Lex 和 Yacc 从入门到精通

熊春雷　翻译

Abstract

在开发程序的过程中经常会遇到文本解析的问题，例如：解析 C 语言源程序，编写 脚本引擎等等，解决这种文本解析的方法有很多，一种方法就是自己手动用 C 或者 C++直接编写解析程序，这对于简单格式的文本信息来说，不会是什么问题，但是 对于稍微复杂一点的文本信息的解析来说，手工编写解析器将会是一件漫长痛苦 而容易出错的事情。本系列文档就是专门用来由浅入深的介绍两个有名的 Unix 工 具 Lex 和 Yacc，并会一步一步的详细解释如何用这两个工具来实现我们想要的任何 功能的解析程序，为了方便理解和应用，我会在该系列的文章中尽可能的采用具 体可行的实例来加以阐释，而且这种实例都是尽可能的和具体的系统平台无关的 ，因此我采用命令行程序作为我们的解析程序的最终结果。

1、环境配置篇

开发 Lex 和 Yacc 程序最需要的程序就是 lex 和 yacc 了，如果你是 Unix 或者 Linux 系统，则 系统自带了这两个工具，无需安装，不过值得说明的是 GNU/Linux 下面的 Lex 是 flex，而 Yacc 则是 bison。另外需要的就是一个 C/C++ 语言编译器，由于我们采用的是 GNU 的 lex 和 yacc，所以，理所当然的我们就使用 GNU 的编译器了，如果是 Unix 或者 Linux 系统 ，那么编译器应该已经安装了。在这里我重点讨论的是 Windows 系统环境下的 Lex 和 Yacc 程序的开发，至于为什么选择 Windows 系统作为开发平台，则是为了尽可能的让初 学者容易入门。

1.1. 必备工具

言归正传，首先列举 Windows 平台下面 Lex 和 Yacc 开发环境所需要安装的程序：

1. Lex(flex.exe)和 Yacc(bison.exe)环境
2. C/C++编译器

1.2. flex 和 bison

值得说明的是，flex.exe和bison.exe是UnxUtils包中的文件，已经将许多 Unix/Linux平台的程序都移植到了Windows平台，可以直接到UnxUtils网站下载，下载解压缩之后在系统的PATH环境变量中增加UnxUtils所有的exe文件所在的目录，使 得DOS命令行可以直接搜索到flex.exe和bison.exe，除此之外还需要从网络上下载 bison需要的bison.simple和bison.hairy两个文件，并且还要分别设置环境变量 BISON_HAIRY指向bison.hairy，BISON_SIMPLE指向bison.simple。

Tip

如果觉得麻烦也可以直接使用我做好的flex和bison环境，点击这里下载。

解压缩 lexyacc.rar 之后运行里面的 lexyacc.bat 文件就会得到一个 lex 和 yacc 环境， 下图是简单的运行结果：

1.3. C/C++编译器

由于我们使用的flex和bison都是GNU的工具，所以为了方便，采用的C/C++ 编译器也 采用GNU的编译器GCC，当然我们需要的也是Windows版本的GCC了。目前Windows平台 的GCC主要是MinGW编译器，可以到 MinGW的主页下载安装。

Important

注意安装了 MinGW 之后一定要将安装后的 MinGW 的 bin 路径设置到环境变量 PATH 中。

我们在下一章里面将会真正的接触到 Lex 和 Yacc 的具体内容，敬请关注:)

2、正则表达式篇

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：

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。

3、一个极其简单的 lex 和 yacc 程序

摘要

在 本章中，将会首先给出一个最基本的 lex 和 yacc 联合使用的框架，这个基本框架 最主要的特点就是能够正确的被编译。在我学习 lex 和 yacc 的过程中经历了无数次 的痛苦折磨，我发现一个一开始足够简单而且能够被正确编译的例子往往能够使 学习者增加学习的兴趣和信心。因此我的所有的文章都尽可能的采用这种方式进 行描述。我写这些文档的最大的愿望就是希望能够减少新手学习的痛苦。希望自 己能够做到这一点！

1. 基本的 lex 文件

例 3.1. frame.l

```
%{
int yywrap(void);
%}
%%
%%
int yywrap(void)
{
        return 1;
}
```

lex 文件和 yacc 文件都是被%%分成了上中下三个部分，在这个程序中的 yywrap 函数 需要说明一下：

yywrap

lex 源文件中的 yywrap 函数是必须的！具体的原因就是因为给了这个函数实 现之后就可以不需要依赖 flex 库了。具体 yywrap 的作用会在后面

的章节应 用的时候进行解释。通常的做法就是直接返回 1，表示输入已经结束了。

2. 基本的 yacc 文件

例 3.2. frame.y

```
%{
void yyerror(const char *s);
%}
%%
program:
        ;
%%
void yyerror(const char *s)
{
}
int main()
{
        yyparse();
        return 0;
}
```

如前所述，yacc 文件被%%分成了上中下三个部分，在这个程序中有几个需要说明 的地方：

program

这 是语法规则里面的第一个非终结符，注意上面的格式哦：“program”后 面紧跟着一个冒号“:”，然后换行之后有一个分号“;”，这表明这个 program 是由空串组成的。至于什么是非终结符以及什么是终结符，还有什 么是语法规则都会在后面的章节中进行详细介 绍。

yyerror

从字面上就可以看出是一个处理错误的函数，在这里为空的原因是为了保证代码尽可能的简洁！实际上这个函数里面的代码通常只有一句输出语句 ，当然如果你喜欢还可以加入纠错代码，使你的解析器具备纠错能力:)

yyparse

其 实这个函数是 yacc 生成的，所以你在代码里面可以直接使用。这个时候 你可能会问：“yacc 生成了 yyparse 函数，那么 lex 是不是也生成了什么函 数呢？”，是的，lex 生成的函数为 yylex 函数。实际上 yyparse 还间接调用 了 yylex 函数，可以在生成的 C 源文件中去核实。

main

每一个 C/C++程序都必须的装备啊，少了怎么能行呢:)所以这个 main 函数你 可以放到任何的地方，当然要保证能够调用 yyparse 就可以了。但是通常的 做法就是将 main 函数放到 yacc 文件中。

从 上面的 yacc 文件中还可以看出被%%分割成为的三个部分，第一部分中要写入 C/C++代码必须用%{和%}括起来；但是第三个部分就可以直接写入 C/C++代码了 ，不需要任何的修饰；中间的那一部分就是 yacc 语法规则了。为了能够让这个 最最简单的 yacc 源程序能够通过 bison 的编译必须要提供一个语法规则，这里给出了一个最简单的规则：一个 program 就是由空字符串构成的。实际上等于什么也没有做。呵呵，对啊，本章的目的就是为了能够编译通过 lex 和 yacc 源程 序，并且也能够被 C/C++编译器编译通过啊。现在是不是已经真的编译通过了呢 ，可以按照下面的编译步骤一步一步的来编译核实。

提示

对 yacc 的描述同样也适用于 lex。

lex 就是词法扫描器，yacc 就是语法分析器，这是通用的说法；具体的实现有所不同 GNU 的 lex 就是 flex，GNU 的 yacc 就是 bison。为了统一，所以在后面的文章 中就只会用 lex 来表达词法扫描器，用 yacc 来表达语法分析器啦!

3. 用 C 语言编译器编译

下面是编译全过程记录，采用了我在第一章中所制作的 lex 和 yacc 转换环境：

```
D:\work\lex_yacc\chapter03>dir
 驱动器 D 中的卷是 工作区
 卷的序列号是 54D0-5FC0

 D:\work\lex_yacc\chapter03 的目录

2006-09-25  20:27    <DIR>          .
2006-09-25  20:27    <DIR>          ..
2006-09-25  20:07                71 frame.l
2006-09-25  20:20               144 frame.y
               2 个文件            215 字节
               2 个目录  7,785,578,496 可用字节

D:\work\lex_yacc\chapter03>flex frame.l

D:\work\lex_yacc\chapter03>dir
 驱动器 D 中的卷是 工作区
 卷的序列号是 54D0-5FC0

 D:\work\lex_yacc\chapter03 的目录
```

```
2006-09-25  20:28    <DIR>          .
2006-09-25  20:28    <DIR>          ..
2006-09-25  20:07                71 frame.l
2006-09-25  20:20               144 frame.y
2006-09-25  20:28            36,997 lex.yy.c
               3 个文件         37,212 字节
               2 个目录  7,785,537,536 可用字节

D:\work\lex_yacc\chapter03>bison -d frame.y

D:\work\lex_yacc\chapter03>dir
 驱动器 D 中的卷是 工作区
 卷的序列号是 54D0-5FC0

 D:\work\lex_yacc\chapter03 的目录

2006-09-25  20:28    <DIR>          .
2006-09-25  20:28    <DIR>          ..
2006-09-25  20:07                71 frame.l
2006-09-25  20:28            19,416 frame.tab.c
2006-09-25  20:28                74 frame.tab.h
```

```
2006-09-25  20:20               144 frame.y
2006-09-25  20:28            36,997 lex.yy.c
               5 个文件         56,702 字节
               2 个目录  7,785,517,056 可用字节

D:\work\lex_yacc\chapter03>
```

