的选项有：

f 在删除过程中不给任何指示，直接删除

r 指示 rm 将参数中列出的全部目录和子目录都递归地删除

i 交互式的删除，每个文件在删除时都给出提示

使用 rm 命令时一定要小心，特别是以 root 用户登录时，我就看到过一个朋友在使用 rm 命令删除/home/tmp 目录时将命令“rm -rf /home/tmp”误输成了 “rm -rf / home/tmp”，结果等他走回电脑面前，整个系统都被删除了！

3) mv 命令：文件/目录移动命令

它的语法格式为：mv [选项] 源文件或目录 目标文件或目录

常用的选项有：

f 如果操作要覆盖某已有的目标文件时不给任何指示

i 交互式的操作，如果操作要覆盖某已有的目标文件时会询问用户是否覆盖

mv 命令的执行效果与参数类型的不同而不同！

第一参数(源)	第二个参数(目标)	结果
文件名	文件名	将源文件名改为目标文件名
文件名	目录名	将文件移动到目标目录
目录名	目录名	目标目录已存在：源目录移动到目标目录

目标目录不存在：改名

目录名 文件名 出错

4. 目录相关操作

1) 创建新目录：mkdir，它的使用与 DOS 下的 md 相同：mkdir 目录名；

2) 删除空目录：rmdir，它的使用与 DOS 下的 rd 相同：rmdir 目录名；

3) 改变目录：cd，它的使用与 DOS 下的 cd 命令基本相同，唯一一个不同的是，不管目录名是什么，cd 与目录名之间必须有空格，也就是：“cd/”、“cd..”、“cd.”都是非法的，而应该输入：“cd /”、“cd ..”、“cd .”，如果直接输入命令“cd”，而不加任何参数，将回到这个用户的主目录。

4) 显示当前目录：pwd

5) 列目录命令：ls，相当于 DOS 下的 dir

它的语法为：ls [选项] [目录或文件]

常用的选项有：

a 显示指定目录下所有的子目录与文件，包括隐藏文件；

c 按文件的修改时间排序

l 采用长格式来显示文件的详细信息，每个文件一行信息，其内容为：文件类型与权限 链接数 文件属主 文件属组 文件大小 最近修改时间 文件名

5. 文件与目录的权限操作

在 Linux 系统中，每一个文件和目录都有相应的访问许可权限，我们可以用它来确定谁可以通过何种方式对文件和目录进行访问和操作。文件或目录的访问权限分为可读、可写和可执行三种，分别以 r，w，x 表示，其含义为：

r w x

文件 可读 可写 可执行

目录 可列出目录 可在目录中做写操作 可以访问该目录

在文件被创建时，文件所有者可以对该文件的权限进行设置。

对于一个文件来说，可以将用户分成三种，并对其分别赋予不同的权限：

1) 文件所有者

2) 与文件所有者同组用户

3) 其它用户

每一个文件或目录的访问权限都有三组，每组用三位表示，如：

d rwx r-x r--

第一部分：这里的 d 代表目录，其它的有：- 代表普通文件 c 代表字符设备文件；

第二部分：文件所有者的权限字，这里为 r w x 表示可读、可写、可执行 (目录的可执行指的可以进入目录)；

第三部分：与文件所有者同组的用户的权限字，这里为 r - x 表示可读、不可写、可执行。由于同组用户之间交流较多，让他看看文件，别乱改就 行了嘛。

第四部分：其它用户的权限字，这里为- - -，当然给我无关的人嘛，我的文件当然不但不给你写，也不让你读。

1) 文件/目录权限设置命令：chmod

这是 Linux 系统管理员最常用到的命令之一，它用于改变文件或目录的访问权限。该命令有两种用法：

用包含字母和操作符表达式的文字设定法

其语法格式为：chmod [who] [opt] [mode] 文件/目录名

其中 who 表示对象，是以下字母中的一个或组合：

u：表示文件所有者
g：表示同组用户
o：表示其它用户
a：表示所有用户

opt 则是代表操作，可以为：

+：添加某个权限

-：取消某个权限

=：赋予给定的权限，并取消原有的权限

而 mode 则代表权限：

r：可读

w：可写

x：可执行

例如：为同组用户增加对文件 a.txt 的读写权限：

chmod g+rw a.txt

用数字设定法

而数字设定法则更为简单：chmod [mode] 文件名

关键是 mode 的取值，一开始许多初学者会被搞糊涂，其实很简单，我们将 rwx 看成二进制数，如果有则有 1 表示，没有则有 0 表示，那么 rwx r-x r- -则可以表示成为：

111 101 100

再将其每三位转换成为一个十进制数，就是 754。

例如，我们想让 a.txt 这个文件的权限为：

自己 同组用户 其他用户

可读 是 是 是

可写 是 是 可执行

那么，我们先根据上表得到权限串为：rw-rw-r--，那么转换成二进制数就是 110 110 100，再每三位转换成为一个十进制数，就得到 664，因此我 们执行命令：

chmod 664 a.txt

2) 改变文件的属主命令：chown

语法格式很简单：chown [选项] 用户名 文件/目录名

其中最常用的选项是“R”，加上这个参数，可以将整个目录里的所有子目录和文件的属主都改变成指定用户。

3) 改变文件属组命令：chgrp

该命令也很简单：chgrp 组名 文件名

练习：

1. Linux 系统由那几部分组成，它们的依赖关系？
2. 简答 Linux 系统的启动过程？
3. Linux 内核的作用是什么？
4. 实际练习一下本节的命令?

试验二：Vi 编辑器

Vi（Vim）是 Linux 下最重要的文本编辑器，在 Linux 的系统管理和网络管理中，会经常使用文本编辑器进行编辑工作，因此，Vi 是进行系统和网络维护的基础。本章主要介绍如何使用 Vi 编辑器进行基本的文本编辑工作。

2.1 Vi 命令格式和操作模式

2.1.1Vi 操作模式

Vi 拥有 3 种编辑模式 :命令行模式(command mode)、输入模式(input mode) 与末行模式（ last line mode ）。以下分点说明它们的功能。

1．命令行模式（ command mode ）

任何输入都会作为编辑命令，而不会出现在屏幕上，若输入错误则有“岬”的声音；任何输入都引起立即反映 ,命令行模式主要使用方向键移动游标位置以进行文字的编辑，在输入模式下按【Esc】键或是在末行模式输入了错误命令，都会回到命令行模式，表 4-1 列出常用的操作方式。

表 2_1 Vim 命令行模式命令

操　　作	说　　明
0	游标移至行首
$	游标移至行尾
PageDn	向下滚动一页
PageUp	向上滚动一页
d+方向键	删除文字
dd	删除一行
yy	整行复制
p	粘贴复制的文字

2．输入模式（input mode）

在 Vim 下编辑文字，您并不能直接插入、替代或删除文字，而必须先进入输入模式。初学者刚开始可能会觉得不方便，但习惯之后，反而会觉得这可以避免一些输入操作上的错误，比如不小心删除了某行文字之类的操作。要进入输入模式，您可以按【a/A】键、【i/I】键或【o/O】键，它们的功能如表 2-2 所示。

表 2-2 Vim 输入模式命令

输　入	说　明
a	在光标后开始插入
A	在行尾后开始插入
I	在光标上开始插入
I	在行首前开始插入
o	在光标后的新行开始插入
O	在光标前的新行开始插入

末行模式主要用来进行一些文字编辑辅助功能，比如字串搜寻、替代、保存文件等，表 2-3 介绍一些常用的命令。

表 2-3 末行模式命令

输　入	说　明
q	结束 Vim 程序，如果文件有过修改，则必须先存储文件
q!	强制结束 Vim 程序，修改后的文件不会存储
wq	存储文件并结束程序

e	添加文件，可赋值文件名称
n	加载赋值的文件

Vi 的用法非常丰富也非常复杂，所以以上仅介绍一些初级常用命令，其他未介绍到的命令，您可以在末行模式下键入 h 或直接按【F1】键查询在线说明文件。

2.2 vi 命令格式介绍

1．进入 vi（在系统提示符下面输入以下指令）：

vi 进入 vi 而不读入任何文件

vi filename 进入 vi 并读入指定名称的文件（新、旧文件均可）。

vi +n filename 进入 vi 并且由文件的第几行开始。

vi +filename 进入 vi 并且由文件的最后一行开始。

vi + /word filename 进入 vi 并且由文件的 word 这个字开始。

vi filename(s) 进入 vi 并且将各指定文件列入名单内，第一个文件先读入。

vedit 进入 vi 并且在输入方式时会在状态行显示“INSERT MODE"。

2。编辑数个文件（利用 vi filename(s)）进入 vi 后）

:args 显示编辑名单中的各个文件名

:n 读入编辑名单中的下一个文件

:rew 读入编辑名单中的第一个文件

:e# 读入编辑名单内的前一个文件

:e file 读入另一个文件进 vi(此文件可不在编辑名单内),若原文件经修改还没有存档，则应先以: w 存档。

:e! file 强迫读入另一个文件进入 vi，原文件不作存档动作。存储及退出 vi

:w filename 存入指定文件，但未退出 vi（若未指定文件名则为当前工作的文件名）。

:wq 或者 :x 或者 zz 存文件，并且退出 vi.

:q 不作任何修改并退出 vi。

:q! 放弃任何修改并退出 vi。

:!command 暂时退出 vi 并执行 shell 指令，执行完毕后再回到 vi。

:sh 暂时退出 vi 到系统下，结束时按 Ctrl + d 则回到 vi。

3.加数据指令

i 在关标位置开始插入字符，结束时候按 ESC 键。
I 在光标所在行的最前面开始加字，结束时按 ESC 键。
a 在光标位置后开始加字，结束时按 ESC 键。
A 在光标所在行的最后面开始加字，结束时按 ESC 键。
o 在光标下加一空白行并开始加字，结束时按 ESC 键。
O 在光标上加一空白行并开始加字，结束时按 ESC 键。
!command 执行 shell 指令，并把结果加在光标所在行的下一行。

4。删除指令

nx 删除由光标位置起始的 n 个字符（含光标位置，按一个 x 表示删除光标所在的字符）
nX 删除由光标位置起始的 n 个字符（不含光标位置）。
ndw 删除光标位置其实的 n 个字符组（word）。
d0 将行的开始到光标位置的字符全部删除。
d$ 或 D 将光标位置起始到行尾的字符全部删除。
ndd 将光标位置起始的 n 行（整行）删除（dd 表示删除光标所在行）。
:start,endd 删除文件的第 start 到 end 行。

5。光标移动

0 移到一行的开始
$ 移到一行的最后
[移到文件开始位置
] 移到文件结束位置
nh 往左移 n 位
nl 或者 spacebar 往右移 n 位
nk 向上移 n 行
n+ 向上移 n 行，光标在该行的起始
ni 向下移 n 行
n- 向下移 n 行，光标在该行的起始
H 移到屏幕的左上角
M 移到屏幕的中间行开头
L 移到屏幕的最后一行
G 移到文件的最后一行

nG 或者:n　　移到文件的第 n 行
nw　　右移 n 个字组，标点符号属于字组
nW　　右移 n 个字组，标点符号不属于字组
nb　　左移 n 个字组，标点符号属于字组
nB　　左移 n 个字组，标点符号不属于字组
Ctrl + u　　屏幕上卷半个菜单
Ctrl + d　　屏幕下卷半个菜单
Ctrl + b　　屏幕上卷一个菜单
Ctrl + F　　屏幕下卷一个菜单

6。修改指令

r　　修改光标文件的字符
R　　从光标位置开始修改，结束时按 ESC 键
new　　更改 n 组字符，结束时按 ESC 键
ncc　　从光标所在位置开始更改 n 行，结束时按 ESC 键

7。重排各行长度

i　　并按 Enter 将该行由光标所在处断开，并进入 insert 方式
J　　把下一行的数据连接到本行之后

寻找指令

/text　　从光标位置往下找字串 text
?text　　从光标位置往上找字串 text
n　　继续找下一个字串（在输入上面的寻找指令之后使用）

8。寻找并且取代指令

:getxt1/s/ /text2/options　　将各行的 text1 替换为 text2
option=g 表示文件中所有的 text1 均被取代，若未输入任何 option,则只有各行中的第一个出现的 text1 被取代
option=go 在屏幕显示各取代的行
option=gc 在每个字串取代之前要求确认
Start,endgtext1/s/ / text2/options　　同上，只寻找并取代第 start~end 行。或:Start,ends/text1/text2/options

9。复制及移动文件

:first,last co dest　　将 first 到 last 行的数据复制到目标行(dest)下面
:Start,end m dest　　将 start 到 end 行的数据移动到目标行（dest）下。

:r filename　　将指定文件的内容读入光标所在行下。
nY　　将光标所在位置开始的 n 行数据暂存
p　　复制暂存数据在光标的下一行
P　　复制暂存数据在光标的上一行

10.其他命令

.	重复前一指令
u	取消前一指令
Ctrl+l	刷新屏幕显示
:set number	显示文件的行号，但不会存文件
:set nonumber	解除行号显示
:set ai	设置每行起始位置（以光标当前位置为起始）
:set noai	取消行起始位置设定
:f 或<Ctrl>+g	告诉用户有关现行编辑文件的数据

2.3 实战演习

步骤 1。在 Shell 提示符下，输入 Vi firstvi 并按<enter>键。

步骤 2。输入 A，输入 This is the first line of a Vi test file.并按<enter>

步骤 3。输入 This is the line of a Vi test file.并按<enter>

步骤 4。输入 is the 3rd line of a Vi test file.并按<enter>

步骤 5。按<Esc>键。

步骤 6。输入：W 并按<enter>键。屏幕内容应该如图下所示

This is the first line of a Vi test file. This is the line of a Vi test file. is the 3r　line of a Vi test file.

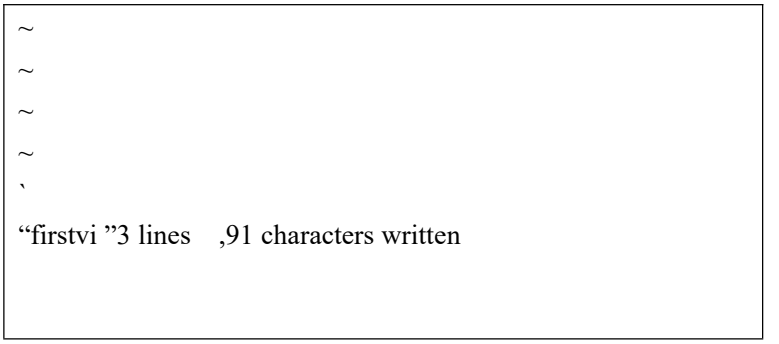


图 2-1 编辑区域

步骤 7：用方向键将光标置于文件第二行的字“line”上的第一个字符“l”上。

步骤 8：输入 I.接着输入 2nd_.(“ ”表示空格)

步骤 9：按<Esc>键.

步骤 10：用方向键将光标置于文件第三行的任意文字上。

步骤 11：输入 I，接着输入 this.

步骤 12：按<Esc>键.

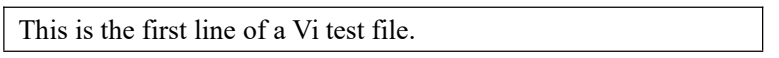
步骤 10：用方向键将光标置于文件第三行的“3r”的“r”上。

步骤 11：输入 A，接着输入 d.

步骤 12：按<Esc>键.

步骤 13：输入 A，接着输入_file.

步骤 14：按<Esc>键.现在的屏幕应该如图所示。



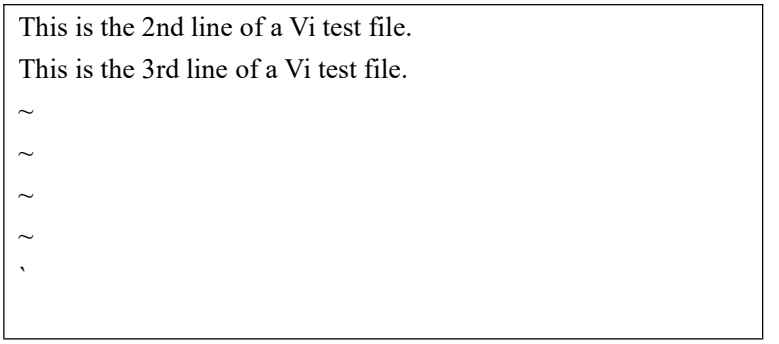


图 2-2 编辑区域

步骤 18：输入:wq，返回到 Shell 提示符下.

2.4 练习

1．利用 VI 编辑文件，并且保存，修改文件内容？

2．在 VI 命令模式下，试验本次内容的各种命令？

试验三：Lex 工具使用

3.1Lex 工具

3.1.1 词法分析程序的自动生成

从前面介绍可知，词法分析程序实际上是一张状态转换矩阵（或状态转换图）和一个控制程序。控制程序很简单，关键是构造状态转换矩阵及其相应的语义动作。构造状态转换矩阵是根据每类单词的正规式，构造相应的DFA，然后综合成一张识别所有单词的 DFA，其中影射函数就是状态转换矩阵。如果在转换矩阵中配上适应的语义动作便完成了词法分析程序的任务。前面介绍的是手工构造过程，下面讨论如何根据单词的正规式及其相应的语义动作自动产生词法分析程序。

3.1.2 正规式的表达

正则表达式在 Unix/Liunx 系统中起着非常重要的作用，在很大一部分的程序中都使用了正则表达式，可以这么说：“在 Unix/Linux 系统中，如果不懂正则表达式就不算会使用该系统”。本文中使用的 Lex 和 Yacc 都是基于正则表达式的应用，因此有必要用一篇文档的形式详细说明在 Lex 和 Yacc 中使用的正则表达式为何物！

其实正则表达式非常简单，用过 DOS 的人都知道通配符吧，说得简单一点，正则表达式就是稍微复杂一点的通配符。这里的正则表达式非常简单，规则非常少，只需要花上几分钟就可以记住。正则表达式的元字符列表如下：

表 3.1　　正规式的表达

元字符	匹配内容
.	除了换行符之外的任意字符
\n	换行符
*	0 次或者多次匹配
+	1 次或者多次匹配
?	0 次或者 1 次匹配
^	行首
$	行尾
a|b	a 或者 b

(ab)+	ab 的一次或者多次匹配
“a+b”	a+b（字面意思）
[]	一类字符

有了上面的元字符之后，就可以用上面的元字符表达出非常复杂的匹配内容出来，就像 DOS 名令中的通配符可以匹配多个指定规则的文件名一样。现在让我们看看上面的元字符的一些应用例子，列表如下：

表 3.2 正规式的表达

表达式	匹配内容
abc	abc
abc*	abc abcc abccc abcccc ……
abc+	abcc abccc abcccc ……
a(bc)+	abcbc abcbcbc abcbcbcbc ……
a(bc)?	abc abcbc
[abc]	a b c 其中之一
[a-z]	a b c d e f g… … z 其中之一
[a\-z]	a – z 三个字符其中之一
[-az]	– a z 三个字符其中之一
[A-Za-z0-9]+	大小写字符和 10 个数字的一个或多个
[\t\n]	空格，跳格，换行三者之一（空白符）
[^ab]	除了 ab 之外的任意字符
[a^b]	a ^ b 三者之一
[a|b]	a | b 三者之一
a|b	a b 两者之一
^abc$	只有 abc 的一行

注意*和+的区别，通配符只是匹配之前最近的元素，可以用小括号将多个元素括起来，整个括号括起来的整体可以看作是一个元素。那么通配符就可以匹配整个括号的内容了。

方括号表示的是一类字符，[abc]就是定义了只有 abc 三个字符的一类字符。这一点和 abc 不同，如果跟上通配符（*+?）的话，那么方括号就可以表示前面的任意的字符之一的一个字符的多个匹配，但是 abc 的话就只能是 c 的多个匹配了。说的更明白点就是 DOS 里面的通配符*表示的是任意字符的零个或者多个，而这里的方括号就是把 DOS 里面的任意字符类缩小为只有方括号表示的类了。另外还要注意连字符-在方括号中的意思，在方括号的中间表示“范围”的意思，而在首部则仅仅表示自己而已。

转义用\，这和 C 语言类似，另外还需要注意三个特殊的元字符(^ | $)

的意义。‘^’放在方括号的首部表示“除了”的意思，在其他地方没有特别意义。‘|’不在方括号中表示“或者”，‘$’通常表示行尾。

通过上面的注释可以看出：使用正则表达式可以表示非常复杂的匹配内容.

正规表达式举例：

1.正则表达式(regular expression)

例 1

name Tangl_99

这就是定义了 name 这个正则表达式,它就等于字符串 Tangl_99.所以,如果你的源程序中出现了 Tangl_99 这个字符传,那么它就等于出现一次 name 正则表达式.

例 2

digit 0|1|2|3|4|5|6|7|8|9

这个表达式就是说,正则表达式 digit 就是 0,1,2,…,9 中的某一个字母.所以无论是 0,2,或者是 9…都是属于 digit 这个正则表达式的.

“|”符号表示”或者”的意思.

那么定义正则表达式 name Tangl_99|Running,同样的,如果你的源程序中出现了 Tangl_99 或者 Running,那么就等于出现了一次 name 正则表达式.

例 3

one 1*

“*”符号表示”零到无限次重复”

那么 one 所表示的字符串就可以是空串(什么字符都没有), 1, 11, 111, 11111, 11111111111, 11111111…等等.总之,one 就是由 0 个或者 N 个 1 所组成(N 可以为任意自然数).

与”*”相同的有个”+”符号.请看下面的例子 1.4

例 4

realone 1+

“+”符号表示”1 到无限次重复”

那么 realone 和 one 不同的唯一一点就是,realone 不包含空串,因为”+”表示至少一次重复,那么 realone 至少有一个 1.所以 realone 所表达的字符串就是 1,11,111, 1111, 11111…,等等.

例 5

digit [0-9]

letter [a-zA-Z]

这里的 digit 等于例 1.2 中的 digit,也就是说,a|b|c 就相当于[a-c].

同理,letter 也就是相当于 a|b|c|d|e|f|…|y|z|A|B|C|D…|Z 不过注意的一点就是,你不能把 letter 写成[A-z],而必须大写和小写都应该各自写出来.

例 6

notA [^A]

“^”表示非,也就是除了这个字符以外的所有字符

所以 notA 表示的就是除了 A 以外的所有字符.

下面让我们来看看一些一般高级程序语言中常用的综合例子.

例 7

digit [0-9]

number {digit}+

letter [a-zA-Z_]

digit 前面多次提起过,就是 0-9 的阿拉伯数字.number 就是所有的数字组合,也就是整数.

Letter 前面也提起过,唯一不同的就是多了一个下划线.因为一般我们的 C 语言中容许有下划线来表示定义的变量名,所以我也把下划线当成英语字母来处理了.

这里 number 中使用上面定义的 digit 正则表达式.在 lex 中,用{digit}就是表示正则表达式 digit.

newline [n]

whitespace [t]+

newline 就是提行的意思.这里我们使用的是 n 这个符号,它和 C 语言中表示提行号一致.问题是大家可能要问到为什么要使用[]符号.因为在 lex 中,如果你使用[],那么里面表示的肯定就是单个字符号,而不会被理解成””和”n”两个字符.

Whitespace 就是空格符号的意思.一般的高级程序语言中有两种,一种就是简单的空格,还有一种就是 t 制表符.使用了”+”符号,就表示了这些空白符号的无限组合.

3.1.3LEX 的使用

1．Lex 工作原理

Lex 工具是一种词法分析程序生成器，它可以根据词法规则说明书的要求来生成单词识别程序，由该程序识别出输入文本中的各个单词

一组单词的正规式及其相应的语义动作叫做 LEX 语言。它是用来描述词法分析程序的，因此通过 LEX 编译程序便可以生产词法分析程序。有了词法分析程序就可以对某高级语言的输入源程序字符串进行词法分析,产生单词的内部形式（类号，内码），这个过程可用图来表示。

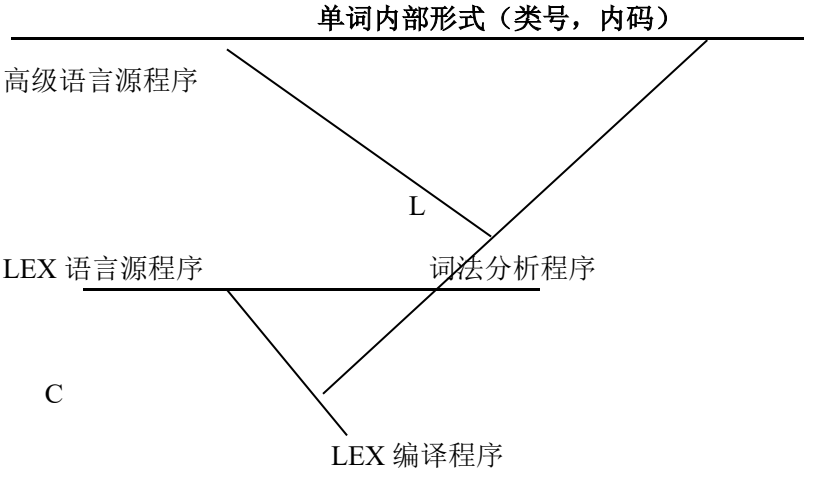


图 3-1 LEX 编译程序图

D1→R1

D2→R2

……

Dn→Rn

其中，每个 Ri 是一个正规式，Di 是代表这个正规式的简名。我们限定，在 Ri 中只许出现字母表Σ中的字符和前面已定义的简名 D1，D2，…，Di-1，不得出现未定义的简名。因为只有这样才能保证辅助定义是正规文法

所对应的正规式，否则它不能保证是正规文法所对应的语言。比如

Di→aDib，Di→ab

这便是嵌入式文法，因此它对应的语言不再是正规式了。

LEX 源程序是用一种面向问题的语言写成的，它的核心是正则表达式，利用正则表达式可以描述符号串的词法结构。用户还可以描述当某一个单词被识别出时要完成的动作。LEX 源程序的一般格式是:

{辅助定义部分}

{识别规则部分}

{用户子程序部分}

LEX 源程序的第一部分是辅助定义。在 LEX 源程序中，用户为方便起见，需要一些辅助定义，如用一个名字代表的正则表达式，或者包含一些所需的头文件，定义一些变量等。

规则部分是 LEX 源程序的核心。它是一张表，左边一列是正则表达式，右边是相应的动作，即用 C++书写的处理代码。当词法程序运行时，它把输入与规则部分的模式进行匹配。每次发现一个匹配时就执行与那种模式相关的 C++代码。

用户子程序部分的内容被 LEX 工具逐字拷贝到要生成的 C++文件中。这一部分通常包括从规则中调用的例程。

2．LEX 编译程序的构造

LEX 编译程序旨在把一个 LEX 源程序改造成一个词法分析程序 L，这个词法分析程序 L 将像一个有限自动机那样进行工作。LEX 程序的编译程序生成过程是很简单的。首先，对每个正规式 Pi 构造一个相应的不确定有限自动机 M 一，然后引进一个初态 X，通过ε弧（如图 3-2 所示）

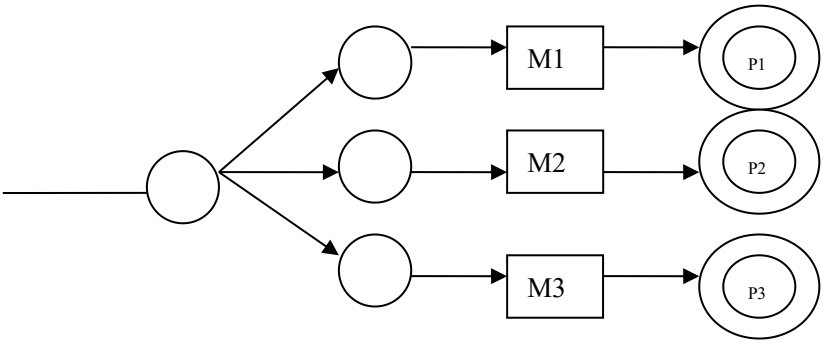


图 3-2　有限自动机

把这些自动机连成一个新的 NFA；最后，用子集法把它改造成一个等价的 DFA 。

根据 LEX 程序的要求，在 LEX 编译程序中还应注意以下几点：

首先，在原来的每个 NFA Mi 中都有它自己的终态，它表明一个匹配于 Pi 词型的输入子串 Pi 已被识别到。一但把它们合并成一个 DFA 后，在一个状态子集中可能包含若干个不同的终态，而且这个 DFA 终态和通常意义的终态也有不同。因为，我们要求的是匹配最长的子串，因此，在到达某个终态之后，这个 DFA 应继续工作下去，以便寻找与更长的子串相匹配，直至无法继续前进为止。

当达到“无法继续前进”时，就回头检查 DFA 所经历的每个状态子集，从后面逐个向前检查，直到某一个子集含有原来 NFA 的终态为止。如果不存在这种子集则认为输入串含有错误;如果这个子集中含有若干个原来 NFA 的终态，则按优先规则应在排在前面的词型 Pi 相匹配。

3.lex 程序的结构

-定义部分

-规则部分

-用户子程序部分

其中规则部分是必须的，定义和用户子程序部分是任选的。

(1) 定义部分

定义部分起始于"%{"符号，终止于"%}"符号，其间可以是包括 include 语句、声明语句在内的 C 语句。

```
%{
 #include "stdio.h"
 #include "y.tab.h"
 extern int lineno;
      %}
```

(2) 规则部分

规则部分起始于"%%"符号，终止于"%%"符号，其间则是词法规则。词法规则由模式和动作两部分组成。模式部分可以由任意的正则表达式组成，动作部分是由 C 语言语句组成，这些语句用来对所匹配的模式进行相应处理。需要注意的是，lex 将识别出来的单词存放在 yytext[]字符数据中，因此该数组的内容就代表了所识别出来的单词的内容。

```
%%
[\t] {;}
[0-9]+\.?|[0-9]*\.[0-9]+
{ sscanf(yytext,"%1f", &yylval.val);
return NUMBER; }
\n { lineno++;return '\n'; }
. { return yytex+[0]; }
%%
```

(3) 用户子程序部分

用户子程序部分可以包含用 C 语言编写的子程序，而这些子程序可以用在前面的动作中，这样就可以达到简化编程的目的。下面是带有用户子程序的 lex 程序片段。

```
        "/*" skipcmnts();
    . /* rest of rules */
    %%
skipcmnts()
{
for ( ; ; )
 {
while (input()!='*');
 if(input()!='/')
 unput(yytext[yylen-1]);
 else return;
        }
```

4．LEX 的使用方式

LEX 可以用两种方式来使用：一种是将 LEX 作为一个单独的工具，用以生成所需的识别程序，而这些识别程序通常都出现在一些非开发编译器的应用领域中，诸如编辑器设计、命令行解释、模式识别、信息检索以及开关系统等等；另一种是将 LEX 和语法分析器自动生成工具结合起来使用，以生成一个编译程序的扫描器和语法分析器.