过程 3.1. 总的来说就是如下的几个步骤：

1. 将前面的例子frame.l和frame.y保存成为相应的文件
2. **flex frame.l**
3. **bison frame.y**
4. **gcc frame.tab.c lex.yy.c**

提示

实际上经过 flex 和 bison 的转换之后的 C/C++源程序是可以直接在 VC 里面使用的！

上面的 frame.tab.c 是由 bison 编译 frame.y 产生的，而 lex.yy.c 则是由 flex 编译 frame.l 产生的。

好了，一个最简单的 lex 和 yacc 程序已经完备了，因此这一章的目的也就已经达到了。在下一章里面将会对这里的框架例子进行扩充以适应自己特殊的需要，逐步逐步的实 现一个分析 C/C++源代码的工具程序，但是每一章的结尾都会尽可能的给出一个可以编 译通过的 lex 和 yacc 源程序。本来也想给出一个计算器的源程序作为例子的，但是这样 的资料已经很多了。这些资料往往不能够让自己说清楚问题，在自己的开发中还是会 遇到千奇百怪的问题，因此为了让自己能够有机会解决一个新手在开发新程序中可能 出现的问题，我也就找了一个我没有开发过的程序来让自己一步一步的解决这些问题 。我想这种方式也许是比较好的学习方式吧:)

本章完！

4、解析 C/C++包含文件

摘要

在这一章里面将要涉及到处理 C/C++的包含宏的解析。也就是说要从一大串 C/C++ 包含文件的声明中提取出文件名，以及相互依赖关系等等。实际上在这一章里面 使用的 Lex 和 Yacc 技术也是非常重要的，这些都会在本章中进行详细讲解。

我们知道对于 C/C++包含文件声明是为程序提供了一些库存的功能，因此存在一种依赖关系，如果把这种依赖关系表达成为 Makefile 的形式，那么就可以自动生成 Makefile。在这一章里面并不会实现自动生成 Makefile 的功能，而是仅仅解析出所有的包含文件名，并记录下来。

1. 分析

我 们知道 C/C++中存在两种形式的包含文件，一种是用“<>”包含的头文件， 一种是“""”包含的头文件，这两种不同的形式表达了头文件的不同的搜索方式。另外还需要注意的是：这两种方式包含的都是磁盘上存在的文件名。也就是说，只 要是磁盘上存在的文件名都可以包含的，都是合法的，因而 C/C++里面存在的有扩展 名的头文件和没有扩展名的头文件都是合法的。并且还需要注意的是 C/C++包含的头 文件是可以续行的。

因而总结起来需要做到如下的几件事情：

1. 处理“<>”和“""”两种包含方式
2. 处理文件名
3. 处理续行

2. Lex 文件

```
%{
#include "main.hpp"// 在其中保存了记录头文件所需要的所有数据结构
#include "frame.tab.h"// 由Yacc自动生成的所有标记声明，实际上都是C宏
extern "C"{
int yywrap(void);
int yylex(void);
}
%}
%x _INCLUDE_
%x _INCLUDE_FILE_
```

```
%%
"#"[ \t]*"include"      {
                            BEGIN _INCLUDE_;// 进入_INCLUDE_状态
                            yylval.clear();// 需要将所有的Include
值初始化
                            return INCLUDE;// 返回INCLUDE标记
                        }
<_INCLUDE_>[\"|<]       {
                            BEGIN _INCLUDE_FILE_;// 进入
_INCLUDE_FILE_状态
                            return *yytext; // 返回引号或者尖括号
                        }
<_INCLUDE_FILE_>[^\">]* {
                            yylval.headerfile+=yytext;// 记录头文
件字符串
                            return HEADERFILE;// 返回头文件标记
                        }
<_INCLUDE_FILE_>[\"|>]  {
                            BEGIN INITIAL;// 恢复到初始状态，默认
状态
                            return *yytext;// 返回引号或者尖括号
                        }
[ \t\n]                 ;// 对于额外的空白都不处理直接扔掉
%%
int yywrap(void)
{
        return 1;// 只处理一个输入文件
}
```

3. Yacc 文件

```
%{
#include <iostream>
#include "main.hpp"
#define YYDEBUG 0 // 将这个变量设置为 1 则表示启动Yacc的调试功能
extern "C"{
void yyerror(const char *s);
extern int yylex(void);
}
std::vector<Include> g_Includes;// 用来记录所有的包含声明
Include             *g_pInclude;// 用来保存新增的包含声明信息的指针
%}
%token INCLUDE
```

```
%token HEADERFILE
%%
program:/* empty */
        | program include_preprocess // 用这种递归的方式从有限的标记表达出无限的内容
        ;
include_preprocess:
          INCLUDE '<'  HEADERFILE '>'
        {
                // 注意这里的$3，实际上就是上面的标记的第三个的意思
              // 因为yylval被声明为Include结构，参见main.hpp文件
              // 因而每个标记都是Include结构类型。
              g_Includes.push_back(Include());
              g_pInclude = &g_Includes.back();
              g_pInclude->clear();// 初始化
              g_pInclude->headerfile   = $3.headerfile;// 可以证明$3的类型就是Include类型
              g_pInclude->is_angle     = true;// 是尖括号
              g_pInclude->is_quotation = false;// 不是双引号
        }
        | INCLUDE '\"' HEADERFILE '\"'
        {
                // 值得说明的是：上面的include_preprocess用$表示，
              // 而不是用$0 表示。从左向右依次为：
              // include_preprocess        $
              // INCLUDE                   $1
              // '\"'                      $2
              // HEADERFILE                $3
              // '\"'                      $4
              g_Includes.push_back(Include());
              g_pInclude = &g_Includes.back();
              g_pInclude->clear();// 初始化
              g_pInclude->headerfile   = $3.headerfile;
              g_pInclude->is_angle     = false;// 不是尖括号
              g_pInclude->is_quotation = true;// 是双引号
        }
        ;
%%
void yyerror(const char *s)
{
        std::cerr<< s << std::endl;
}
int main()
{
```

```
#if YYDEBUG
        yydebug = 1;
#endif//YYDEBUG
        yyparse();// 进行语法分析，这个函数是Yacc自动生成的
        // 下面的这行代码仅仅使用了STL的输出算法到标准输出

std::copy(g_Includes.begin(),g_Includes.end(),std::ostream_iterator<I
nclude>(std::cout,"\n"));
        return 0;
}
```

4. main.hpp 文件

```
#pragma once

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#include <iterator>

// 对于每一个项目最好都用一个独立的数据结构来保存相应的信息

struct Include

{

        void clear();// 设置Include的初始值

        std::string headerfile;// 记录头文件全名（包括路径）

        bool is_quotation;// 是否是双引号""括起来的头文件

        bool is_angle;// 是否是尖括号<>括起来的头文件

        // 下面的这个函数仅仅是用来输出到C++流而准备的

        friend std::ostream&operator<<(std::ostream&s,const Include&I);
```

```
};

std::ostream&operator<<(std::ostream&s,const Include&I);

// 下面的这个宏定义用来取消 Lex 和 Yacc 默认的 YYSTYPE 定义，因为默认的
YYSTYPE 定义

// 仅仅只能够记录整数信息，因此要保存额外的信息必须这样定义宏，可以参
见 Yacc

// 自动生成的标记头文件 frame.tab.h。

#define YYSTYPE Include
```

5. main.cpp 文件

```
#include "main.hpp"

// 初始化所有的 Include 信息，避免前后关联

void Include::clear()

{

        headerfile.clear();

        is_quotation = false;

        is_angle = false;

}

// 为了能够方便输出，在这里直接准备好了一个流输出函数

std::ostream&operator<<(std::ostream&s,const Include&I)

{

        if(I.is_angle)

                s << "采用尖括号的" ;
```

```
        if(I.is_quotation)
                s << "采用双引号的" ;
        s << "头文件：[" << I.headerfile << "]" ;
        return s;
}
```

6. Makefile 文件

```
LEX=flex
YACC=bison
CC=g++
a.exe:lex.yy.o frame.tab.o main.o
        $(CC) lex.yy.o frame.tab.o main.o -o a.exe
lex.yy.o:lex.yy.c frame.tab.h main.hpp
        $(CC) -c lex.yy.c
frame.tab.o:frame.tab.c main.hpp
        $(CC) -c frame.tab.c
main.o:main.hpp main.cpp
        $(CC) -c main.cpp
frame.tab.c frame.tab.h:frame.y
        $(YACC) -d frame.y
lex.yy.c:frame.l
        $(LEX) frame.l
```

```
clean:
	rm -f *.o *.c *.h
```

7. sample.cpp 文件

```
#include <iostream>
#include <string>
#include <ffmpeg/avformat.h>
#include <ffmpeg/avcodec.h>
#include <ffmpeg/avutils.h>
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"
#include "../hello.h"
#	include "space.h"
```