LEX 源程序须由 LEX 系统由 LEX 系统的翻译程序进行处理。作为处理的结果，将输出一个名为 lex.yy.c 的 c 语言程序文件。此文件含有两部分内容：一是根据正规式所构造的状态转移表；二是用来驱动该状态转移表得总

控程序 yylex()。同时，LEX 源程序中所列的语义动作 Ai 也被直接拷贝到文件 lex.yy.c 之中。最后，再用 C 编译程序对 lex.yy.c 进行编译，并将支持扫描器运行的有关库函数（它们都包含在文件 L.lib 之中）连入目标码程序，便得到了所要生成的词法分析器。而后，在编译程序运行时，每从主程序调用 yylex()一次，就从被编译的输入字符流中识别一个单词。显然，对不同的 LEX 源程序而言，LEX 系统所生成的扫描器的驱动程序 yylex()都是相同的，仅状态转移表相异。但是。如果所要开发的编译程序对该驱动程序有特殊要求（例如要求回送单词之值或其它相关语义信息），也可由开发者自行编写驱动程序，而不必使用 LEX 系统所提供的 yylex 函数。

例如：已知有一个 LEX 源文件，文件名为:mylex,即可在 UNIX 系统下用如下的命令调用 LEX 对其进行处理：

LEX mylex.l

所得到的输出是名字为 lex.yy.c 的文件。在用下面的命令调用 C 编译器对其进行编译：

cc lex.yy.c –ll

其中选择项-ll,表示需调用 LEX 的库，此时所得到的文件 a.out 便是可执行的词法分析程序。如果用户对所得的目标代码文件命名，例如将其命令为 b,则可用下面的命令进行编译：

cc lex.yy.c –ll -o b

5.LEX 预定包括

1）二义性的解决

Lex 输出总是首先将可能的最长子串与规则相匹配。如果某个子串可与两个或更多

的规则匹配。

2）C 代码的插入

任何写在定义部分”%{”和”%}”之间的文本将直接复制到外置于任何过程的输出程序之中。

辅助过程中的任何文本都将被直接复制到 LEX 代码末尾的输出程序中。

将任何跟在行为部分的正规表达式之后（中间至少有一个空格）的代码插入到识别过程 yylex 的恰当位置，并在与对应的正则表达式匹配时执行它. 代表一个行为的 C 代码可以即是一个 C 语言，也可以是一个由任何说明及由位于花括号中的语言组成的复杂的 C 语言。

3）内部名字

表 3.4 Lex 变量和名字

LEX 的内部名字	含义/使用

Lex.yy.c 或 lexyy.c	Lex 输出文件名
yylex	Lex 扫描例程
input	Lex 缓冲的输入例程
ECHO	Lex 缺省行为
yyin	FILE* 类型。它指向 lexer 正在解析的当前文件。
yyout	FILE* 类型。它指向记录 lexer 输出的位置。 缺省情况下，yyin 和 yyout 都指向标准输入和输出。
yytext	匹配模式的文本存储在这一变量中（char*）。
yyleng	给出匹配模式的长度。
yylineno	提供当前的行数信息。（lexer 不一定支持。）

表 3.5　　Lex 函数

yylex()	这一函数开始分析。 它由 Lex 自动生成。
yywrap()	这一函数在文件（或输入）的末尾调用。如果函数的返回值是 1，就停止解析。 因此它可以用来解析多个文件。代码可以写在第三段，这就能够解析多个文件。 方法是使用 yyin 文件指针（见上表）指向不同的文件，直到所有的文件都被解析。最后，yywrap() 可以返回 1 来表示解析的结束。
yyless(int n)	这一函数可以用来送回除了前 n 个字符外的所有读出标记。
yymore()	这一函数告诉 Lexer 将下一个标记附加到当前标记后。

3.2lex 工具的使用方法

首先编写一个 lex 程序

vi lex.l

```
%{
#include "stdio.h"
%}
%%
[\n] ;
[0-9]+      printf("Interger: %s \n",yytext);
[0-9]*\.[0-9]+      printf("Float: %s\n",yytext);
[a-zA-Z][a-zA-Z0-9]*       printf("Word:%s\n",yytext);
.     printf("Other symbol:%c\n",yytext[0]);
%%
```

然后使用 lex 将 lex.l 转换成 C 语言程序

$lex lex.l （Linux Shell 命令）

使用上述命令产生的 C 语言程序为 lex.yy.c

然后使用 C 编译程序将 lex.yy.c 编译成可执行程序 regn

$cc -c lex.yy.c

$cc lex.yy.o -ll -o regn

下面可以使用 regn 来识别单词

```
$vi testfile
x=355
y=113
p=x/y

# ./regn < testfile
```

检验结果如下：

```
Word:x
Other symbol:=
Interger: 355
Word:y
Other symbol:=
Interger: 113
Word:p
Other symbol:=
Word:x
Other symbol:/
Word:y
#
```

3.4.LEX 实战演习

例 1．把输入串的小写字母转换成相应的大写字母，可有如下的 LEX 源程序。

```
%{
#include <stdio.h>
#ifndef     FALSE
#define     FALSE 0
#endif
#ifndef     TRUE
#define     TRUE    1
#endif
%}
   %%
[a-z] printf("%c"，   yytext[0]+'A'-'a')；
“/*” {char c;
  Int done=FALSE;
  ECHO;
do
{while((c=input())!=’*’)
putchar(c);
putchar(c);
while((c=input())!=’*’)
putchar(c);
putchar(c);
if(c= =’/’)done=TRUE;
}while(!done)
%%
void    main(void)
{yylex();}
```

其中%%是分界符，表示识别规则的开始，[a-z]是识别小写字母的规则，printf()是识别出小写字母时采取的动作，即将小写字母变换成相应的大写字母。yytext[0]是工作单元，是用以存放 yylex 识别的字符或字符串自身的值。

例 2．将十进制数转换成十六进制

```
%{
#include “stdlib.h”
```

```
 #include “stdio.h”
 int count=0;
%}
digit     [0-9]
number {digit}+
%%
{number} {int n=atoi(yytext);
   printf(“0x”,n);
   if(n>9) count++;}
   %%
main()
{yytext();
fprintf(stderr, “number of   replacements=%d”,count);
return   0;
}
```

例 3．输出指定一个给文本添加行号的扫描程序，它将其输出发送到屏幕上。

```
%{
  #include “stdio.h”
int lineno=1;
%}
Line .*\n
%%
{line}    {printf(“%5d %s”,lineno++,yytext);}
%%
main()
{yytext();return0;
}
```

例 4．C 语言十进制无符号数的识别为例，给出 LEX 源程序如下

```
%{
   #define   FCON   1
   #define   ICON    2
  %}
   D [0-9]
   %%
   {D}+                          {return ICON;}
```

```
({D}+|{D}*”.”{D}+|{D}+”.”{D}*)([eE][+|-]?{D}+)?   {return FCON;}
%%
```

其中，问号”?”是 **LEX** 系统正规式的一个运算符，其作用在于指示其前的子正规式可有可无，例如**[+|-]**？的含义是:这里可以有一个正号或一个负号，或没有符号。

5.

```
  %{
   #include <stdio.h>
   #ifndef   FALSE
   #define   FALSE 0
   #endif
   #ifndef   TRUE
   #define   TRUE 1
   #endif
  %}
  %%
[A-Z]   {putchar(tolower(yytext[0]));}
        “/*” { char c;
              Int   done=FALSE;
              ECHO;
             do
           { while((c=input())!=’*’)
              putchar(c);
              putchar(c);
        while((c=input())!=’*’)
              putchar(c);
              putchar(c);
              if(c==’/’) done=TRUE;
        }while(!done)
        }
        %%
        Void main(void)
        {yylex();}
```

6.

```
        %{
```

```
/*   Selects only lines that end or   begin with the letter   'a'.
       Deletes everything else.
*/
#include <stdio.h>
%}
ends_with_a     .*a\n
begins_with_a   a.*\n
%%
{ends_with_a}              ECHO;
{begins_with_a}         ECHO;
.*\n   ;
%%
main( )
{ yylex( );   return 0; }
```

练习：

1. 指出下列 LEX 正规式所匹配的字符串：
   1) “{”[^{}]*”}”
   2) ^[^a-z][A-Z][0-9]$
   3) [^0-9][\r\n]
   4) \’([^\n]|\’\’)+\’
   5) \”([^”\n]|\\[“\n])*\”
2. 给出识别 C 语言全部实型的自动机？
3. 写出一个 LEX 正规式，它能匹配 C 语言的所用无符号整数。
4. 写出一个 LEX 正规式，它能匹配 C 语言的标识符。
5. 写出一个 LEX 正规式，它将一个正文文件中的全部小写字符均换成大写字符，并将其中的制表字符，空白字符序列均列单个空格字符进行替换。（提示:再语义动作中使用全称变量 YYTEXT）
6. 写出识别 C 语言中所有单词的 LEX 程序？
7. 试分析分隔符、空格、跳格及回车对词法分析的影响？
8. 编写一个 LEX 程序，它能统计一个 C 语言程序中所含的用户定义标识符的个数，并能找出最长标识符中的字符个数。

实验四：YACC/Bison 工具

YACC (Yet Another Compiler _Compiler)工具是一种语法分析程序生成器，它是 LALR(1)分析器的自动生成器。它可以将有关某种语言的语法说明书转换成相应的语法分析程序，由该程序完成对相应语言中语句的语法分析工作。

YACC 和 LEX 一样是贝尔实验室在 UNIX 上首先实现的，而且与 LEX 有直接的接口，它是 UNIX 的标准应用程序。目前，YACC/Bison 与 LEX 和 FLEX 一样，可以在 UNIX，Linux,Ms-dos 运行。

4.1 YACC 程序结构

1．YACC 程序结构

在使用 yacc 工具前，必须首先编写 yacc 程序，因为有关语法分析程序是根据 yacc 程序生成的。yacc 程序实际上是有关语法规则的说明书，它也是由定义部分、规则部分和子程序部分组成的。yacc 程序的定义部分类似于 lex 程序的定义部分，只是在其后可带有 yacc 声明，其中包括词法单词、语法变量、优先级和结合性信息。yacc 程序的规则部分由语法规则和相应的动作组成，子程序部分可以包括在前面规则部分用到的子程序定义。接下来是 main 主程序，它调用 yyparse 子程序来对输入进行语法分析，而 yyparse 反复地调用 yylex 子程序来获得输入单词，在语法出错时可通过 yyerror 子程序来处理。

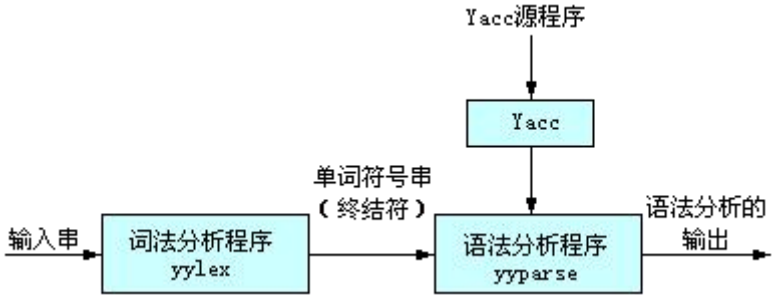


图 3.1 YACC 工作示意图

2．语法分析器的生成器 YACC

一个翻译器可用 yacc 的方式构造出来。其构造的步骤如下

首先，准备一个包含翻译器的 yacc 说明的文件，如 trtranslate.y.UNIX

系统的命令为：

```
Yacc translate.y
```

把文件 yacc translate.y 翻译器为 C 程序文件，叫做 y.tab.c，使用的方法是 LALR 方法，程序 y.tab.c 包含用 C 语言编写的 LALR 分析器和其它用户准备的 C 语言的例程。为了使 LALR 分析表少占用空间，使用了紧凑技术压缩表的大小。用命令：

```
cc  y.tab.c  -Iy
```

编译 y.tab.c，其中的 Iy 表示使用 LR 分析程序的库（名字 Iy 随系统而定）。编译得到目标程序 a.out，它完成最初的 Yacc 程序指定的翻译。如果需要其它过程，它们可以和 y.tab.c 一起编译或装入，就和使用 C 的程序一样。

4.2 YACC 源程序的组成

YACC 源程序的由 3 部分组成：

声明部分

%%

翻译规则部分

%%

用 C 语言编写的支持例程部分

1．声明部分

YACC 程序的声明部分有任选的两节。第一节处于分界符%{和}%之间。

它是一些普通的 C 语言的声明。在这里放置由第二部分和第三部分的翻译规则或过程使用的声明。 例如下面程序。下面的词法分析器是非常粗糙的，它用 C 的函数 getchar()每次第一个是数字字符，则取它的值存入变量 yylval 中，返回记号 DIGIT，否则把字符本身作为记号类返回.如果输入中有非法字符的话，这可能会引起分析器宣布一个错误而停机。

```
%{
#include 〈ctype.h〉
%}
%token DIGIT
%%
Line    : expr '\n'                  {printf("%d\n",$1);}
        ;
expr    : expr '+' term              {$$=$1+$3}
        |term
        ;
term    : term'*' factor                {$$=$1*$3}
        |factor
        ;
factor   : '('expr')'               {$$=$2}
        |DIGIT
        ;
%%
yylex()
{int n;
C=getchar();
if(isdigit(c))
      yylval=c-'0';
      return DIGIT;}
      return c;}
      }
```

2．翻译规则部分

这部分位于第一个%%后面，放置翻译规则，每条规则由一个文法产生和有关的语义动作组成。产生式集合：

〈leftside〉→〈alt1〉|〈alt2〉|…|〈altn〉

在 Yacc 中写成

〈leftside〉：〈alt1〉　{语义动作 1}

　　|〈alt2〉　{语义动作 2}

　　|…

　　|〈altn〉{语义动作 n}

　　;