8. 运行结果

```
pandaxcl@PANDAXCL-F78E7D /d/work/lex_yacc/chapter06
$ make
flex frame.l
bison -d frame.y
g++ -c lex.yy.c
g++ -c frame.tab.c
g++ -c main.cpp
```

```
g++ lex.yy.o frame.tab.o main.o -o a.exe


pandaxcl@PANDAXCL-F78E7D /d/work/lex_yacc/chapter06
$ ./a.exe < sample.cpp
        采用尖括号的头文件：[iostream]
采用尖括号的头文件：[string]
采用尖括号的头文件：[ffmpeg/avformat.h]
采用尖括号的头文件：[ffmpeg/avcodec.h]
采用尖括号的头文件：[ffmpeg/avutils.h]
采用尖括号的头文件：[stdio.h]
采用尖括号的头文件：[stdlib.h]
采用双引号的头文件：[hello.h]
采用双引号的头文件：[../hello.h]
采用双引号的头文件：[space.h]


pandaxcl@PANDAXCL-F78E7D /d/work/lex_yacc/chapter06
$
```

9. 总结

总的来说，上面的解析C/C++包含预处理信息的时候需要了解如下的概念：

词法状态

> 所谓的词法状态就是指对文本进行词法分析的时候，词法分析器当前所处的状态，默认情况下，词法分析器都处于INITIAL状态，这个INITIAL状态 是Lex内置的状态。用户可以通过%x来重新定义各种各样的状态。

至 于为什么要使用状态，我们来看一个实际的例子：上面分析头文件的时候 采用了两个自定义的状态：_INCLUDE_状态和_INCLUDE_FILE_状态，_INCLUDE_状态是当遇到了#include 开始的，因为这个状态之后是尖括号或者 是双引号括起来的头文件名，在后面分析模板（使用尖括号）和分析字符串 （使用双引号）的时候也会遇到尖括号和双引号，因而需要区分这两种情况 ，所以才需要使用_INCLUDE_状态，以此来区分是包含文件还是模板或者是字 符串了。这一点非常重要!

同样，状态 _INCLUDE_FILE_存在也是为了区分双引号包含的头文件名称的， 因为双引号不同于尖括号，双引号在头文件名的开始和结束都是相同的，因 此为了区分头部和尾部的双引号，必须再增加一个状态。实际上这可以用来 简化词法分析器的编写，当您遇到这种类似的问题的时候可以考虑再增加一 种新的状态，通常来说就可以解决问题啦:)

不过还有一点特别需要强调的是当您感觉所添加的状态太多了，出现了混乱 现象，就说明用 Lex 状态已经不大适合处理这种问题了，就应该考虑采用 Yacc 的一条独立的语法规则来进行处理了:)这也是 Yacc 语法文件存在的原因，要 不然全部都可以采用词法分析文件来解决啦，还要语法分析文件干什么!

递归表达

这 里需要特别注意的是：frame.y 文件中 program 的构成采用了左递归的形 式。从代码中可以看出：program 可以是空（什么也没有）也可以是由现有 的 program 内容再追加一条 include_preprocess 类构成。当 program 内容为 空的时候增加一条 include_preprocess 类就表示 program 只有一条 include_preprocess 内容，当 program 已经有了一条 include_preprocess 内 容之后再增加一条 include_preprocess 内容就可以表示两条 include_preprocess 内容了，依次类推，可以表达无数的包含信息，从而 表达了无限的内容了。特别需要注意的是，这里的 program 表示的仅仅是现 有的内容，包括但不限于 include_preprocess 内容，还可以有其他的内容 ，这一点可以在增加其他内容的时候体现出来，因为 C/C++源代码不仅仅是 由包含信息构成的嘛:)

特别需要注意的是,这里要特表强调一下使用左递归,不是说右递归不行,而 是出于程序运行效率考虑最好使用左递归。具体原因在后续的文档中会有详细 的说明的:)

YYSTYPE, yylval, $$,$1,$2,...$n

因 为编写词法分析程序和语法分析程序的目的就是为了操作分析出来的数据 ，所以就需要有一种比较方便的形式来表达这些分析出来的数据。一种是词 法分析程序使用的方式，叫做 yylval；一种是语法分析程序使用的，叫做$n ，从上面的词法分析程序和语法分析程序中已经可以看到它们在相应的文件 中的使用了。

至于 YYSTYPE 那就更简单了，因为要表达词法分析程序和语法分析程序中的数据 ，既然是数据，在 C/C++中就有数据类型的概念，这里的 YYSTYPE 就是 yylval 和 $n 的数据类型。

特别需要注意的是，语法分析程序中每一个语法规则冒号左边的类的值用 $$表 示，而冒号右边的第一项用$1 表示，第二项用$2 表示，依次类推。

标记和值

标 记指的是由%token 定义的 INCLUDE 和 HEADERFILE，他们都对应着一个具体 值，而且具体值类型还有可能完全不一样。这里需要特别强调的一点是： 每一个标记都对应着一个值，你可以不使用，但是他就是确确实实存在着 ，而且从始至终都保持着这种对应。例如上面的 INCLUDE 标记的值就没有使 用，但是 HEADERFILE 标记的值就被使用了。在 Lex 和 Yacc 中标记都是用一个 C 宏定义的一个整数，而标记的值都是由 YYSTYPE 定义着的一个变量，这个 变量的名字就是 yylval，其中保存着相关的信息，这个信息就是在词法分 析文件中进行设置的，而在语法分析文件中就直接采用了。

实 际上%token 还可以更进一步的简化 Yacc 语法程序的编写，从而避免一些不 必要的错误。从上面的语法分析来看，对于不同的$n，还需要记住$n 的精确 类型和变量名，这一点其实是不必要的，可以通过%token <headerfile> HEADERFILE 来声明标记，那么在 Yacc 程序的语法规则 中就可以直接使用$3 来表示 yylval.headerfile 了，从而也就不需要记住那 些具体变量名啦:)

值得注意的是，尽管标记 是可以用%token 来定义，但是并不仅仅限于这种方 式，Yacc 中还可以用%type 来定义，采用%type 来定义的目的就是为那些不是 标记的类也准备一个对应的值的，例如：完全可以为 include_preprocess 定 义一个值，用来保存一些额外的信息，不过本文中并不需要，后续的文档中 就会需要这个功能了，在此先简单说明一下:)

词法动作

对于词法分析程序中的每一个正则表达式对应的规则，都有相应的 C/C++语句来做一些额外的处理，这个额外的处理就是词法动作。

语法动作

对 于语法分析程序中的每一个语法规则，都有相应的 C/C++语句来做一些额 外的处理，这个额外的处理就是语法动作。不过语法动作和词法动作的不同 之处在于，语法动作允许嵌入式的语法动作，而词法动作不行。至于什么是 嵌入式的语法动作，在后续的文档中会有详细的说明的!

好了，本章中还残留有一些问题故意没有解决，例如：包含文件的续行问题！留个 读者自己思考，可以在本文所讨论的基础上稍微改动一下就可以了。后续的文档正 在努力写出，敬请关注;）

5、开发 Lex 和 Yacc 程序的一般步骤

摘要

经过前面章节的准备，到目前为止一个完整的 C++应用框架已经完整的搭建起来了。现在的事情就是考虑如何利用这个框架来实现自己的目的功能程 序了。在这一章并不涉及到实际的开发而是先学习一下简单的理论知识。 本章将会根据我的个人开发经验来说明一下开发 Lex 和 Yacc 程序的一般开发 步骤，这里的内容也会随着后续的开发逐渐的完善起来，当在后续的开发 中遇到不明白的地方可以回到这一章来看一看，也许就明白了:)

1. 一般步骤

就我在开发 Lex 和 Yacc 程序的经验来说，如果要解析一个文件，那么必须经过 下面的步骤才能够写出正确的语法分析文件和词法分析文件：

1. 将被解析的文本进行分类。
2. 由分类构成 program。
3. 将类及子类分解为标记。
4. 完成了上面的所有步骤之后，我们就已经将被解析文本分解成功了，剩下的 事情就是为保存这些分解出来的信息而准备好一个树形数据结构，在这个时 候使用 C++的优势就显现出来了，我们可以借用 STL 里面的很多容器和算法的 。

2. 第一步：将被解析的文本分类

将被解析的文本进行分类，实际上是一件非常重要的事情，分类的好坏直接影响 到了后续的开发，分类分的好能够使得保存信息的数据结构简单明了和高效，但 是如果分得不好，可能导致冗余数据结构，大量得冗余信息以及需要全部修改代 码。

先举个实际的例子吧：C/C++源代码。怎样把 C/C++源代码中的语法元素用最少并 且能够保存全部信息的数据结构来表示，这本身就是一种挑战。在 C/C++的发展历 程中也经历过大大小小的变动，这说明要想对 C/C++源代码中的语法元素进行分类 是一件非常烦杂的事情，不过对于我们来说，这些语法元素已经存在了好多年了， 我们不需要从来开始，而是可以直接利用已经存在的知识来加快我们的开发。

C/C++的源代码中的每一个语法元素可以作为一个分类，例如：关键字、预处理宏 、函数、变量、语句、结构体、类、联合体等等。

但是，并不是说这里的开发是一帆风顺的，在本类文档的后续章节中也许会对这里 的类容进行修正，这只能说明我们对于 C/C++的语法构成还是存在误区而不是 C/C++ 语法本身有问题。对我们熟悉的问题尚且如此，那么对于我们不熟悉的问题更加如 此了，所以后续的开发中对前述代码的修正也就难免了，这就是重构，使得代码越 来越合理，越来越高效。

3. 第二步：由分类构成 program

对于上一步已经做好了的分类，最终的目的就是要用这些分类来成功的构成最终的 C/C++源代码。通过前面的章节的学习，我们知道这里的 program 表达的就是 C/C++ 源代码内容，当然你也可以用你自己喜欢的任何名字，例如“source”，这个就看 个人的喜好了，Lex 和 Yacc 对这个没有特别的要求。

这里我们采用 program 代表 C/C++源代码。上面的 C/C++分类：关键字、预处理宏、函数、变量、语句、结构体、类、联合体等等。在这一步里面就是考虑如何利用这些分类来构成完整的 C/C++源代码。这一步也充满了挑战，但是相对于第一步 来说就容易多了，因为有据可寻啊。我们可以遵循 Lex 和 Yacc 规范来一步一步的 将上面的分类组合起来构成 C/C++源代码。

常见的问题包括：

1. 如何用有限的分类来构成无限数量的文本（C/C++源代码）
2. 如何避免移进和规约冲突
3. 如何设置标记（tokens）

上面的概念暂时只需要初步的了解，所谓的标记就是能够直接表示文本文档（C/C++ 源代码）其中内容的概念，例如关键字类可以通过 int,long,char 等等直接表示，那 么 int,long,char 就是标记的值，标记就是代表这些值的一个标志而已，在 Lex 和 Yacc 中就是用 C 宏来表示标记的例如用 INT 宏表示 int,LONG 宏表示 long，CHAR 宏表示 char 等等，但是 INT 宏，LONG 宏和 CHAR 宏并不是直接定义为对应的 int，long，char 的，而仅仅只是一个整数；另外还需要强调的是这里的 INT 宏，LONG 宏和 CHAR 宏是自 己定义的，而不是 Lex 和 Yacc 内置的，因此可以随心所欲的定义，你完全可以用 INT 表示 char，LONG 表示 int，CHAR 表示 long 等，但是这样做并不好。关于这里的概念的 详细内容会在后续的开发中进行详细的解释。

4. 第三步：将类分解为标记

在对文本（C/C++源代码）进行了分类之后能否直接用标记表达出来呢？一部分分类 可以用标记直接表达出来了，还有一部分就不能或者不容易用标记表达出来。对于 这些不能或者不容易用标记表达出来的分类就还需要细分，这样最终的目的就是将 所有的类都能够用标记表达出来。

举个例子：在上面的C/C++源代码分类中就有“语句”类，那么语句其实还可以细分 为空语句、简单语句和复合语句三种，而且简单语句和符合语句也还要通过其他的 标记（标识符，运算符等等）进一步表达。

5. 第四步：保存信息

当所有的类以及子类都被标记表达出来之后，Lex 和 Yacc 程序也可以说完成了，但是 这仅仅只是在程序内部将文本分析完成了，对于我们人来说并没有什么实际的作用 ，我们最最希望的就是能够将这些分析出来的信息保存为另外一种方便阅读和理解 的方式。因此就需要自己另外设计数据结构来保存这些信息了，通常的情况 就是为每一个分类设计一个C++类，这样就可以将文本的内容以及结构信息完整的保 存下来啦。

通常的做法就是将这些内容和结构信息以简 单的文本形式直接输出，实际上真正的 应用还需要对这些信息进行认真的处理之后再输出。常见的应用有：语法高亮，流 程图自动生成，VC 中的类浏览，从C++源代码自动重新生成UML文档，从源程序的注 释自动生成程序文档（javadoc，doxygen）等等都是将分析出来的结构信息和内容 深入处理之后才输出的。呵呵，不过当您学会了 Lex 和 Yacc 之后，上面的这些应用 对于您来说也不是什么大不了的事情啦:）

6. 总结

从上面的讨论可以看出“分类”这一步是非常重要的步骤，占用的开发时间也是 非常多的，但是为了保证开发的正确性以及能够保质保量的完成任务，就需要认 真的重视这一步骤的重要性，多花些时间也是值得的。

好了，在这一章里面我讨论了开发 Lex 和 Yacc 的一般步骤，但是算不上特别详细，因为本类文章主要考虑的是一个应用问题，强调的是应用，对于那些特别理论的 东西我就在这里不多讲了，如果需要深入的了解可以参看编译原理相关书籍。

在后续的章节里面将会详细的分析C/C++源文档，采用的方法都是这里所陈述的 方法和步骤，如果在后续的章节中发现不太理解的地方可以参看这一章。

另外还需要格外说明的就是，因为我们分析的是C/C++源代码，所以这里的分类就 已经完成了，如果不太清楚可以参看C/C++语法说明。在后续的文档中将会按照问 题的需要来组织文档的结构了:） 敬请关注!

6、解析 C/C++包含文件

摘要

在这一章里面将要涉及到处理 C/C++的包含宏的解析。也就是说要从一大串 C/C++ 包含文件的声明中提取出文件名，以及相互依赖关系等等。实际上在这一章里面 使用的 Lex 和 Yacc 技术也是非常重要的，这些都会在本章中进行详细讲解。

我们知道对于 C/C++包含文件声明是为程序提供了一些库存的功能，因此存在一种依赖 关系，如果把这种依赖关系表达成为 Makefile 的形式，那么就可以自动生成 Makefile 。在这一章里面并不会实现自动生成 Makefile 的功能，而是仅仅解析出所有的包含文 件名，并记录下来。

1. 分析

我们知道 C/C++中存在两种形式的包含文件，一种是用“<>”包含的头文件，一种是“""”包含的头文件，这两种不同的形式表达了头文件的不同的搜索方式。另外还需要注意的是：这两种方式包含的都是磁盘上存在的文件名。也就是说，只 要是磁盘上存在的文件名都可以包含的，都是合法的，因而 C/C++里面存在的有扩展 名的头文件和没有扩展名的头文件都是合法的。并且还需要注意的是 C/C++包含的头 文件是可以续行的。

因而总结起来需要做到如下的几件事情：

1. 处理“<>”和“""”两种包含方式
2. 处理文件名
3. 处理续行

2. Lex 文件

```
%{

#include "main.hpp"// 在其中保存了记录头文件所需要的所有数据结构

#include "frame.tab.h"// 由 Yacc 自动生成的所有标记声明，实际上都是 C 宏

extern "C"{
```

```
int yywrap(void);
int yylex(void);
}
%}
%x _INCLUDE_
%x _INCLUDE_FILE_
%%
"#"[ \t]*"include"      {
                            BEGIN _INCLUDE_;// 进入_INCLUDE_状态
                            yylval.clear();// 需要将所有的
Include 值初始化
                            return INCLUDE;// 返回INCLUDE标记
                        }
<_INCLUDE_>[\"|<]       {
                            BEGIN _INCLUDE_FILE_;// 进入
_INCLUDE_FILE_状态
                            return *yytext; // 返回引号或者尖括号
                        }
<_INCLUDE_FILE_>[^\">]* {
                            yylval.headerfile+=yytext;// 记录头文
件字符串
                            return HEADERFILE;// 返回头文件标记
                        }
```

```
<_INCLUDE_FILE_>[\"|>]  {
                                BEGIN INITIAL;// 恢复到初始状态，默认状态
                                return *yytext;// 返回引号或者尖括号
                        }
[ \t\n]                 ;// 对于额外的空白都不处理直接扔掉
%%
int yywrap(void)
{
        return 1;// 只处理一个输入文件
}
```

3. Yacc 文件

```
%{
#include <iostream>
#include "main.hpp"
#define YYDEBUG 0 // 将这个变量设置为 1 则表示启动 Yacc 的调试功能
extern "C"{
void yyerror(const char *s);
extern int yylex(void);
}
std::vector<Include> g_Includes;// 用来记录所有的包含声明
Include             *g_pInclude;// 用来保存新增的包含声明信息的指针
```

```
%}

%token INCLUDE

%token HEADERFILE

%%

program:/* empty */

        | program include_preprocess // 用这种递归的方式从有限的标记表达出无限的内容

        ;

include_preprocess:

          INCLUDE '<'  HEADERFILE '>'

        {

                // 注意这里的$3，实际上就是上面的标记的第三个的意思

            // 因为 yylval 被声明为 Include 结构，参见 main.hpp 文件

            // 因而每个标记都是 Include 结构类型。

            g_Includes.push_back(Include());

            g_pInclude = &g_Includes.back();

            g_pInclude->clear();// 初始化

            g_pInclude->headerfile   = $3.headerfile;// 可以证明$3的类型就是Include类型

            g_pInclude->is_angle     = true;// 是尖括号

            g_pInclude->is_quotation = false;// 不是双引号

        }

        | INCLUDE '\"' HEADERFILE '\"'
```

```
        {
                // 值得说明的是：上面的 include_preprocess 用$表示，
            // 而不是用$0 表示。从左向右依次为：
            // include_preprocess        $
            // INCLUDE                   $1
            // '\"'                      $2
            // HEADERFILE                $3
            // '\"'                      $4
            g_Includes.push_back(Include());
            g_pInclude = &g_Includes.back();
            g_pInclude->clear();// 初始化
            g_pInclude->headerfile   = $3.headerfile;
            g_pInclude->is_angle     = false;// 不是尖括号
            g_pInclude->is_quotation = true;// 是双引号
        }
        ;
%%
void yyerror(const char *s)
{
        std::cerr<< s << std::endl;
}
int main()
```

```
{
#if YYDEBUG
        yydebug = 1;
#endif//YYDEBUG
        yyparse();// 进行语法分析，这个函数是 Yacc 自动生成的
        // 下面的这行代码仅仅使用了 STL 的输出算法到标准输出

std::copy(g_Includes.begin(),g_Includes.end(),std::ostream_iterator<I
nclude>(std::cout,"\n"));
        return 0;
}
```