在 Yacc 产生式中，单引号括起来的字符“c”是由终结符号 C 组成的记号；没有引号的字母数字串若也没有声明为记号。则是非终结符。右部的各个选择之间由竖线隔开，最后一个右部的后面用分号，表示该产生式集合的结束。第一个左部非终结符是开始符号。

Yacc 的语义动作是 C 语句序列。在语义动作中，符号$$表示引用左部非终结符的属性值，而$1 表示引用右部第 i 个文件符号的属性值。每当规约一个生产式时，执行与之相关联的语义动作，所以语义动作一般是从$i 的值决定$$的值。在这个 YACC 说明中，两个 E 产生式：

E→ E+T|T

及和它们有关的语义动作写成：

```
expr:expr'+'term          {$$=$1+$3;}
     |term
     ;
```

注意：第一个产生式的非终结符 term 是右部的第三个文法符号，“+”是第二个文法符号。第一个产生式的语义动作把右部的 expr 和 term 的值相加，把结果赋给左部的非终结符 expr,作为它的值第二个产生式的语义动作说明缺省，因为右部只有一个文法符号时，值的复写是缺省的语义动作，即它的语义动作是{$$=$1;}

注意，加了一个新的开始产生式：

```
line: expr '\n'                {printf("%d\n",$1);}
```

在这个 YACC 说明中，该产生式的意思是，这个台式计算器的输入是一个表达式后面跟一个换行字符，它的语义动作是打印表达式的十进制值并且换行。

3．用 C 语言编写的支持例程部分。YACC 说明的第三部分是一些用 C 语言编写的支持例程。名字 yylex()的词法分析器必须提供。其它的过程，如错误恢复例程，需要的话，也可以加上。

词法分析器 yylex()返回二元组（记号类，属性值）。返回的记号类，如 DIGIT,必须在 YACC 说明的第一部分声明。属性值必须通过 YACC 定义的变量 yylval 传给分析器。

4.4 用 Yacc 处理二义文法

现在我们修改这个 Yacc 程序，使台式计算器更加有用。首先，我们让台式计算器计算一列表达式，每行一个，也允许表达式之间有空白行。为做到这样，改第一第规则为

```
lines : lines epxr '\n' {printf ( "%g \ n" , $2 ); }
```

| lines ‘\ n’
|
;
按照 Yacc 的规定，第三行的空选择表示　。
其次，允许表达式使用多个数字组成的数，并将算符增加到包括+，　(一元和二元)，*和 / 。这一回我们用二义文法
E　E + E | E　E | E * E | E / E | (E) |　E | number
来描述表达式。最终的 Yacc 程序见图 3.26。
因为图 3.26 的 Yacc 程序的文法是二义的，LALR 算法将产生分析动作的冲突。Yacc 会报告产生的分析动作冲突的数目。可以用编译选择　V 获得产生的项目集和分析动作冲突的描述，这些信息在一个附加的文件 y.output 中，该文件还包含了 LR 分析表的可读表示，以及 Yacc 是怎样解决这些分析动作的冲突的。
当 Yacc 报告它发现分析动作冲突时，明智的做法是建立和查阅文件 y.output，以明白为什么会出现分析动作的冲突和它们是否已经得到正确解决。

```
%{
# include
# include
# define YYSTYPE double /*  将 Yacc 栈定义为 double 类型  */
%}

%token NUMBER
%left ‘+’ ‘　’
%left ‘*’ ‘/ ’
%right UMINUS
%%
lines : lines expr ‘\ n’ {printf ( “%g \ n”, $2 ) }
| lines ‘\ n’
| /*　　*/
;
expr : expr ‘+’ expr {$$ = $1 + $3; }
| expr ‘　’ expr {$$ = $1　　$3; }
| expr ‘*’ expr {$$ = $1 * $3; }
| expr ‘/ ’ expr {$$ = $1 / $3; }
| ‘(’ expr ‘)’ {$$ = $2; }
| ‘　’ expr %prec UMINUS {$$ =　　$2; }
```

```
| NUMBER
;
%%
yyles ( ) {
int c;
while ( ( c = getchar ( ) ) = = ‘ ’ );
if ( ( c = = ‘.’ ) | | (isdigit (c) ) ) {
ungetc (c, stdin);
scanf ( “% lf ”, &yylval);
return NUMBER;
}
return c;
}
```

图 3.26 更高级的台式计算器的 Yacc 说明

除非另有说明，否则 Yacc 按下面两条规则解决分析动作的冲突：

（1）对于归约　归约冲突，选择在 Yacc 程序中最先出现的那个冲突产生式。按此规则，为了正确解决排版文法（3.16）的冲突，只要把产生式（1）放在产生式（3）前面就足够了。

（2）对于移进　归约冲突，优先于移进。这条规则正确地解决了悬空 else 的移进　归约冲突。

由于这些缺省的规则并不总是编译器编写者所想要的，因而 Yacc 允许编译器编写者提供一些解决移进　归约冲突的说明。在声明部分，可以为终结符指定优先级和结合性。声明

%left ‘+’ ‘　’

表示+和　有同样的优先级并且它们是左结合的。声明

％right ‘　’

表示算符　为右结合，还可以用声明限制两元算符为不可结合的（即该算符的两个相邻出现根本不被允许），如

%nonassoc ‘<’

终结符的优先级按它们在声明部分出现的次序而定，先出现的优先级低，同一声明中的终结符有相同的优先级。这样，图 3.26 的声明

%right UMINUS

使得 UMINUS 的优先级高于前面五个终结符。

Yacc 解决移进　归约冲突时，首先参考这个冲突涉及的产生式和终结符的优先级和结合性。通常，产生式的优先级和结合性同它最右边终结符的一致，在大多数情况下，这是合理的决策。

如果 Yacc 必须在移进输入符号 a 和按产生式 A　　归约这两个动作之间进

行选择的话，那么当这个产生式的优先级高于 a，或者优先级相同但产生式左结合时，取归约动作，否则选择移进。例如，给定产生式

E　E + E | E * E

若搜索符是+，归约项目的产生式是 E　E + E，那么优先于归约，因为右部的+和搜索符有同样的优先级，而+是左结合的。如果搜索符是*。那么取移进，因为搜索符的优先级高于这个产生式中的+的优先级。

在那些最右终结符不能给产生式以适当的优先级和结合性的情况下，我们可以给产生式附加标记

%prec 终结符

来强制，使得它的优先级和结合性同该标记终结符的一样，这个终结符可以仅仅是个占位符，就像图 3.26 的 UMINUS 那样，它不由词法分析器返回，仅用来决定一个产生式的优先级和结合性。图 3.26 中，声明

％right UMINUS

给记号 UMINUS 指定高于*和/的优先级。在翻译规则部分，标记

％prec UMINUS

在产生式

expr : ‘　’ expr

的后面，它使得该产生式的一元减算符的优先级高于其它任何算符。

Yacc 不向编译器设计者报告用这种优先级和结合性能解决的移进　归约冲突。

4.5Yacc 的错误恢复

前面介绍的短语级错误恢复方法显然不适用于 Yacc 这样分析器自动生成的情况。Yacc 用的是紧急方式的错误恢复的思想。首先，它要求编译器设计者决定哪些“主要的”非终符将有错误恢复与它们相关联，这些非终结符的典型选择是用于产生表达式、语句、程序块和过程的那些非终结符。然后编译器设计者把形式为 A　error　的错误产生式加到文法上，其中 A 是主要非终结符，　是文法符号串，也可能是空串，error 是 Yacc 保留字。Yacc 将从这样的说明产生分析器，把错误产生式当作普通产生式处理。

当 Yacc 产生的分析器遇到错误时，它用特别的方式来处理其项目集含错误产生式的状态。遇到错误时。Yacc 从栈中弹出状态，直到发现栈顶状态的项目集包含形为 A　·error　的项目为止。然后分析器把虚构的终结符 error“移进”栈，好象它在输入中看见了这个终结符。

当为时，立即进行对 A 的归约并执行产生式 A　error 的语义动作（它可能是报告错误信息并设置标记禁止生成目标代码）。然后分析器抛弃若干

输入符号直到发现一个能回到正常处理的输入符号为止。

如果 非空，Yacc 在输入串上向前寻找能够归约为 的子串。如果含的都是终结符，那么它在输入上寻找这样的串，把其中的符号移进栈，这时，error 在分析器的栈顶。随后分析器把 error 归约为 A，恢复正常分析。

例如，出错产生式

stmt error ;

要求分析器看见错误时，跳过下一个分号，好象这个语句已经看见一样。

例 3.40 图 3.27 的 Yacc 程序在图 3.26 的基础上增加了错误产生式。错误产生式为

lines : error '\ n'

当输入行有语法错误时，分析器从栈中弹出状态，直至碰到一个有移进 error 动作的状态。状态 0 是唯一的这种状态，因为它的项目包含

lines ·error '\ n'

状态 0 总是在栈底。分析器把终结符 error 移进栈，废弃输入符号，直至发现换行字符。然后分析器把换行符移进栈，把 error '\ n' 归约成 lines, 输出诊断信息"重新输入上一行"。专门的 Yacc 例程 yyerrok 用于使分析器回到正常操作方式。

```
%{
# include
# include
# define YYSTYPE double /* 将 Yacc 栈定义为 double 类型 */
%}

%token NUMBER
%left '+' '  '
%left '*' '/ '
%right UMINUS
%%
lines : lines expr '\ n' {printf ( "%g \ n", $2 ) }
| lines '\ n'
| /*    */
| error  '\ n'  { printf (  "重新输入上一行："); yyerrok;}
;
expr : expr '+' expr {$$ = $1 + $3; }
| expr '  ' expr {$$ = $1    $3; }
```

```
| expr '*' expr {$$ = $1 * $3; }
| expr '/ ' expr {$$ = $1 / $3; }
| '(' expr ')' {$$ = $2; }
| '    ' expr %prec UMINUS {$$ =     $2; }
| NUMBER
;
%%
yylex ( ) {
int c;
while ( ( c = getchar ( ) ) = = ' ' );
if ( ( c = = '.' ) | | (isdigit (c) ) ) {
ungetc (c, stdin);
scanf ( "% lf ", &yylval);
return NUMBER;
}
return c;
}
```

图 3.27 有错误恢复的台式计算器

4.6YACC 工具的使用方法

实例：我们将 yacc 程序分成片段，把这些片段组合在一起就是 yacc 程序。我们要使用的语法规则是一个有关四则运算的语法规则，可用 BNF 范式描述

```
list: expr \n
list expr \n
expr :NUMBER
expr + expr
expr - expr
expr * expr
expr / expr
  (expr)
```

其含义是 list 是一个表达式序列，每个后面带有一个新行。表达式是一个数值，或是由运算符连起来的两个表达式，以及用圆括号括起来的表达式。下面是有关上述语法规则的 yacc 程序片段。

```
$vi hoc.y
  %{
```

```
                    #define YYSTYPE double
                    %}
                    %token NUMBER
                      %left '+' '-'
                    %left '*' '/'
                    %%
                     list:
| list '\n'
| list expr '\n' { printf("\t%. 8g\n",$2);}
;
expr : NUMBER {$$=$1;}
          | expr '+' expr {$$ = $1 + $3; }
          | expr '-' expr {$$ = $1 - $3; }
            | expr '*' expr {$$ = $1 * $3; }
          | expr '/' expr {$$ = $1 / $3; }
          | '('expr')' {$$ = $2; }
%%
```

上述 yacc 程序片段实际上是它的定义部分和规则部分。在 yacc 声明部分，%token NUMBER 表明了 NUMBER 是一个单词符号，%left 则表明了运算符号的左结合性，并且'*'和'/'和优先级比'+'和'-'的优先级高。在 yacc 程序的规则部分，备用规则是用'|'隔开的，规则中的动作实际上是 C 语句序列，其中$n(即$1,$2 等)是用来引用规则中的第几个成份，而$$则代表了整个规则的返回值。

3。**下面的 yacc 程序片段是 main 主程序**

```
#include <stdio.h>
#include <ctype.h>
char *progname;
int lineno=1;
main(argc,argv)
int argc;
char *argv[];
{ progname = argv[0];
yyparse();
}
```

main 主程序调用 yyparse 子程序来处理来处理输入,而 yyparse 又是通过 yylex 子程序来获得输入单词并通过 yyerror 子程序来报告出错信息。下面是有关这两个子程序的 yacc 程序片段

```
yylex()
```

```
{ int c;
while ((c=getchar()) == ' ' || c=='\t') ;
if (c==EOF)
return 0;
if (c=='.'||isdigit(c)){
ungetc(c,stdin);
scanf("%lf", &yylval);
return NUMBER;
}
if(c=='\n')
lineno++;
return c;
}
yyerror(s)
char *s;
{ warning (s,(char *)0);
}
warning(s,t)
char *s,*t;
{ fprintf(stderr,"%s:%s",progname,s);
if(t)
fprintf(stderr,"%s",t);
fprintf(stderr," near line %d\n",lineno);
}
```

这样就完成了整个 yacc 程序
接下来就使用 yacc 将 hoc.y 转换成 C 语言程序

```
$yacc hoc.y
```

使用上述命令产生的 C 语言程序为 y.tab.c,这时可以使用 C 编译程序将它编译成可执行程序 hoc.

```
$cc y.tab.c -o hoc
```

下面是使用 hoc 的例子

```
# ./hoc
4*3*2
24
(1+2)*(3+4)
21
```

```
1/2
0.5
355/133
2.6691729
-3-4
./hoc:Syntax error near line 5
```

上述结果显示中，分别表明了计算结果，最后一次计算出错的原因是由于在规则定义中未来定义单目减运算符号。

在 YACC 的源程序中，除了 BNF 描述的语法规则外，还可以包括当这些语法规则被识别出来时需要完成的语义动作，其语义动作可以是一段 C 程序(或 RATFOR 程序)。语义动作的内容可以是填写和查找符号表、做语义检查或生成语法树和代码生成等，若动作加在一条规则的末尾，则表明用此规则归约时所做的动作。动作也可插入在某规则的文法符号之间，这时需注意伪变量的位置关系。因为LR类分析表只有当归约时才能调语义处理动作，所以 YACC 对于在语法规则的文法符号之间插入的语义动作自动增加规则和非终结符，使其语义动作都在一条规则的末尾，即归约时做。对此的详细说明请参见附录 C。

此外 YACC 还可以处理某些二义性文法的规则，我们在第 7 章中曾介绍过二义性文法在 LR 分析中的应用，YACC 也给出了二义性文法终结符之间的优先关系和结合性的书写规定。对用户书写的二义性文法规则按其优先级和结合性自动生成相应的分析表，对于用优先级和结合性能解决的冲突，YACC 不报告错误。当所给的条件仍不能解决冲突时才报错。