4. main.hpp 文件

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
// 对于每一个项目最好都用一个独立的数据结构来保存相应的信息
struct Include
{
        void clear();// 设置Include的初始值
```

```
        std::string headerfile;// 记录头文件全名（包括路径）
        bool is_quotation;// 是否是双引号""括起来的头文件
        bool is_angle;// 是否是尖括号<>括起来的头文件
        // 下面的这个函数仅仅是用来输出到 C++流而准备的
        friend std::ostream&operator<<(std::ostream&s,const Include&I);
};
std::ostream&operator<<(std::ostream&s,const Include&I);
// 下面的这个宏定义用来取消 Lex 和 Yacc 默认的 YYSTYPE 定义，因为默认的 YYSTYPE 定义
// 仅仅只能够记录整数信息，因此要保存额外的信息必须这样定义宏，可以参见 Yacc
// 自动生成的标记头文件 frame.tab.h。
#define YYSTYPE Include
```

5. main.cpp 文件

```
#include "main.hpp"
// 初始化所有的 Include 信息，避免前后关联
void Include::clear()
{
        headerfile.clear();
        is_quotation = false;
        is_angle = false;
}
```

```
// 为了能够方便输出，在这里直接准备好了一个流输出函数
std::ostream&operator<<(std::ostream&s,const Include&I)
{
        if(I.is_angle)
                s << "采用尖括号的" ;
        if(I.is_quotation)
                s << "采用双引号的" ;
        s << "头文件：[" << I.headerfile << "]" ;
        return s;
}
```

6. Makefile 文件

```
LEX=flex
YACC=bison
CC=g++
a.exe:lex.yy.o frame.tab.o main.o
        $(CC) lex.yy.o frame.tab.o main.o -o a.exe
lex.yy.o:lex.yy.c frame.tab.h main.hpp
        $(CC) -c lex.yy.c
frame.tab.o:frame.tab.c main.hpp
        $(CC) -c frame.tab.c
main.o:main.hpp main.cpp
```

```
	$(CC) -c main.cpp
frame.tab.c frame.tab.h:frame.y
	$(YACC) -d frame.y
lex.yy.c:frame.l
	$(LEX) frame.l
clean:
	rm -f *.o *.c *.h
```

7. sample.cpp 文件

```
#include <iostream>
#include <string>
#include <ffmpeg/avformat.h>
#include <ffmpeg/avcodec.h>
#include <ffmpeg/avutils.h>
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"
#include "../hello.h"
#	include "space.h"
```

8. 运行结果

```
pandaxcl@PANDAXCL-F78E7D /d/work/lex_yacc/chapter06
$ make
```

```
flex frame.l

bison -d frame.y

g++ -c lex.yy.c

g++ -c frame.tab.c

g++ -c main.cpp

g++ lex.yy.o frame.tab.o main.o -o a.exe


pandaxcl@PANDAXCL-F78E7D /d/work/lex_yacc/chapter06

$ ./a.exe < sample.cpp

          采用尖括号的头文件：[iostream]

采用尖括号的头文件：[string]

采用尖括号的头文件：[ffmpeg/avformat.h]

采用尖括号的头文件：[ffmpeg/avcodec.h]

采用尖括号的头文件：[ffmpeg/avutils.h]

采用尖括号的头文件：[stdio.h]

采用尖括号的头文件：[stdlib.h]

采用双引号的头文件：[hello.h]

采用双引号的头文件：[../hello.h]

采用双引号的头文件：[space.h]


pandaxcl@PANDAXCL-F78E7D /d/work/lex_yacc/chapter06

$
```

9. 总结

总的来说，上面的解析 C/C++包含预处理信息的时候需要了解如下的概念：

词法状态

所谓的词法状态就是指对文本进行词法分析的时候，词法分析器当前所处的状态，默认情况下，词法分析器都处于 INITIAL 状态，这个 INITIAL 状态 是 Lex 内置的状态。用户可以通过%x 来重新定义各种各样的状态。

至 于为什么要使用状态，我们来看一个实际的例子：上面分析头文件的时候 采用了两个自定义的状态：_INCLUDE_状态和_INCLUDE_FILE_状态，_INCLUDE_状态是当遇到了#include 开始的，因为这个状态之后是尖括号或者 是双引号括起来的头文件名，在后面分析模板（使用尖括号）和分析字符串 （使用双引号）的时候也会遇到尖括号和双引号，因而需要区分这两种情况 ，所以才需要使用_INCLUDE_状态，以此来区分是包含文件还是模板或者是字 符串了。这一点非常重要!

同样，状态 _INCLUDE_FILE_存在也是为了区分双引号包含的头文件名称的， 因为双引号不同于尖括号，双引号在头文件名的开始和结束都是相同的，因 此为了区分头部和尾部的双引号，必须再增加一个状态。实际上这可以用来 简化词法分析器的编写，当您遇到这种类似的问题的时候可以考虑再增加一 种新的状态，通常来说就可以解决问题啦:)

不过还有一点特别需要强调的是当您感觉所添加的状态太多了，出现了混乱 现象，就说明用 Lex 状态已经不大适合处理这种问题了，就应该考虑采用 Yacc 的一条独立的语法规则来进行处理了:)这也是 Yacc 语法文件存在的原因，要 不然全部都可以采用词法分析文件来解决啦，还要语法分析文件干什么!

递归表达

这 里需要特别注意的是：frame.y 文件中 program 的构成采用了左递归的形 式。从代码中可以看出：program 可以是空（什么也没有）也可以是由现有 的 program 内容再追加一条 include_preprocess 类构成。当 program 内容为 空的时候增加一条 include_preprocess 类就表示 program 只有一条 include_preprocess 内容，当 program 已经有了一条 include_preprocess 内 容之后再增加一条 include_preprocess 内容就可以表示两条 include_preprocess 内容了，依次类推，可以表达无数的包含信息，从而 表达了无限的内容了。特别需要注意的是，这里的 program 表示的仅仅是现 有的内容，包括但不限于 include_preprocess 内容，还可以有其他的内容 ，这一点可以在增加其他内容的时候体现出来，因为 C/C++源代码不仅仅是 由包含信息构成的嘛:)

特别需要注意的是,这里要特表强调一下使用左递归,不是说右递归不行,而 是出于程序运行效率考虑最好使用左递归。具体原因在后续的文档中会有详细 的说明的:)

YYSTYPE, yylval, $$,$1,$2,...$n

因 为编写词法分析程序和语法分析程序的目的就是为了操作分析出来的数据 ，所以就需要有一种比较方便的形式来表达这些分析出来的数据。一种是词 法分析程序使用的方式，叫做 yylval；一种是语法分析程序使用的，叫做$n ，从上面的词法分析程序和语法分析程序中已经可以看到它们在相应的文件 中的使用了。

至于 YYSTYPE 那就更简单了,因为要表达词法分析程序和语法分析程序中的数据 ，既然是数据，在 C/C++中就有数据类型的概念，这里的 YYSTYPE 就是 yylval 和 $n 的数据类型。

特别需要注意的是,语法分析程序中每一个语法规则冒号左边的类的值用$$表 示，而冒号右边的第一项用$1 表示，第二项用$2 表示，依次类推。

标记和值

标 记指的是由%token 定义的 INCLUDE 和 HEADERFILE,他们都对应着一个具体 值，而且具体值类型还有可能完全不一样。这里需要特别强调的一点是： 每一个标记都对应着一个值，你可以不使用，但是他就是确确实实存在着 ，而且从始至终都保持着这种对应。例如上面的 INCLUDE 标记的值就没有使 用,但是 HEADERFILE 标记的值就被使用了。在 Lex 和 Yacc 中标记都是用一个 C 宏定义的一个整数，而标记的值都是由 YYSTYPE 定义着的一个变量,这个 变量的名字就是 yylval,其中保存着相关的信息,这个信息就是在词法分 析文件中进行设置的，而在语法分析文件中就直接采用了。

实 际上%token 还可以更进一步的简化 Yacc 语法程序的编写，从而避免一些不 必要的错误。从上面的语法分析来看，对于不同的$n，还需要记住$n 的精确 类型和变量名，这一点其实是不必要的，可以通过%token <headerfile> HEADERFILE 来声明标记，那么在 Yacc 程序的语法规则 中就可以直接使用$3 来表示 yylval.headerfile 了，从而也就不需要记住那 些具体变量名啦:)

值得注意的是，尽管标记 是可以用%token 来定义，但是并不仅仅限于这种方 式，Yacc 中还可以用%type 来定义，采用%type 来定义的目的就是为那些不是 标记的类也准备一个对应的值的，例如：完全可以为 include_preprocess 定 义一个值，用来保存一些额外的信息，不过本文中并不需要,后续的文档中 就会需要这个功能了,在此先简单说明一下:)

词法动作

对于词法分析程序中的每一个正则表达式对应的规则，都有相应的 C/C++语句来做一些额外的处理，这个额外的处理就是词法动作。

语法动作

对 于语法分析程序中的每一个语法规则，都有相应的 C/C++语句来做一些额 外的处理，这个额外的处理就是语法动作。不过语法动作和词法动作的不同 之处在于，语法动作允许嵌入式的语法动作，而词法动作不行。至于什么是 嵌入式的语法动作，在后续的文档中会有详细的说明的!

好了，本章中还残留有一些问题故意没有解决，例如：包含文件的续行问题！留个 读者自己思考，可以在本文所讨论的基础上稍微改动一下就可以了。后续的文档正 在努力写出，敬请关注;)

7、筛选信息(容错处理)