在第 7 章中曾介绍过二义性文法在 LR 分析中的应用，当给出了二义性文法终结符之间的优先关系和结合性的规定后，可能会解决 LR 项目集中的冲突，用二义性文法的 LR 分析和同样语言非二义性文法的 LR 分析相比可提高对输入串分析的速度，例如：表达式的二义性文法的 LR 分析速度比非二义性文法的 LR 分析速度要快的多。

4.7 练习

1. 给出识别 C 语言全部实型的自动机？
2. 写出识别 C 语言中所有单词的 LEX 程序？
3. 试分析分隔符、空格、跳格及回车对词法分析的影响？
4. 考虑问题：
   ① 编译程序的实现应考虑的问题有那些？
   ② 编译程序的实现途径有那些？

附录 1

1. 分析 C 的词法分析器

```
/*词法分析程序*/

*****************************************/

#include<stdlib.h>

#include<stdio.h>

#include<string.h>


/*****************************************

  初始化函数

*****************************************/

void init()

{

      char *key[]={"

","auto","break","case","char","const","continue","default","do","double",
```

```
"else","enum","extern","float","for","goto","if","int","long","register",

"return","short","signed","sizeof","static","struct","switch","typedef",
                    "union","unsigned","void","volatile","while"};        /*C 语
言所有关键字，共 32 个*/
    char *limit[]={" ","(",")","[","]","->",".","!","++","--","&","~",

"*","/","%","+","-","<<",">>","<","<=",">",">=","==","!=","&&","||",
                    "=","+=","-=","*=","/=",",",";","{","}","#","_","'"};/*运算、限
界符*/
    FILE *fp;
    int i;
    char c;
    fp=fopen("k.txt","w");
    for(i=1;i<=32;i++)
        fprintf(fp,"%s\n",key[i]);
    fclose(fp);                    /*初始化关键字表*/
    fp=fopen("l.txt","w");
```

```
    for(i=1;i<=38;i++)
        fprintf(fp,"%s\n",limit[i]);
    c='';
    fprintf(fp,"%c\n",c);
    fclose(fp);                    /*初始化运算、限界符表*/
    fp=fopen("i.txt","w");
    fclose(fp);                    /*初始化标识符表*/
    fp=fopen("c.txt","w");
    fclose(fp);                    /*初始化常数表*/
    fp=fopen("output.txt","w");
    fclose(fp);                    /*初始化输出文件*/
}

/*****************************************
  十进制转二进制函数
*****************************************/
char * dtb(char *buf)
{
    int temp[20];
```

```
char *binary;
int value=0,i=0,j;
for(i=0;buf[i]!='\0';i++)
    value=value*10+(buf[i]-48);          /*先将字符转化为十进制数*/
if(value==0)
{
    binary=malloc(2*sizeof(char));
    binary[0]='0';
    binary[1]='\0';
    return(binary);
}
i=0;
while(value!=0)
{
    temp[i++]=value%2;
    value/=2;
}
temp[i]='\0';
binary=malloc((i+1)*sizeof(char));
```

```
    for(j=0;j<=i-1;j++)
        binary[j]=(char)(temp[i-j-1]+48);
    binary[i]='\0';
    return(binary);
}

/*****************************************
  根据不同命令查表或造表函数
*****************************************/
int find(char *buf,int type,int command)
{
    int number=0;
    FILE *fp;
    char c;
    char temp[30];
    int i=0;
    switch(type)
    {
        case 1: fp=fopen("k.txt","r");break;
```

```
        case 2: fp=fopen("i.txt","r");break;
        case 3: fp=fopen("c.txt","r");break;
        case 4: fp=fopen("l.txt","r");
}
c=fgetc(fp);
while(c!=EOF)
{
        while(c!='\n')
        {
                temp[i++]=c;
                c=fgetc(fp);
        }
        temp[i]='\0';
        i=0;
        number++;
        if(strcmp(temp,buf)==0)
        {
                fclose(fp);
                return(number);          /*若找到，返回在相应表中的序号*/
```

```
        }
        else
            c=fgetc(fp);
    }
    if(command==1)
    {
        fclose(fp);
        return(0);                          /*找不到，当只需查表，返回 0，否
则还需造表*/
    }
    switch(type)
    {
        case 1: fp=fopen("k.txt","a");break;
        case 2: fp=fopen("i.txt","a");break;
        case 3: fp=fopen("c.txt","a");break;
        case 4: fp=fopen("l.txt","a");
    }
    fprintf(fp,"%s\n",buf);
    fclose(fp);
```

```
        return(number+1);                    /*造表时，将字符串添加到表尾并返
回序号值*/
}

/*****************************************
  数字串处理函数
*****************************************/
void cs_manage(char *buffer)
{
        FILE *fp;
        char *pointer;
        int result;
        pointer=dtb(buffer);
        result=find(pointer,3,2);          /*先查常数表，若找不到则造入常数表
并返回序号值*/
        fp=fopen("output.txt","a");
        fprintf(fp,"%s\t\t\t3\t\t\t%d\n",buffer,result);
        fclose(fp);                        /*写入输出文件*/
}
```

```
/*****************************************
  字符串处理函数
*****************************************/
void ch_manage(char *buffer)
{
      FILE *fp;
      int result;
      result=find(buffer,1,1);                  /*先查关键字表*/
      fp=fopen("output.txt","a");
      if(result!=0)
          fprintf(fp,"%s\t\t\t1\t\t\t%d\n",buffer,result);      /*若找到，写入输出
文件*/
      else
      {
           result=find(buffer,2,2);           /*若找不到，则非关键字，查标识符
表，还找不到则造入标识符表*/
           fprintf(fp,"%s\t\t\t2\t\t\t%d\n",buffer,result);
      }                                              /*写入输出文件*/
```

```
    fclose(fp);
}

/*****************************************
  出错处理函数
*****************************************/
void er_manage(char error,int lineno)
{
    printf("\nerror: %c ,line %d",error,lineno);      /*报告出错符号和所在行
数*/
}

/*****************************************
  扫描程序
*****************************************/
void scanner()
{
    FILE *fpin,*fpout;
    char filename[20];
```

```
char ch;
int i=0,line=1;
int count,result,errorno=0;
char array[30];
char *word;
printf("\nthe file name:");
scanf("%s",filename);
if((fpin=fopen(filename,"r"))==NULL)
{
    printf("cannot open file");
    return;
}
ch=fgetc(fpin);
while(ch!=EOF)
{                       /*按字符依次扫描源程序，直至结束*/
    i=0;
    if(((ch>='A')&&(ch<='Z'))||((ch>='a')&&(ch<='z'))||(ch=='_'))
    {               /*以字母开头*/
```

```
while(((ch>='A')&&(ch<='Z'))||((ch>='a')&&(ch<='z'))||(ch=='_')||((ch>='0')&
&(ch<='9')))
                {
                    array[i++]=ch;
                    ch=fgetc(fpin);
                }
                word=(char *)malloc((i+1)*sizeof(char));
                memcpy(word,array,i);
                word[i]='\0';
                ch_manage(word);
                if(ch!=EOF)
                    fseek(fpin,-1L,SEEK_CUR);
            }
            else if(ch>='0'&&ch<='9')
            {               /*以数字开头*/
                while(ch>='0'&&ch<='9')
                {
                    array[i++]=ch;
```

```
            ch=fgetc(fpin);
        }
        word=(char *)malloc((i+1)*sizeof(char));
        memcpy(word,array,i);
        word[i]='\0';
        cs_manage(word);
        if(ch!=EOF)
            fseek(fpin,-1L,SEEK_CUR);
    }
    else if((ch==' ')||(ch=='\t'))
        ;                   /*消除空格符和水平制表符*/
    else if(ch=='\n')
        line++;                 /*消除回车并记录行数*/
    else if(ch=='/')
    {                           /*消除注释*/
        ch=fgetc(fpin);
        if(ch=='=')
        {                       /*判断是否为‘/=’符号*/
            fpout=fopen("output.txt","a");
```

```
            fprintf(fpout,"/=\t\t\t4\t\t\t32\n");
            fclose(fpout);
              }
        else if(ch!='*')
        {                   /*若为除号，写入输出文件*/
            fpout=fopen("output.txt","a");
            fprintf(fpout,"/\t\t\t4\t\t\t13\n");
            fclose(fpout);
            fseek(fpin,-1L,SEEK_CUR);
              }
        else if(ch=='*')
        {                   /*若为注释的开始 ,消除包含在里面的所有字
符*/
              count=0;
            ch=fgetc(fpin);
            while(count!=2)
            {               /*当扫描到‘*’且紧接着下一个字符为‘/’才是注
释的结束*/
              count=0;
```

```
                while(ch!='*')
                        ch=fgetc(fpin);
                count++;
                    ch=fgetc(fpin);
                if(ch=='/')
                            count++;
                            else
                        ch=fgetc(fpin);
                }
                }
 }
else if(ch=='"')
{                       /*消除包含在双引号中的字符串常量*/
      fpout=fopen("output.txt","a");
      fprintf(fpout,"%c\t\t\t4\t\t\t37\n",ch);
      ch=fgetc(fpin);
      while(ch!='"')
            ch=fgetc(fpin);
      fprintf(fpout,"%c\t\t\t4\t\t\t37\n",ch);
```

```
            fclose(fpout);
        }
        else
        {           /*首字符为其它字符,即运算限界符或非法字符*/
            array[0]=ch;
            ch=fgetc(fpin);          /*再读入下一个字符，判断是否为双字
符运算、限界符*/
            if(ch!=EOF)
            {               /*若该字符非文件结束符*/
                array[1]=ch;
                word=(char *)malloc(3*sizeof(char));
                memcpy(word,array,2);
                word[2]='\0';
                result=find(word,4,1);        /*先检索是否为双字符运
算、限界符*/
                if(result==0)
                {                                   /*若不是*/
                    word=(char *)malloc(2*sizeof(char));
                    memcpy(word,array,1);
```

```
                word[1]='\0';
                result=find(word,4,1);          /*检索是否为单字符运
算、限界符*/
                if(result==0)
                {                                   /*若还不是，则为非
法字符*/
                    er_manage(array[0],line);
                      errorno++;
                    fseek(fpin,-1L,SEEK_CUR);
                }
                else
                {       /*若为单字符运算、限界符，写入输出文件并
将扫描文件指针回退一个字符*/
                    fpout=fopen("output.txt","a");
                    fprintf(fpout,"%s\t\t\t4\t\t\t%d\t\n",word,result);
                      fclose(fpout);
                    fseek(fpin,-1L,SEEK_CUR);
                      }
            }
```

```
                else
                {                   /*若为双字符运算、限界符，写输出文件
*/
                    fpout=fopen("output.txt","a");
                    fprintf(fpout,"%s\t\t\t4\t\t\t%d\n",word,result);
                    fclose(fpout);
                  }
            }
            else
            {                   /*若读入的下一个字符为文件结束符*/
                word=(char *)malloc(2*sizeof(char));
                memcpy(word,array,1);
                word[1]='\0';
                result=find(word,4,1);          /*只考虑是否为单字符运
算、限界符*/
                if(result==0)                   /*若不是，转出错处理*/
                    er_manage(array[0],line);
                  else
                {                                   /*若是，写输出文件*/
```

```
                    fpout=fopen("output.txt","a");
                    fprintf(fpout,"%s\t\t\t4\t\t\t%d\n",word,result);
                    fclose(fpout);
                }
                }
            }
            ch=fgetc(fpin);
        }
        fclose(fpin);
        printf("\nThere are %d error(s).\n",errorno);          /*报告错误字符
个数*/
}

/*****************************************
  主函数
*****************************************/
main()
{
    init();                 /*初始化*/
```

```
    scanner();              /*扫描源程序*/

}
```

程序的输出结果

1．常量的输出 1，100

2．保留字：auto

break	case	char	const
continue	default	do	double
else	enum	extern	float
for	goto	if	int
long	register	return	short
signed	sizeof	static	struct
switch	typedef	union	unsigned
void	volatile	while	

3．标识符的输出：

main	i	j
m	n	printf

3．界符

(	)	[	]
->	.	!	++
--	&	~	*
/	%	+	-
<<	>>	<	<=
>	>=	==	!=
&&	||	=	+=
-=	*=	/=	,
;	{	}	#
_	'	"	

结果输出：

```
main        2        1
(           4        1
)           4        2
{           4        34
int         1        17
i           2        2
```

,	4	32
j	2	3
,	4	32
m	2	4
,	4	32
n	2	5
;	4	33
i	2	2
=	4	27
1	3	1
;	4	33
j	2	3
=	4	27
4	3	2
;	4	33
m	2	4
=	4	27
i	2	2
+	4	15
j	2	3
;	4	33
n	2	5
=	4	27
i	2	2
-	4	16
j	2	3
;	4	33
printf	2	6
(	4	1
"	4	37
"	4	37
,	4	32
m	2	4
,	4	32
n	2	5
)	4	2
;	4	33
}	4	35

2．下面给出一个**测试用 Pascal 代码片断**

```
begin
ab2a:=9;
 if x>=0 then x:=x+1;
while a=0 do
b:=a*x/33455;
end
#
```

分析器的 C 代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_WD_LEN 255
#define MAX_INT 32767
#define MAX_SRC_LEN 1000
#define MAX_WD_CNT 100
#define KWD_CNT 6
/***********************************************/

union value_type{
   int d;
    char c;
```

```
    char s[MAX_WD_LEN];
  };

typedef struct{
    int syn;
    union value_type value;
  }word_type;

/************************************************/
char *keywords[20]={"begin","if","then","while","do","end"};
char source[MAX_SRC_LEN];
word_type word_stack[MAX_WD_CNT];
int line=1,wtop=0,ip=0;

/************************************************/
void p_word_stack(){
 int i;word_type w;
  printf("syn\t|value\n");
 printf("________|________\n");
 for(i=0;i<wtop;i++){
 w=word_stack[i];
  if( (w.syn>=1 && w.syn<=10) || w.syn==18 || w.syn==21 ||
w.syn==22
|| w.syn==24)
                printf("%d\t|%s\n",w.syn,w.value.s);
              else if(w.syn==11)
```

```
                printf("%d\t|%d\n",w.syn,w.value.d);
else
                printf("%d\t|%c\n",w.syn,w.value.c);
        }
           return ;
}
void tell_err(){
 printf("error in line %d\n",line);
 exit(1);
  return ;
}
void scan(){
 word_type w;
 char c;
  int j=0;
    if(isdigit(c=source[ip])){
                    w.syn=11;    /* dd*   */
                     w.value.d=c-'0';
                   while(isdigit(c=source[++ip]))
                 w.value.d=w.value.d*10+c-'0';
                   if(!isalpha(c))
                  word_stack[wtop++]=w;
               else
              t ell_err();
                    return;
```