```
#if 0
```

在通常的情况下，我们只关心文本中的一部分信息，但是为了编写词法和语法分析程
序，又不得不将所有的结构信息全部描写出来，例如：我们仅仅关心 C++源文档中的类名字
信息，而不关心类是否有成员变量，是否有成员函数以及是否有其它的一些 C++内容。将结
构信息全部描述出来的做法是费时费力的，通常的情况往往导致项目的不可完成或者延期
完成。另外，作为程序设计者和代码编写者，都希望将功能局域化而不扩散难度，也非常
希望编写的代码能够简单的不予理睬还没有理解的内容，专心处理自己关心的内容。本篇
文档就以着重考虑处理 C/C++类名称信息为例，忽略其它的一切没有进行语法描述的 C/C++
信息。这就是 Lex 和 Yacc 的错误(error)处理的一个应用:)我非常喜欢:)

下面给出词法和语法分析器的源代码，因为这么简单的程序，看源代码是学习的最好
方法:)

```
#endif

//////////////////////////////////////////////////////////////////////
// 词法扫描器文件:lex.l
%{
#include <string>
```

```
// 将 yylval 的值类型由默认的 int 修改为 std::string 类型，实际上可以修改为你认为的任
// 何类型，仅仅只是需要定一个这样 YYSTYPE 宏即可，特别注意，这个宏定义必须在后面
// 的标记文件 yacc.tab.h 之前定义，并且在 yacc 文件中也要有这个 YYSTYPE 定义，并且必
// 须和这里的保持一致。实际上 YYSTYPE 的定义在生成的标记文件 yacc.tab.h 中有一个宏
// 判断，如果用户也就是我们定义了 YYSTYPE 宏，那么就用我们定义的 YYSTYPE，否则就用
// 默认的 YYSTYPE，也就是 int 类型:)

#define YYSTYPE std::string
#include "yacc.tab.h"

#define LEX_RETURN(arg) yylval=yytext;return arg
%}
d    [0-9]
l    [a-z]
u    [A-Z]
a    {l}|{u}
%%
[;{}]                   {LEX_RETURN(yytext[0]);}
"class"                 {LEX_RETURN(CLASS);}

(_|{a})(_|{a}|{d})*     {LEX_RETURN(IDENTIFIER);}

[ \t\n]                 /* 忽略空白 */
.                       /* 忽略其它一切没有被处理的文本 */
%%
int yywrap()
{
    return 1;
}
////////////////////////////////////////////////////////////////////////
///////////


////////////////////////////////////////////////////////////////////////
///////////
// 语法分析器文件:yacc.y
%{
#include <iostream>
```

```
#define YYSTYPE std::string
extern int yylex();
void yyerror(const char*msg);
%}
%token CLASS IDENTIFIER
%%
program:/* 空 */
        | program class //处理C++ 类
        | program error ';' // 一旦出现错误直接跳到最近的分号处，回复正常的扫描过程
                            // 特别注意这里的标记符号 error，它是由 yacc 自动生成的标记
                            // 和上面的 CLASS 和 IDENTIFIER 标记一样都可以直接应用到语法
                            // 描述中
        ;
class:// 特别注意一下下面的class语法描述又调用了program，这是一种嵌套结构的常见做法
        CLASS IDENTIFIER '{' program '}' ';' {std::cout<<"发现类名:"<<$2<<std::endl;}
      ;
%%
void yyerror(const char*msg)
{
    // 错误处理，仅仅是简单的输出一个错误标记，在具体应用中应当能够分析出这种错
    // 误是否已经被处理了，这里为了说明上面的错误信息过滤没有进行这种识别
    std::cerr<< "发现错误" << std::endl;
}
int main()
{
    yyparse();
    return 0;
}
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Makefile 文件
CC=g++
CFLAGS=
LEX=flex
YACC=bison
YACCFLAGS=-d
```

```
TARGET=lexyacc

$(TARGET):lex.yy.c yacc.tab.h yacc.tab.c
    $(CC) $(CFLAGS) lex.yy.c yacc.tab.c -o $(TARGET)
lex.yy.c:lex.l
    $(LEX) lex.l
yacc.tab.c yacc.tab.h:yacc.y
    $(YACC) $(YACCFLAGS) yacc.y

clean:
    rm -f lex.yy.c yacc.tab.h yacc.tab.c
//////////////////////////////////////////////////////////////////////
//////////

// 从上面的代码中可以看出，通过容错处理之后，我们就可以专心于特定的功能代码编写
// 而不需要考虑其它的信息，这样就可以极大的降低解决问题的难度。在后续的文档中都
// 会采用这种技巧来实现特定的功能。如果对上面的一些描述还不是很清晰的话，可以参
// 见我之前已经写出来的系列文档，在本章中值得说明的只有两点：
// 1：yacc 自动生成的 error 标记的使用
// 2：改变默认的 yylval 的 int 类型为 std::string 类型
// 其实我是在尽可能的使用 C++库，目的当然是降低编写代码的难度，减少代码，便于说
// 明问题;)
//
// 好了，本篇文档到此就已经说明了本文开始所提出的问题:D，后续的文档正在努力给出
// 。其实编写 Lex 和 Yacc 程序非常简单，只需要注意几个常见错误就可以完成一般的任务
// 了，在下一篇里面将会讲解常见的错误及其处理方法:)敬请关注:)

// 下面是实例应用
//////////////////////////////////////////////////////////////////////
//////////
// 测试文件：sample.cpp
class Point
{
    int x;
    int y;
    int GetX();
    int GetY();
};
```

```
class Rect
{
    int x;
    int y;
    int w;
    int h;
    int GetX();
    int GetY();
    int GetW();
    int GetH();
};

class Wrapper
{
    class Inner1{};
    class Inner2{
        class InnerInner1{float f;};
        class InnerInner2{};
        std::string name;
    };
    bool sex;
};
//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////
// 编译并运行过程
D:\home\blog\lexyacc>make
flex lex.l
bison -d yacc.y
g++ lex.yy.c yacc.tab.c -o lexyacc