```
            }
            if(isalpha(c=source[ip])){
            w.syn=10;     /*   (ll|d) */
          w.value.s[0]=c;
             while(isalpha(c=source[++ip]) || isdigit(c))
               w.value.s[++j]=c;
            w.value.s[j+1]='\0';
               for(j=0;j<KWD_CNT;j++){
                  if(strcmp(keywords[j],w.value.s)==0)
                    w.syn=j+1;
            }
            word_stack[wtop++]=w;
                    return ;
 }
switch(c=source[ip]){
case '+' :
                    w.syn=14;   /* '+' */
                       w.value.c='+';
                             word_stack[wtop++]=w;
                  ip++;
                    break;
   case '-' :
                  w.syn=15; /* '-' */
                  w.value.c='-';
                  word_stack[wtop++]=w;
```

```
                vip++;
                 break;
  case '*' :
                w.syn=16;   /* '*' */
                 w.value.c='*';
                 word_stack[wtop++]=w;
                ip++;
                break;
case '/' :
                 w.syn=17;
                w.value.c='/';
                word_stack[wtop++]=w;
                 ip++;
                break;
    case ':' :
                w.syn=19;
                w.value.c=':';
                if( (c=source[++ip]) !='='){
               word_stack[wtop++]=w;

                }
                 else if(c=='='){
                 strcpy(w.value.s,":=");
                 w.syn=18;
                 word_stack[wtop++]=w;
```

```
            ip++;
           }
           break;
case '<' :
            w.syn=20;
           w.value.c='<';
            if( (c=source[++ip]) !='>' && c!='='){
            word_stack[wtop++]=w;
            }
           else if(c=='>'){
            w.syn=21;
           strcpy(w.value.s,"<>");
            word_stack[wtop++]=w;
            ip++;
           }
  else if(c=='='){
           w.syn=22;
           strcpy(w.value.s,"<=");
            word_stack[wtop++]=w;
            ip++;
           }
           break;
            w.syn=23;
           w.value.c='>';
            if( (c=source[++ip]) !='='){
```

```
            word_stack[wtop++]=w;
            }
            else if(c=='='){
              w.syn=24;
              strcpy(w.value.s,">=");
                word_stack[wtop++]=w;
              ip++;
            }
              break;
case '=' :
            w.syn=25;
            w.value.c='=';
              word_stack[wtop++]=w;
            ip++;
              break;
case ';' :
               w.syn=26;
               w.value.c=';';
               word_stack[wtop++]=w;
                      ip++;
                    break;
  case '(' :
                  w.syn=27;
                  w.value.c='(';
                  word_stack[wtop++]=w;
```

```
                ip++;
                  break;
        case ')' :
                  w.syn=28;
                w.value.c=')';
                word_stack[wtop++]=w;
               ip++;
                break;
        case ' ' :
                while(source[++ip]==' ');
             break;
        case '\n' :
                  line++;
                  while(source[++ip]=='\n')line++;
                  break;

        case '\t' :
                while(source[++ip]=='\t');
                break;
        case '\r' :
             while(source[++ip]=='\r');
             break;
        default:
                tell_err();
}
```

```
        return;
}
int main(){
        FILE* fp;
        int i=0;
            word_type w;
        fp=fopen("input.txt","r");
        while(!feof(fp))
                source[i++]=getc(fp);
        fclose(fp);
        while(source[ip]!='#')
                scan(ip);
        w.syn=0;
        w.value.c='#';
        word_stack[wtop++]=w;
        p_word_stack();
}
```

测试结果

```
syn     |value
________|________
1       |begin
10      |ab2a
18      |:=
11      |9
26      |;
2       |if
```

```
10      |x
24      |>=
11      |0
3       |then
10      |x
18      |:=
10      |x
14      |+
11      |1
26      |;
4       |while
10      |a
25      |=
11      |0
5       |do
10      |b
18      |:=
10      |a
16      |*
10      |x
17      |/
11      |33455
26      |;
6       |end
0                                       |#
```

附录 2

```
/*递归分析程序*/

#include <stdio.h>

#include<dos.h>

#include<stdlib.h>

#include<string.h>


char a[50] ,b[50],d[200],e[10];

char ch;

int n1,i1=0,flag=1,n=5;


int E();

int E1();

int T();

int G();

int S();

int F();
```

```
void input();

void input1();

void output();




void main()

/*递归分析*/

{

        int f,p,j=0;

        char x;

    d[0]='E';

    d[1]='=';

    d[2]='>';

    d[3]='T';

    d[4]='G';

    d[5]='#';

        printf("请输入字符串(长度<50,以#号结束 ) \n");

        do{

                scanf("%c",&ch);
```

```
                a[j]=ch;
                j++;
        }while(ch!='#');
        n1=j;
        ch=b[0]=a[0];
        printf("文法\t 分析串\t\t 分析字符\t 剩余串\n");
        f=E1();
        if (f==0) return;
        if (ch=='#')
        {    printf("accept\n");
                                p=0;
                x=d[p];
                while(x!='#') {
                        printf("%c",x);p=p+1;x=d[p];
/*输出推导式*/
                }
        }
        else {
                printf("error\n");
```

```
                printf("回车返回\n");
                getchar();getchar();
                return;
        }
        printf("\n");
        printf("回车返回\n");
        getchar();
        getchar();
}

int E1()
{   int f,t;
        printf("E-->TG\t");
        flag=1;
        input();
        input1();
        f=T();
        if (f==0) return(0);
        t=G();
```

```
        if (t==0) return(0);
                else return(1);
}

int E()
{   int f,t;
        printf("E-->TG\t");
    e[0]='E';e[1]='=';e[2]='>';e[3]='T';e[4]='G';e[5]='#';
        output();
        flag=1;
        input();
        input1();
        f=T();
        if (f==0) return(0);
        t=G();
        if (t==0) return(0);
                else return(1);
}
```

```
int T()
{     int f,t;
            printf("T-->FS\t");
            e[0]='T';e[1]='=';e[2]='>';e[3]='F';e[4]='S';e[5]='#';
      output();
            flag=1;
            input();
            input1();
            f=F();
            if (f==0) return(0);
            t=S();
            if (t==0) return(0);
                  else return(1);
}

int   G()
{     int f;
            if(ch=='+') {
                  b[i1]=ch;
```

```
                printf("G-->+TG\t");


e[0]='G';e[1]='=';e[2]='>';e[3]='+';e[4]='T';e[5]='G';e[6]='#';
                output();
            flag=0;
                input();input1();
                ch=a[++i1];
                f=T();
                if (f==0) return(0);
                G();
                return(1);
        }
        printf("G-->^\t");
        e[0]='G';e[1]='=';e[2]='>';e[3]='^';e[4]='#';
        output();
        flag=1;
        input();input1();
        return(1);
}
```

```
int S()

{

        int f,t;

        if(ch=='*') {

                b[i1]=ch;printf("S-->*FS\t");



e[0]='S';e[1]='=';e[2]='>';e[3]='*';e[4]='F';e[5]='S';e[6]='#';

                output();

                flag=0;

                input();input1();

                ch=a[++i1];

                f=F();

                if (f==0) return(0);

                t=S();

                if (t==0) return(0);

                                                        else return(1);}

        printf("S-->^\t");

        e[0]='S';e[1]='=';e[2]='>';e[3]='^';e[4]='#';
```

```
            output();
            flag=1;
            a[i1]=ch;
            input();input1();
            return(1);
}

int F()
{   int f;
            if(ch=='(') {
                        b[i1]=ch;printf("F-->(E)\t");

            e[0]='F';e[1]='=';e[2]='>';e[3]='(';e[4]='E';e[5]=')';e[6]='#';
                        output();
                        flag=0;
                        input();input1();
                        ch=a[++i1];
                        f=E();
                        if (f==0) return(0);
```

```
                if(ch==')') {
                        b[i1]=ch;printf("F-->(E)\t");
                    flag=0;input();input1();
                        ch=a[++i1];
                }
                  else {
                        printf("error\n");
                        return(0);
                  }
}
else if(ch=='i') {
                b[i1]=ch;printf("F-->i\t");
                e[0]='F';e[1]='=';e[2]='>';e[3]='i';e[4]='#';
                output();
                flag=0;input();input1();
                ch=a[++i1];
}
     else {printf("error\n");return(0);}
   return(1);
```

```
}

void input()
{
        int j=0;
        for (;j<=i1-flag;j++)
        printf("%c",b[j]);
/*输出分析串*/
        printf("\t\t");
        printf("%c\t\t",ch);
/*输出分析字符*/
}

void input1()
{
        int j;
        for (j=i1+1-flag;j<n1;j++)
        printf("%c",a[j]);                    /*输出剩余字符*/
    printf("\n");
```

```
}

void output(){                                  /*推导式计算*/
        int m,k,j,q;
    int i=0;
        m=0;k=0;q=0;
    i=n;
        d[n]='=';d[n+1]='>';d[n+2]='#';n=n+2;i=n;
        i=i-2;
        while(d[i]!='>'&&i!=0) i=i-1;
        i=i+1;
        while(d[i]!=e[0]) i=i+1;
        q=i;
        m=q;k=q;
        while(d[m]!='>')    m=m-1;
        m=m+1;
        while(m!=q) {
                d[n]=d[m];m=m+1;n=n+1;
        }
```

```
            d[n]='#';
        for(j=3;e[j]!='#';j++){
                        d[n]=e[j];
                        n=n+1;
            }
            k=k+1;
            while(d[k]!='')   {
                        d[n]=d[k];n=n+1;k=k+1;
            }
            d[n]='#';
}
```

附录 3

```
/*LL(1)分析法源程序，只能在 VC++中运行   */

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<dos.h>

char A[20];/*分析栈*/

char B[20];/*剩余串*/

char v1[20]={'i','+','*','(',')','#'};/*终结符   */

char v2[20]={'E','G','T','S','F'};/*非终结符     */


int j=0,b=0,top=0,l;/*L 为输入串长度  */


typedef struct type/*产生式类型定义          */

{

        char origin;/*大写字符   */
```

```
        char array[5];/*产生式右边字符  */
        int length;/*字符个数          */
}type;

type e,t,g,g1,s,s1,f,f1;/*结构体变量   */
type C[10][10];/*预测分析表          */

void print()/*输出分析栈   */
{
        int a;/*指针*/
        for(a=0;a<=top+1;a++)
                printf("%c",A[a]);
        printf("\t\t");
}/*print*/

void print1()/*输出剩余串*/
{
        int j;
        for(j=0;j<b;j++)/*输出对齐符*/
```

```
                printf(" ");
        for(j=b;j<=l;j++)
                printf("%c",B[j]);
        printf("\t\t\t");
}/*print1*/

void main()
{
        int m,n,k=0,flag=0,finish=0;
        char ch,x;
        type cha;/*用来接受 C[m][n]*/
        /*把文法产生式赋值结构体*/
        e.origin='E';
        strcpy(e.array,"TG");
        e.length=2;
        t.origin='T';
        strcpy(t.array,"FS");
        t.length=2;
        g.origin='G';
```

```
        strcpy(g.array,"+TG");
        g.length=3;
        g1.origin='G';
        g1.array[0]='^';
        g1.length=1;
s.origin='S';
        strcpy(s.array,"*FS");
        s.length=3;
        s1.origin='S';
        s1.array[0]='^';
        s1.length=1;
        f.origin='F';
        strcpy(f.array,"(E)");
        f.length=3;
        f1.origin='F';
        f1.array[0]='i';
        f1.length=1;

        for(m=0;m<=4;m++)/*初始化分析表*/
```

```
                    for(n=0;n<=5;n++)
                     C[m][n].origin='N';/*全部赋为空*/
    /*填充分析表*/
    C[0][0]=e;C[0][3]=e;
    C[1][1]=g;C[1][4]=g1;C[1][5]=g1;
    C[2][0]=t;C[2][3]=t;
    C[3][1]=s1;C[3][2]=s;C[3][4]=C[3][5]=s1;
    C[4][0]=f1;C[4][3]=f;
    printf("提示:本程序只能对由'i','+','*','(',')'构成的以'#'结束的字符串进行分
析,\n");
    printf("请输入要分析的字符串:");
    do/*读入分析串*/
    {
          scanf("%c",&ch);
          if ((ch!='i') &&(ch!='+')
&&(ch!='*')&&(ch!='(')&&(ch!=')')&&(ch!='#'))
            {
            printf("输入串中有非法字符\n");
             exit(1);
```

```
            }
            B[j]=ch;
            j++;
}while(ch!='#');
l=j;/*分析串长度*/
ch=B[0];/*当前分析字符*/
A[top]='#'; A[++top]='E';/*'#','E'进栈*/
printf("步骤\t\t 分析栈 \t\t 剩余字符 \t\t 所用产生式 \n");
do
{
    x=A[top--];/*x 为当前栈顶字符*/
     printf("%d",k++);
     printf("\t\t");
     for(j=0;j<=5;j++)/*判断是否为终结符*/
                        if(x==v1[j])
                        {
                           flag=1;
                        break;
                        }
```

```
 if(flag==1)/*如果是终结符*/
 {
 if(x=='#')
 {
      finish=1;              /*结束标记*/
      printf("acc!\n");/*接受  */
      getchar();
      getchar();
         exit(1);
              }                  /*if*/
if(x==ch)
              {
print();
 print1();
 printf("%c 匹配\n",ch);
 ch=B[++b];                  /*下一个输入字符*/
 flag=0;                     /*恢复标记*/
              }                  /*if*/
 else              /*出错处理*/
```

```
                {
                print();
                print1();
                printf("%c 出错\n",ch);/*输出出错终结符*/
                exit(1);
                  }                          /*else*/
                   }                             /*if*/
                   else                          /*非终结符处理*/
                   {
                       for(j=0;j<=4;j++)
                            if(x==v2[j])
                                        {
                                        m=j;        /*行号*/
                                        break;
                                         }
                          for(j=0;j<=5;j++)
                          if(ch==v1[j])
{
              n=j;                          /*列号*/
```

```
                break;
                                        }
            cha=C[m][n];
    if(cha.origin!='N')                 /*判断是否为空*/
 {
              print();
          print1();
          printf("%c->",cha.origin);        /*输出产生式*/
          for(j=0;j<cha.length;j++)
           printf("%c",cha.array[j]);
          printf("\n");
    for(j=(cha.length-1);j>=0;j--)          /*产生式逆序入栈*/
  A[++top]=cha.array[j];
if(A[top]=='^')                         /*为空则不进栈*/
  top--;
           }                    /*if*/
  Else                          /*出错处理*/
{
  print();
```

```
            print1();
            printf("%c 出错\n",x);                    /*输出出错非终结符*/
            exit(1);
            }                                          /*else*/
            }                                         /*else*/
    }while(finish==0);
}/*main*/
```

附录 4

/*LR 分析法的程序*/