D:\home\blog\lexyacc>lexyacc.exe < sample.cpp
发现错误
发现类名:Point
发现错误
发现类名:Rect
发现类名:Inner1
发现错误
发现类名:InnerInner1
发现类名:InnerInner2
发现错误
发现类名:Inner2
发现错误
```

发现类名:Wrapper

D:\home\blog\lexyacc>

//

目　录

Lex 和 Yacc 简明教程

作者 :Thomas Niemann

序言

本书将教会你如何使用 lex 和 yacc 构造一个编译器。lex 和 yacc 是两个用来生成词汇分析器和剖析器的工具。我假设你能够运用 C 语言编程，并且理解数据结构的含义，例如“链表”和“树”。

导言部分描写了构建编译器所需的基本部分，以及 lex 和 yacc 之间的互动关系。后面两章更加详细的描写了 lex 和 yacc。以此为背景，我们构建了一个经典的计算器程序。这个计算器支持常用的算术符号和控制结构，例如实现了像 if-else 和 while 这样的控制结构。经过小小的修改，我们就把这个计算器转换成一个可以运行在基于栈的计算机上的编译器。后面的间节讨论了在编写编译器是经常发生的问题。本书中使用的例程的源代码可以从下面列出的网站上下载到。

THOMAS NIEMANN

波特兰，俄勒冈州

网站：epaperpress.com

导言

在 1975 年之前，编写编译器一直是一个非常费时间的工作。这一年 Lesk[1975] 和 Johson [1975] 发表了关于 lex 和 yacc 的论文。这些工具极大地简化了编写编译器的工作。在 Aho[1986] 中描写了关于如何具体实现 lex 和 yacc 的细节。从下列地方可以等到 lex 和 yacc 工具：

- Mortice Kern System (MKS)，在 www.mks.com，
- GNU flex 和 bison，在 www.gnu.org，
- Ming，在 www.mingw.org，
- Cygwin，在 www.cygwin.org，还有
- 我的 Lex 和 Yacc 版本，在 epaperpress.com

MKS 的版本是一个高质量的商业产品，零售价大约是 300 美元。GNU 软件是免费的。flex 的输出数据也可以被商业版本所用，对应的商业版本号是 1.24，bison 也一样。Ming 和 Cygwin 是 GNU 软件在某些 32 位 Windows 系统上的移植产物。事实上 Cygwin 是 UNIX 操作系统在 Windows 上的一个模拟接口，其中包括了 gcc 和 g++ 编译器。

我的版本是基于 Ming 的，但量是使用 Visual C++ 编译的并且包含一个改善文件操作程序的小补丁。如果你下载我的版本，请记住解压缩的时候一定要保留文件夹结构。

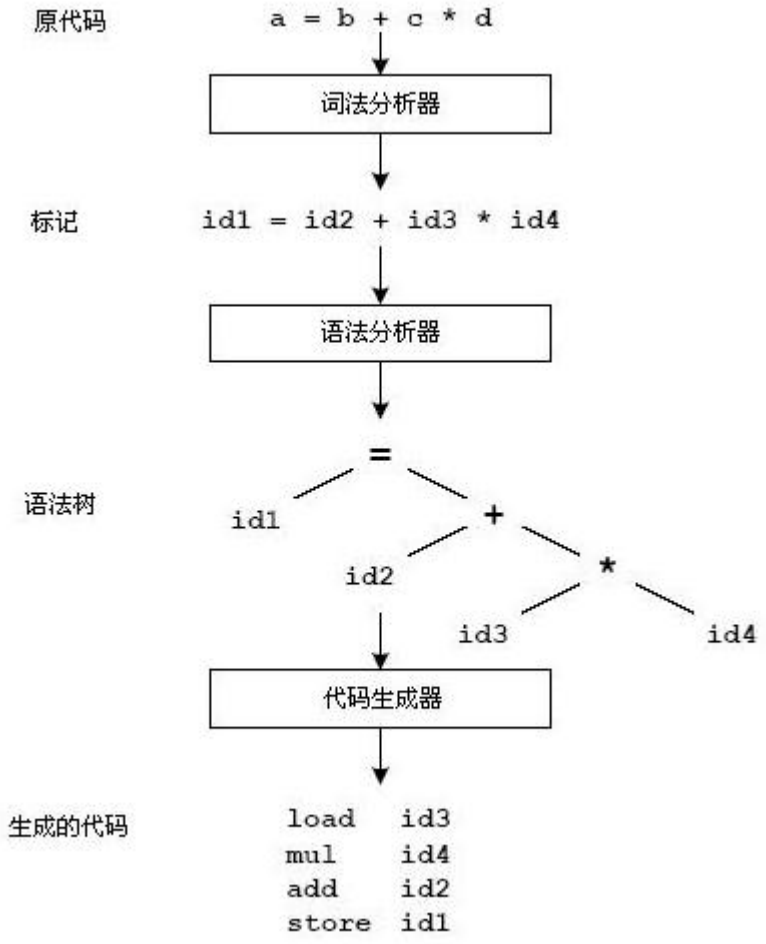


图 1：编译顺序

Lex 为词法分析器或扫描器生成 C 程序代码。它使正则表达式匹配输入的字符串并且把它们转

换成对应的标记。标记通常是代表字符串或简单过程的数值。图 1 说明了这一点。

当 Lex 发现了输入流中的特定标记，就会把它们输入一个特定的符号表中。这个符号表也会包含其它的信息，例如数据类型（整数或实数）和变量在内存中的位置。所有标记的实例都代表符号表中的一个适当的索引值。

Yacc 为语法分析器或剖析器生成 C 程序代码。Yacc 使用特定的语法规则以便解释从 Lex 得到的标记并且生成一棵语法树。语法树把各种标记当作分级结构。例如，操作符的优先级和相互关系在语法树中是很明显的。下一步，生成编译器原代码，对语法树进行一次第一深度历遍以便生成原代码。有一些编译器直接生成机器码，更多的，例如上图所示，输出汇编程序。

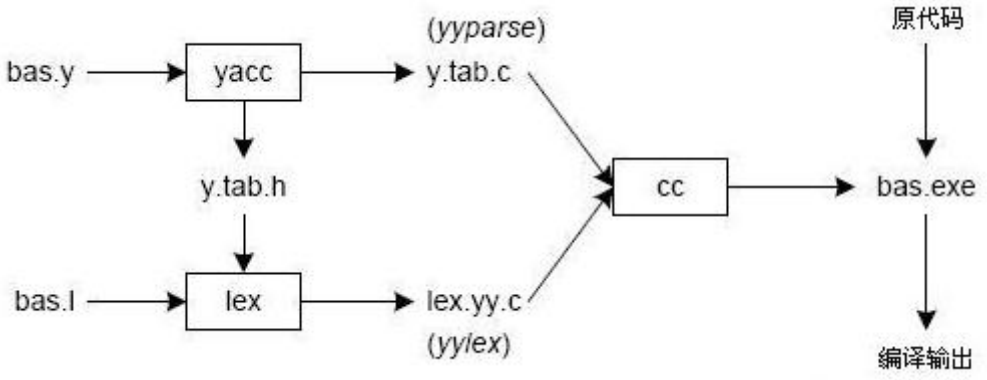


图 2：用 Lex/Yacc 构建一个编译器

图 2 显示了 Lex 和 Yacc 使用的命名约定。我们首先要说明我们的目标是编写一个 BASIC 编译器。首先我们要指定 Lex 的所有的模式匹配规则（bas.l）和 Yacc 的全部语法规则（bas.y）。下面列举了产生我们的编译器，bas.exe，的命令。

```
yacc -d bas.y                  # 生成 y.tab.h, y.tab.c
lex bas.l                      # 生成 lex.yy.c
cc lex.yy.c y.tab.c -o bas.exe     # 编译／连接
```

Yacc 读入 bas.y 中的语法描述而后生成一个剖析器，即 y.tab.c 中的函数 yyparse。bas.y 中包含的是一系列的标记声明。“-d”选项使 Yacc 生成标记声明并且把它们保存在 y.tab.c 中。Lex 读入 bas.l 中的正则表达式的说明，包含文件 y.tab.h，然后生成词汇解释器，即文件 lex.yy.c 中的函数 yylex。

最终，这个解释器和剖析器被连接到一起，而组成一个可执行程序，bas.exe。我们从 main 函数中调用 yyparse 来运行这个编译器。函数 yyparse 自动调用 yylex 以便获取每一个标志。

Lex

理论

在第一阶段，编译器读入源代码然后把字符串转换成对应的标记。使用正则表达式，我们可以为 Lex 设计特定的表达式以便从输入代码中扫描和匹配字符串。在 Lex 上每一个字符串对应一个动作。通常一个动作返回一个代表被匹配的字符串的标记给后面的剖析器使用。作为学习的开始，在此我们不返回标记而是简单的打印被匹配的字符串。我们要使用下面这个正则表达式扫描标识符。

letter(letter|digit)*

这个问题表达式所匹配的字符串以一个简单字符开头，后面跟随着零个或更多个字符或数字。这个例子很好的显示了正则表达式中所允许的操作。

重复，用“*”表示

交替，用“|”表示

串联

每一个正则表达式代表一个有限状态自动机 (FSA)。我们可以用状态和状态之间的转换来代表一个 FSA。其中包括一个开始状态以及一个或多个结束状态或接受状态。

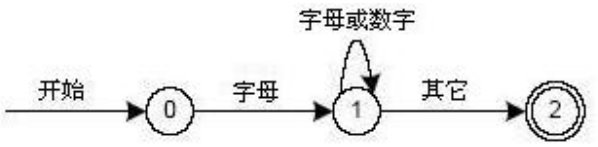


图 3：有限状态自动机

在图 3 中状态 0 是开始状态，而状态 2 是接受状态。当读入字符时，我们就进行状态转换。当读入第一个字母时，程序转换到状态 1。如果后面读入的也是字母或数字，程序就继续保持在状态 1。如果读入的字符不是字母或数字，程序就转换到状态 2，即接受状态。每一个 FSA 都表现为一个计算机程序。例如，我们这个 3 状态机器是比较容易实现的：