```
#include<stdio.h>

#include<string.h>


char *action[10][3]={"S3#","S4#",NULL,                    /*ACTION 表*/

                     NULL,NULL,"acc",

                                                          "S6#","S7#",NULL,

                                                          "S3#","S4#",NULL,

                                                          "r3#","r3#",NULL,

                                                          NULL,NULL,"r1#",

                                                          "S6#","S7#",NULL,

                                                          NULL,NULL,"r3#",

                                                          "r2#","r2#",NULL,

                                                          NULL,NULL,"r2#"};

int goto1[10][2]={1,2,                                   /*QOTO 表*/

                  0,0,
```

```
                                        0,5,
                                        0,8,
                                        0,0,
                                        0,0,
                                        0,9,
                                        0,0,
                                        0,0,
                                        0,0};
char vt[3]={'a','b','#'};                       /*存放非终结符*/
char vn[2]={'S','B'};                           /*存放终结符*/
char *LR[4]={"E->S#","S->BB#","B->aB#","B->b#"};/*存放产生式*/

int a[10];
char b[10],c[10],c1;
int top1,top2,top3,top,m,n;

void main(){
        int g,h,i,j,k,l,p,y,z,count;
        char x,copy[10],copy1[10];
```

```
top1=0;top2=0;top3=0;top=0;
a[0]=0;y=a[0];b[0]='#';
count=0;z=0;
printf("请输入表达式\n");
do{
        scanf("%c",&c1);
        c[top3]=c1;
        top3=top3+1;
}while(c1!='#');
printf("步骤\t 状态栈\t\t 符号栈\t\t 输入串\t\tACTION\tGOTO\n");
do{
        y=z;m=0;n=0;                    /*y,z 指向状态栈栈顶*/
        g=top;j=0;k=0;
        x=c[top];
        count++;
        printf("%d\t",count);
        while(m<=top1){                 /*输出状态栈*/
        printf("%d",a[m]);
       m=m+1;
```

```
                }
                printf("\t\t");
                while(n<=top2){                    /*输出符号栈*/
                printf("%c",b[n]);
               n=n+1;
                }
                printf("\t\t");
                while(g<=top3){                    /*输出输入串*/
                printf("%c",c[g]);
               g=g+1;
                }
                printf("\t\t");
                while(x!=vt[j]&&j<=2) j++;
                if(j==2&&x!=vt[j]){
              printf("error\n");
                return;
                }
if(action[y][j]==NULL){
 printf("error\n");
```

```
return;
                }
                else
                strcpy(copy,action[y][j]);
                if(copy[0]=='S'){                              /*处理移进*/
                z=copy[1]-'0';
      top1=top1+1;
                top2=top2+1;
                a[top1]=z;
                b[top2]=x;
                top=top+1;
                i=0;
                while(copy[i]!='#'){
                printf("%c",copy[i]);
                i++;
                }
                printf("\n");
                }
                if(copy[0]=='r'){                              /*处理归约*/
```

```
i=0;
while(copy[i]!='#'){
printf("%c",copy[i]);
i++;
}
h=copy[1]-'0';
strcpy(copy1,LR[h]);
while(copy1[0]!=vn[k]) k++;
l=strlen(LR[h])-4;
top1=top1-l+1;
top2=top2-l+1;
y=a[top1-1];
p=goto1[y][k];
a[top1]=p;
b[top2]=copy1[0];
z=p;
printf("\t");
printf("%d\n",p);
}
```

```
        }while(action[y][j]!="acc");

        printf("acc\n");

}
```

附录 5

```
/*逆波兰式的产生及计算*/

#include<stdio.h>

#include<math.h>

#define max 100

char ex[max];            /*存储后缀表达式*/

void trans(){            /*将算术表达式转化为后缀表达式*/

        char str[max];     /*存储原算术表达式*/

        char stack[max];           /*作为栈使用*/

        char ch;

        int sum,i,j,t,top=0;

        printf("****************************************\n");

        printf("*输入一个求值的表达式，以#结束。*\n");

        printf("****************************************\n");

        printf("算数表达式：");

        i=0;                              /*获取用户输入的表达式*/

        do{

                i++;
```

```
                scanf("%c",&str[i]);
        }while(str[i]!='#' && i!=max);
sum=i;
        t=1;i=1;
        ch=str[i];i++;
        while(ch!='#'){
                switch(ch){
                case '(':                       /*判定为左括号*/
                                top++;stack[top]=ch;
                                        break;
        case ')':                       /*判定为右括号*/
         while(stack[top]!='('){
                ex[t]=stack[top];top--;t++;
                                }
                                top--;
                                break;
        case '+':                       /*判定为加减号*/
         case '-':
                while(top!=0&&stack[top]!='('){
```

```
                ex[t]=stack[top];top--;t++;
            }
            top++;stack[top]=ch;
            break;
case '*':                   /*判定为乘除号*/
 case '/':
                while(stack[top]=='*'||stack[top]=='/'){
                        ex[t]=stack[top];top--;t++;
                }
                top++;stack[top]=ch;
                break;
case ' ':break;
default:while(ch>='0'&&ch<='9'){        /*判定为数字*/
                ex[t]=ch;t++;
                ch=str[i];i++;
                        }
                i--;
                ex[t]='#';t++;
            }
```

```
                ch=str[i];i++;
            }
            while(top!=0){
                ex[t]=stack[top];t++;top--;
            }
            ex[t]='#';
            printf("\n\t 原来表达式：");
            for(j=1;j<sum;j++)
                printf("%c",str[j]);
           printf("\n\t 后缀表达式：",ex);
            for(j=1;j<t;j++)
                printf("%c",ex[j]);
}
void compvalue(){                                   /*计算后缀表达式的值*/
            float stack[max],d;                     /*作为栈使用*/
            char ch;
            int t=1,top=0;                  /*t 为 ex 下标，top 为 stack 下标*/
            ch=ex[t];t++;
      while(ch!='#'){
```

```
switch(ch){
   case '+':
                              stack[top-1]=stack[top-1]+stack[top];
                              top--;
                              break;
  case '-':
                              stack[top-1]=stack[top-1]-stack[top];
                              top--;
                              break;
  case '*':
                              stack[top-1]=stack[top-1]*stack[top];
                              top--;
                              break;

  case '/':
                 if(stack[top]!=0)
                                 stack[top-1]=stack[top-1]/stack[top];
                 else{
                              printf("\n\t 除零错误!\n");
```

```
                    exit(0);                    /*异常退出*/
                    }
                    top--;
                    break;
                default:
                    d=0;
                    while(ch>='0'&&ch<='9'){
                        d=10*d+ch-'0';          /*
将数字字符转化为对应的数值*/
                        ch=ex[t];t++;
                    }
                    top++;
                    stack[top]=d;
                }
                ch=ex[t];t++;
            }
            printf("\n\t 计算结果:%g\n",stack[top]);
}
```

```
main(){
        trans();
        compvalue();
}
```

附录 1：c 语言一 一个 c 语言扫描器的 LEX 源文件

五、 LEX 源文件示例——C 语言词法分析器

本节，我们给出一个针对 C 语言单词识别的 LEX 源程序文件 c.lex，供读者参考。此文件已在 UNIX 环境下编译调试通过。

程序 c.lex——一个 C 语言扫描器的 LEX 源文件

```
1. %{
2. #include <stdio.h>
3. #include <ctype.h>
4. #include <string.h>
5. #define EOI0/* end of input symbol*/
6. #define NAME1/* identifier */
7. #define STRING2/* string constant ".....”*/
8. #define ICON3/* integer or character constant*/
9. #define FCON4/* floating point constant */
10. #define PLUS5/* + */
11. #define MINUS6/* - */
12. #define STAR7/* * */
13. #define AND8/* & */
14. #define QUEST9/* ? */
15. #define COLON10/* : */
16. #define ANDAND11/* && */
17. #define OROR12/* || */
18. #define RELOP13/* < <= > >= */
19. #define EQUOP14/* == != */
20. #define DIVOP15/* / % */
21. #define OR16/* | */
22. #define XOR17/* ^ */
23. #define SHIFTOP18/* >> << */
24. #define INCOP19/* ++-- */
25. #define UNOP20/* ! ~ */
26. #define STRUCTOP21/* . -> */
27. #define TYPE22/* int, long, etc. */
28. #define CLASS23/* extern,static, typedef,etc. */
29. #define STRUCT24/* struct, union*/
30. #define RETURN25/* return */
31. #define GOTO26/* goto */
```

```
32. #define IF27/* if */
33. #define ELSE28/* else */
34. #define SWITCH29/* switch */
35. #define BREAK30/* break */
36. #define CONTINUE31/* continue */
37. #define WHILE32/* while */
38. #define DO33/* do */
39. #define FOR34/* for */
40. #define DEFAULT35/* default */
41. #define CASE36/* case */
42. #define SIZEOF37/* sizeof */
43. #define LP38/* ( left prenthesis */
44. #define RP39/* ) */
45. #define LC40/* { */
46. #define RC41/* } */
47. #define LB42/* \[ */
48. #define RB43/* \] */
49. #define COMMA44/* , */
50. #define SEMI45/* ; */
51. #define EQUAL46/* = */
52. #define ASSIGNOP47/* += -= etc. */
53. /* The following definitions are used for preprocess symbols */
54. #define JINGHAO48/* # */
55. #define INCLUDE49/* include */
56. #define DEFINE50/* define */
57. #define IFDEF51/* ifdef */
58. #define IFNDEF52/* ifndef */
59. #define ENDIF53/* endif */
60. /* The following definitions are used for ′ ′ ,′ \t′ ,′ \n′ etc. */
61. #define WHITE200
62. #define ERRORCHAR300/* error characters */
63. int idorkeyword(char *lx);
64. %}
65. /* The following definitions are macros */
66. letter\[A-Za-z\]
67. alnum\[A-Za-z0-9\]
68. h\[0-9a-fA-F\]
69. o\[0-7\]
```

```
70. d\[0-9\]
71. suffix\[UuLl\]
72. white\[ \t\n\040\]
73. %startCOMMENT/* used for comment lines */
74. %%
75. /******** THE RECOGNISING RULES *****************/
76. "/*"{printf("%s",yytext);BEGIN COMMENT;}
77. <COMMENT>\n{printf("%s",yytext);}
78. <COMMENT>"*/"{printf("%s",yytext);BEGIN 0;}
79. <COMMENT>.{printf("%s",yytext);}
80.  \"(\\.|\[^\"\])*\"return STRING;
81.  \"(\\.|\[^\"\])*\[\r\n\] {printf("Adding missing \"to string constant\n");
82. return STRING;}
83. '.' return ICON;
84. '\\.' return ICON;
85. '\\{o}({o}{o}?)?' return ICON;
86. '\\\[xX\]{h}({h}{h}?)?' return ICON;
87. 0{o}*{suffix}?return ICON;
88. 0\[xX\]{h}+{suffix}?return ICON;
89. \[1-9\]{d}*{suffix}?return ICON;
90. (\[1-9\]{d}*|{d}+\.{d}*|{d}*\.{d}+)(\[eE\]\[-+\]?{d}+)?\[fF\]? return FCON;
91. "("return LP;
92. ")"return RP;
93. "{"return LC;
94. "}"return RC;
95. "\["return LB;
96. "\]"return RB;
97. ("->")|(".")return STRUCTOP;
98. ("++")|("--")return INCOP;
99. \[/%\]return DIVOP;
100. \[~!\]return UNOP;
101.  \<\<|\>\>return SHIFTOP;
102. \[<>\]=?return RELOP;
103. \[!=\]=return EQUOP;
104. \[-+*/%&|\^\]=return ASSIGNOP;
105. (\<\<|\>\>)=return ASSIGNOP;
106. "*"return STAR;
107. "+"return PLUS;
```

```
108. "-"return MINUS;
109. "="return EQUAL;
110. "&"return AND;
111. "^"return XOR;
112. "|"return OR;
113. "&&"return ANDAND;
114. "||"return OROR;
115. "?"return QUEST;
116. ":"return COLON;
117. ","return COMMA;
118. ";"return SEMI;
119. {letter}{alnum}*return idorkeyword(yytext);
120. "#"return JINGHAO;
121. {white}+return WHITE;/* Ignore white space */
122. .{printf(" Invalid char %s＼n",yytext);
123. return ERRORCHAR;}
124. %%
125. /* beginning of auxiliary functions */
126. typedef struct{char *name; int val;} KWRDSTRUCT;
127. KWRDSTRUCT Ktab\[\]=
128. {
129. {"auto",CLASS},
130. {"break",BREAK},
131. {"case",CASE},
132. {"char",TYPE},
133. {"continue",CONTINUE},
134. {"default",DEFAULT},
135. {"do",DO},
136. {"double",TYPE},
137. {"else",ELSE},
138. {"extern",CLASS},
139. {"float",TYPE},
140. {"for",FOR},
141. {"goto",GOTO},
142. {"if",IF},
143. {"int",TYPE},
144. {"long",TYPE},
145. {"register",CLASS},
```

```
{"return",RETURN},
{"short",TYPE},
{"sizeof",SIZEOF},
{"static",CLASS},
{"struct",STRUCT},
{"switch",SWITCH},
{"typedef",CLASS},
{"union",STRUCT},
{"unsigned",TYPE},
{"void",TYPE},
{"while",WHILE},
/****************************************
The following definitions are used for preprocess *
****************************************/
{"include",INCLUDE},
{"define",DEFINE},
{"ifdef",IFDEF},
{"ifndef",IFNDEF},
{"endif",ENDIF}
} ;
/* bsrch(): check if a is member of tab */
KWRDSTRUCT *bsrch(KWRDSTRUCT *a,KWRDSTRUCT *tab,int number)
{
int i;
for(i=0;i<number;i++)
if(!strcmp(a->name,tab\[i\].name))return &tab\[i\]; /* it is a keyword */
return NULL; /*It is a identifier*/
}
/* idorkeyword(): Do a binary search for a possible Keyword in Ktab.
Return the token if it's in the table;return NAME otherwise */
int idorkeyword(char *lx)
{
KWRDSTRUCT *p;
KWRDSTRUCT dummy;
static int number=sizeof(Ktab)/sizeof(KWRDSTRUCT);
dummy.name=lx;
p=bsrch(&dummy,Ktab,number);
return(p ? p->val:NAME);
```

```
184. }
185. /* The following functions are used only for testing */
186. writeout(int c,char *text)
187. {
188. printf("(");
189. switch(c)
190. {
191. case 0: printf("EOI, %s ) ",text); break;
192. case 1: printf("NAME, %s ) ",text); break;
193. case 2: printf("STRING, %s ) ",text); break;
194. case 3: printf("ICON, %s ) ",text); break;
195. case 4: printf("FCON, %s ) ",text); break;
196. case 5: printf("PLUS, %s ) ",text); break;
197. case 6: printf("MINUS, %s ) ",text); break;
198. case 7: printf("STAR, %s ) ",text); break;
199. case 8: printf("AND, %s ) ",text); break;
200. case 9: printf("QUEST, %s ) ",text); break;
201. case 10: printf("COLON, %s ) ",text); break;
202. case 11: printf("ANDAND, %s ) ",text); break;
203. case 12: printf("OROR, %s ) ",text); break;
204. case 13: printf("RELOP, %s ) ",text); break;
205. case 14: printf("EQUOP, %s ) ",text); break;
206. case 15: printf("DIVOP, %s ) ",text); break;
207. case 16: printf("OR, %s ) ",text); break;
208. case 17: printf("XOR, %s ) ",text); break;
209. case 18: printf("SHIFTOP, %s ) ",text); break;
210. case 19: printf("INCOP, %s ) ",text); break;
211. case 20: printf("UNOP, %s ) ",text); break;
212. case 21: printf("STRUCTOP, %s ) ",text); break;
213. case 22: printf("TYPE, %s ) ",text); break;
214. case 23: printf("CLASS, %s ) ",text); break;
215. case 24: printf("STRUCT, %s ) ",text); break;
216. case 25: printf("RETURN, %s ) ",text); break;
217. case 26: printf("GOTO, %s ) ",text); break;
218. case 27: printf("IF, %s ) ",text); break;
219. case 28: printf("ELSE, %s ) ",text); break;
220. case 29: printf("SWITCH, %s ) ",text); break;
221. case 30: printf("BREAK, %s ) ",text); break;
```

```
222. case 31: printf("CONTINUE, %s ) ",text); break;
223. case 32: printf("WHILE, %s ) ",text); break;
224. case 33: printf("DO, %s ) ",text); break;
225. case 34: printf("FOR, %s ) ",text); break;
226. case 35: printf("DEFAULT, %s ) ",text); break;
227. case 36: printf("CASE, %s ) ",text); break;
228. case 37: printf("SIZEOF, %s ) ",text); break;
229. case 38: printf("LP, %s ) ",text); break;
230. case 39: printf("RP, %s ) ",text); break;
231. case 40: printf("LC, %s ) ",text); break;
232. case 41: printf("RC, %s ) ",text); break;
233. case 42: printf("LB, %s ) ",text); break;
234. case 43: printf("RB, %s ) ",text); break;
235. case 44: printf("COMMA, %s ) ",text); break;
236. case 45: printf("SEMI, %s ) ",text); break;
237. case 46: printf("EQUAL, %s ) ",text); break;
238. case 47: printf("ASSIGNOP, %s ) ",text); break;
239. /* The following definitions are used for preprocess symbols */
240. case 48: printf("JINGHAO, %s ) ",text); break;
241. case 49: printf("INCLUDE, %s ) ",text); break;
242. case 50: printf("DEFINE, %s ) ",text); break;
243. case 51: printf("IFDEF, %s ) ",text); break;
244. case 52: printf("IFNDEF, %s ) ",text); break;
245. case 53: printf("ENDIF, %s ) ",text); break;
246. default:break;
247. }
248. }
249. void main(int argc,char **argv)
250. {
251. int c;
252. if(argc>=2)
253. {
254. if((yyin=fopen(argv\[1\],"r"))==NULL)
255. {printf("Can't open file %s＼n",argv\[1\]);exit(0);}
256. }
257. while(c=yylex())
258. {static j=0;
259. if(c<200){writeout(c,yytext);j++;}
```

260. else continue;

261. if(j == j/4*4)printf("＼n");

262. }

263. return;

264. }

为便于阅读，特作以下说明。

(1) 在文件 **c.lex** 中，从第 **5** 行至第 **62** 行列出了 **C** 语言各类单词的名字及其类别码的定义，其中还包含了输入结束标志、空白字符定义名、错误字符定义名，以及用于 **C** 编译预处理的若干符号的名字(仅用于调试，实际上经预处理后的 **C** 程序不含此类单词)。

(2) 第 **66** 行至第 **72** 行所列的是一组用于定义正规式的宏定义。其中，宏 **suffix** 用来代表数值常数的后缀字符(例如，字符 **L** 表示数据为长整型常数等)；宏 **white** 代表制表、换行及空格三个字符中的任一字符，即把它们均视为“空白字符”。

(3) 第 **73** 行定义了一个开始条件名 **COMMENT**，它与第 **76** 行至第 **79** 行定义的识别规则联合使用，可对 **C** 程序中的注释行进行处理。

(4) 第 **80** 行用于识别处理字符串常量；第 **81** 行的语义动作描述了当扫描器发现一个字符串尚未结束，而遇到回车字符时的处理工作(在 **C** 语言中这是一个词法错误)。

(5) 第 **83** 行至第 **89** 行的词型用于识别整型常数(包括十进制、八进制、十六进制的整型及长整型常数)。第 **90** 行的词型用于识别浮点型(包括科学记数法表示的)常数。

(6) 第 **119** 行用于识别和处理关键字和标识符。具体的作法是：若该词型和当前的词文匹配，则以此词文(存放在全局变量 **yytext\[\]**中)为实参调用函数 **idorkeyword()**(见第 **176** 行至第 **184** 行的程序)，此函数在第 **127** 行到第 **165** 行定义的关键字表 **Ktab** 中(通过调用第 **167** 行至第 **173** 行定义的 **bsrch()**函数)进行查找，若在表中找到该单词，则回送相应的关键字及其类别码；否则，回送类别码 **NAME**(值为 **1**，定义见第 **6** 行)。

(7) 第 **186** 行至第 **264** 行给出了用于调试此 **LEX** 源文件的程序段，它从用户指定的文件或标准输入逐一读取字符，并以有序对的形式输出所识别的每一个单词的类别码和词文。

用 Lex(flex)和 yacc(bison)写的简单计算器

Lex 文件如下：

```
  %{
#include "cal.tab.h"
%}
%option noyywrap
integer      [0-9]+
dreal        ([0-9]*"."[0-9]+)
ereal        ([0-9]*"."[0-9]+[EedD][+-]?[0-9]+)
real         {dreal}|{ereal}
```

```
nl          \n
plus         "+"
minus        "-"
times        "*"
divide       "/"
lp          "("
rp          ")"
module       "%"
power        "^"
%%
[ \t]       ;   /*skip any blanks */
{integer}     { sscanf(yytext, "%d", &yylval.integer);
              return INTEGER;
             }
{real}        { sscanf(yytext, "%lf", &yylval.real);
/*yylval = atof(yytext);   it doesn't work under MSVSC*/
              return REAL;
             }
{plus}       { return PLUS;}
{minus}      { return MINUS;}
{times}      { return TIMES;}
{divide}     { return DIVIDE;}
{module}     { return MODULE;}
{power}      { return POWER;}
{lp}         { return LP;}
{rp}         { return RP;}
{nl}         { return NL;}
.           { return yytext[0];}
```

以上是 Lex 文件的代码（cal.l),lex 是用来得到 token。

有了 token 之后呢，就用 yacc（本人用的是 GNU 的可以在 windows 下面运行的 bison）

才处理这些符号。也就是写出一个个的状态，最后得到分析结果。

下面是 yacc 文件的代码（cal.y):

```
%{
#include   <stdio.h>
#include   <math.h>


%}
%union{ double    real; /* real value */
        int    integer; /* integer value */
      }
%token <real> REAL
%token <integer> INTEGER

%start   lines
%token NUMBER NL
%token PLUS MINUS TIMES DIVIDE MODULE POWER LP RP</P< p>

%type <real>   rexpr
%type <integer> iexpr


%left PLUS MINUS      /*left associative */
%left TIMES DIVIDE MODULE /*left associative */
%left POWER
%left UNARYMINUS

%%
lines: /* nothing */
```

```
    | lines line NL
    | lines error NL
      { yyerror();yyerrok; }
    ;
line : iexpr
       {printf("%d\n",$1);}
    |  rexpr
       {printf("%lf\n",$1);}
    ;
iexpr: INTEGER
     { $$ = $1; }
    | iexpr PLUS iexpr
     { $$ = $1 + $3;}
    | iexpr MINUS iexpr
     { $$ = $1 - $3;}
    | iexpr TIMES iexpr
     { $$ = $1 * $3;}
    | iexpr DIVIDE iexpr
     { if($3)
       $$ = $1 / $3;
       else
       {
       $$ = $1;
       printf (stderr, "%d.%d-%d.%d: division by zero",
                       @3.first_line, @3.first_column,
```

```
                    @3.last_line, @3.last_column);
    }
    }
  | iexpr MODULE iexpr
    { $$ = $1 % $3; }
  | iexpr POWER iexpr
    { $$ = pow($1, $3);}
  | MINUS iexpr %prec UNARYMINUS
    { $$ = - $2;}
  | LP iexpr RP
    { $$ = $2;}
  | LP iexpr error
    { $$ = $2; yyerror("missing ')'"); yyerrok;}
  | PLUS iexpr %prec UNARYMINUS
    { $$ = $2;}
  ;  </P< p>

rexpr :REAL
      { $$ = $1; }
  | rexpr PLUS rexpr
      { $$ = $1 + $3; }
  | rexpr MINUS rexpr
      { $$ = $1 - $3; }
  | rexpr TIMES rexpr
      { $$ = $1 * $3; }
  | rexpr DIVIDE rexpr
      {
        if ($3)
          $$ = $1 / $3;
        else
```

```
        {
            $$ = $1;
            printf (stderr, "%d.%d-%d.%d: division by zero",
                    @3.first_line, @3.first_column,
                    @3.last_line, @3.last_column);
         }
       }
    | rexpr POWER rexpr
      { $$ = pow($1,$3); }
    | LP rexpr RP
      { $$ = $2; }
    | LP rexpr error
      { $$ = $2; yyerror("missing ')'"); yyerrok;}
    | MINUS rexpr %prec UNARYMINUS
      { $$ = -$2; }
    | PLUS rexpr %prec UNARYMINUS
      { $$ = $2;}

    | iexpr PLUS rexpr
      { $$ = (double)$1 + $3;}
    | iexpr MINUS rexpr
      { $$ = (double)$1 - $3;}
    | iexpr TIMES rexpr
      { $$ = (double)$1 * $3;}
    | iexpr DIVIDE rexpr
      { if($3)
        $$ = (double)$1 / $3;
       else
      { $$ = $1;
            printf (stderr, "%d.%d-%d.%d: division by zero",
                    @3.first_line, @3.first_column,
                    @3.last_line, @3.last_column);
      }
      }
    | iexpr POWER rexpr
      { $$ = pow((double)$1,$3); }
    | rexpr PLUS iexpr
      { $$ = $1 + (double)$3;}
```

```
    | rexpr MINUS iexpr
      { $$ = $1 - (double)$3;}
    | rexpr TIMES iexpr
      { $$ = $1 * (double)$3;}
    | rexpr DIVIDE iexpr
      { if($3)
        $$ = $1 / (double)$3;
       else
        { $$ = $1;
          printf (stderr, "%d.%d-%d.%d: division by zero",
                  @3.first_line, @3.first_column,
                  @3.last_line, @3.last_column);
        }
      }
    | rexpr POWER iexpr
      { $$ = pow($1,(double)$3); }</P< p>

    ;
%%

void  main()
{
    yyparse();
}

int  yyerror(char*  msg)
{
     printf("Error:  %s  encountered  \n",  msg);
}
```

这样一个支持 + ， - ，× ，/ ，^ ，以及括号运算的计算器就做成了。所用时间不会超过半个小时，如果用 c，或 c + + 写个算符优先文法的话可是一个不小的工程。由此可见 lex 和 yacc 的魅力了!

编译命令是：fLex cal.l

bison -d -v cal.y

pause

/P>

$$$ 简易计算器 $$$

一、功能

实现浮点数的四则运算；

二、需要的文件

Lex 文件：calculater.l

Yacc 文件：calculater.y

三、编译的方法

lex calculater.l// 产生 lex.yy.c

yacc -d calculater.y// 产生 y.tab.h 和 y.tab.c

gcc y.tab.c -ll// 产生 a.out

四、Lex 文件

```
/*calculater.l*/
%{                                          // 括号中的内容直接复制到
lex.yy.c
#include "y.tab.h"                          // 包含了一些记号的定义，比如
NUM
%}
```

```
Number    [0-9]+(\.[0-9]+)?                    // 正则定义

%%                                             // 转换规则
{number}{
yylval = atof(yytext);                         // yylval 保存记号的属性值，yytext
是输入的字符流
return NUM;                                    // yylex()返回的是记号
}
[ \t]+{}                                       // 空格和制表符，什么都不做
\n|.{return *yytext;}                          // .匹配除换行符以外的任何字符，这个
规则必须放// 在最后，不然会屏蔽掉其他的规则
%%
```

五、Yacc 文件

```
/*calculater.y*/
%{
#include <stdio.h>
#define YYSTYPE double                         // Yacc 堆栈采用 double 类型
%}

%token NUM                                     // 记号声明
%left '+' '-'                                  // 为终结符指定结合规则和优先
级
%left '*' '/'                                  // 为了处理二义文法带来的冲突
// 排列规则：优先级别低的位于上面，高的位于下面

%%                                 // 以下是，包含语义动作的上下文无关文法
line:list '\n' { exit(0); }                    // 采用的是二义文法，因其简单，且能被
Yacc 处理
;
list:list expr ';' { printf("%g\n", $2); }
|
;
expr:expr '+' expr{ $$ = $1 + $3; }            // 左边的非终结符是$$，右边的
文法符号对应
                |expr '-' expr{ $$ = $1 - $3; }                    // 于：$1、$2、
```

```
$3……
                    |expr '*' expr{ $$ = $1 * $3; }
                    |expr '/' expr { $$ = $1 / $3; }
                    |'(' expr ')'{ $$ = $2; }
                    |'-' expr{ $$ = -$2; }
                    |NUM
;
%%

yyerror(char *m)                    // yyerror()、yylex()、main()必须被定义
{                                   // 否则编译不成功
fprintf(stderr, "error: %s\n", m);
}

main()
{
yyparse();
}
#include "lex.yy.c"
```

$$$ 中缀表达式 --〉后缀表达式 $$$

一、概述

表达式的构成和简易计算器相同；

二、需要的文件

Lex 文件：calculater.l（和简易计算器相同）

Yacc 文件：calculater.y

三、Yacc 文件

```
/*calculater.y*/
%{
#include <stdio.h>
#define YYSTYPE double
%}

%token NUM
```

```
%left '+' '-'
%left '*' '/'

%%
line:list '\n' { exit(0); }
;
list:list expr ';' { printf("\n"); }
|
;
expr:expr '+' expr      { printf("+ "); }
|expr '-' expr      { printf("- "); }
|expr '*' expr      { printf("* "); }
|expr '/' expr       { printf("/ "); }
|'(' expr ')'
|NUM              { printf("%g ", yylval); }
;
%%

yyerror(char *m)
{
fprintf(stderr, "error: %s\n", m);
}

main()
{
yyparse();
}
#include "lex.yy.c"
```

四、总结

Yacc 文件中的文法部分完全相同，仅仅是“语义动作”不同而已。

参考文献

1. 陶英华，韩美琦等。实例图解 Red Hat Linux9 应用指南.中国水利水电出版社。2003.6
2. 徐虹，何嘉，张钟 等，操作系统实验指导—基于 Linux 内核。清华大学出版社 2004.11
3. 刘磊等，编译原理及实现技术。机械工业出版社。2005.7