```
start: goto state0
state0: read c
        if  c = letter goto state1
    goto state0
state1: read c
    if  c = letter goto state1
    if  c = digit goto state1
    goto state2
state2: accept string
```

这就是 lex 所使用的技术。lex 把正则表达式翻译成模拟 FSA 的一个计算机程序。通过搜索计算机生成的状态表，很容易使用下一个输入字符和当前状态来判定下一个状态。

现在我们应该可以容易理解 lex 的一些使用限制了。例如 lex 不能用于识别像括号这样的外壳结构。识别外壳结构需要使用一个混合堆栈。当我们遇到“（”时，就把它压入栈中。当遇到“）”

时，我们就在栈的顶部匹配它，并且弹出栈。然而，lex 只有状态和状态转换能力。由于它没有堆栈，它不适合用于剖析外壳结构。yacc 给 FSA 增加了一个堆栈，并且能够轻易处理像括号这样的结构。重要的是对特定工作要使用合适的工具。lex 善长于模式匹配。yacc 适用于更具挑战性的工作。

练习

表意字符	匹配字符
.	除换行外的所有字符
\n	换行
*	0 次或无限次重复前面的表达式
+	1 次或更多次重复前面的表达式
?	0 次或 1 次出现前面的表达式
^	行的开始
$	行的结尾
a|b	a 或者 b
(ab)+	1 次或玩多次重复 ab
"a+b"	字符串 a+b 本身（C 中的特殊字符仍然有效）
[]	字符类

表 1：简单模式匹配

表达式	匹配字符
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	a, b, c 中的一个
[a-z]	从 a 到 z 中的任意字符
[a\-z]	a, -, z 中的一个
[-az]	-, a, z 中的一个
[A-Za-z0-9]+	一个或更多个字母或数字
[\t\n]+	空白区
[^ab]	除 a,b 外的任意字符
[a^b]	a, ^, b 中的一个
[a|b]	a, |, b 中的一个
a|b	a, b 中的一个

表 2：模式匹配举例

lex 中的正则表达式由表意字符（表 1）组成。表 2 中列举了模式匹配的例子。在方括号（[]）中，通常的操作失去了本来含意。方括号中只保留两个操作，连字号（“-”）和抑扬号（“^”）。当把连字号用于两个字符中间时，表示字符的范围。当把抑扬号用在开始位置时，表示对后面的表达式取反。如果两个范式匹配相同的字符串，就会使用匹配长度最长的范式。如果两者匹配的长度相同，就会选用第一个列出的范式。

```
... 定义 ...
%%
... 规则 ...
%%
... 子程序 ...
```

lex 的输入文件分成三个段，段间用 %% 来分隔。上例很好的表明了这个意思。第一个例子是最短的可用 lex 文件：

```
%%
```

输入字符将被一个字符一个字符直接输出。由于必须存在一个规则段，第一个 %% 总是要求存在的。然而，如果我们不指定任何规则，默认动作就是匹配任意字符然后直接输出到输出文件。默认的输入文件和输出文件分别是 stdin 和 stdout。下面是效果完全相同的例子，显式表达了默认代码：

```
%%
 /* 匹配除换行外的任意字符 */
. ECHO;
 /* 匹配换行符 */
\n ECHO;
%%
int yywrap(void) {
    return 1;
}
int main(void) {
    yylex();
    return 0;
}
```

上面规则段中指定了两个范式。每一个范式必须从第一列开始。紧跟后面的必须是空白区（空格，TAB 或换行），以及对应的任意动作。动作可以是单行的 C 代码，也可以是括在花括号中的多行 C 代码。任何不是从第一列开始的字符串都会被逐字拷贝进所生成的 C 文件中。我们可以利用这个特殊行为在我们的 lex 文件中增加注释。在上例中有“.”和“\n”两个范式，对应的动作都是 ECHO。lex 预先字义了一些宏和变量。ECHO 就是一个用于直接输出范式所匹配的字符的宏。这也是对任何未匹配字符的默认动作。通常 ECHO 是这样定义的：

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

变量 yytext 是指向所匹配的字符串的指针（以 NULL 结尾），而 yyleng 是这个字符串的长度。变量

yyout 是输出文件，默认状态下是 stdout。当 lex 读完输入文件之后就会调用函数 yywrap。如果返回 1 表示程序的工作已经完成了，否则返回 0。每一个 C 程序都要求一个 main 函数。在本例中我们只是简单地调用 yylex，lex 扫描器的入口。有些 lex 实现的库中包含了 main 和 yywrap。这就是为什么我们的第一个例子，最短的 lex 程序，能够正确运行。

名称	功能
int yylex(void)	调用扫描器，返回标记
char *yytext	指针，指向所匹配的字符串
yyleng	所匹配的字符串的长度
yylval	与标记相对应的值
int yywrap(void)	约束，如果返回 1 表示扫描完成后程序就结束了，否则返回 0
FILE *yyout	输出文件
FILE *yyin	输入文件
INITIAL	初始化开始环境
BEGIN	按条件转换开始环境
ECHO	输出所匹配的字符串

表 3：lex 中预定义变量

下面是一个根本什么都不干的程序。所有输入字符都被匹配了，但是所有范式都没有定义对应的动作，所以没有任何输出。

```
%%
.
\n
```

下面的例子在文件的每一行前面插入行号。有些 lex 实现预先定义和计算了 yylineno 变量。输入文件是 yyin，默认指向 stdin。

```
%{
    int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

定义段由替代式，C 代码，和开始状态构成。定义段中的 C 代码被简单地原样复制到生成的 C 文件的顶部，而且必须用 %{ 和 %} 括起来。替代式简化了正则表达式规则。例如，我们可以定义数字

和字母：

```
digit [0-9]
letter [A-Za-z]
%{
    int count;
%}
%%
/* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

范式和对应的表达式必须用空白区分隔开。在规则段中，替代式要用花括号括起来（如 {letter}）以便和其字面意思区分开来。每当匹配到规则段中的一个范式，与之相对应的 C 代码就会被运行。

下面是一个扫描器，用于计算一个文件中的字符数，单词数和行数（类似 Unix 中的 wc 程序）：

```
%{
int nchar, nword, nline;
%}
%%
\n { nline++; nchar++; }
[^ \t\n]+ { nword++, nchar += yyleng; }
. { nchar++; }
%%
int main(void) {
yylex();
printf("%d\t%d\t%d\n", nchar, nword, nline);
return 0;
}
```

YACC

理论

yacc 的文法由一个使用 BNF 文法（Backus-Naur form）的变量描述。BNF 文法规则最初由 John Backus 和 Peter Naur 发明，并且用于描述 Algol60 语言。BNF 能够用于表达上下文无关语言。现代程序语言中的大多数结构可以用 BNF 文法来表达。例如，数值相乘和相加的文法是：

E -> E + E

E -> E * E

E -> id

上面举了三个例子，代表三条规则（依次为 r1，r2，r3）。像 E（表达式）这样出现在左边的结构叫非终结符（nonterminal）。像 id（标识符）这样的结构叫终结符（terminal，由 lex 返回的标记），它们只出现在右边。这段文法表示，一个表达式可以是两个表达式的和、乘积，或者是一个标识符。我们可以用这种文法来构造下面的表达式：

E -> E * E (r2)

-> E * z (r3)

-> E + E * z (r1)

-> E + y * z (r3)

-> x + y * z (r3)

每一步我们都扩展了一个语法结构，用对应的右式替换了左式。右面的数字表示应用了哪条规则。为了剖析一个表达式，我们实际上需要进行倒序操作。不是从一个简单的非终结符开始，根据语法生成一个表达式，而是把一个表达式逐步简化成一个非终结符。这叫做“自底向上”或者“移进 - 归约”分析法，这需要一个堆栈来保存信息。下面就是用相反的顺序细述了和上例相同的语法：

1 . x + y * z 移进

2 x . + y * z 归约 (r3)

3 E . + y * z 移进

4 E + . y * z 移进

5 E + y . * z 归约 (r3)

6 E + E . * z 移进

7 E + E * . z 移进

8 E + E * z . 归约 (r3)

9 E + E * E . 归约 (r2) 进行乘法运算

10 E + E . 归约 (r1) 进行加法运算

11 E . 接受

点左面的结构在堆栈中，而点右面的是剩余的输入信息。我们以把标记移入堆栈开始。当堆栈顶部和右式要求的记号匹配时，我们就用左式取代所匹配的标记。概念上，匹配右式的标记被弹出堆栈，而左式被压入堆栈。我们把所匹配的标记认为是一个句柄，而我们所做的就是把句柄向左式归约。这个过程一直持续到把所有输入都压入堆栈中，而最终堆栈中只剩下最初的非终结符。在第 1

步中我们把 x 压入堆栈中。第 2 步对堆栈应用规则 r3，把 x 转换成 E。然后继续压入和归约，直到堆栈中只剩下一个单独的非终结符，开始符号。在第 9 步中，我们应用规则 r2，执行乘法指令。同样，在第 10 步中执行加法指令。这种情况下，乘法就比加法拥有了更高的优先级。

考虑一下，如果我们在第 6 步时不是继续压入，而是马上应用规则 r1 进行归约。这将导致加法比乘法拥有更高的优先级。这叫做“移进 - 归约”冲突（shift-reduce conflict）。我们的语法模糊不清，对一个表达式可以引用一条以上的适用规则。在这种情况下，操作符优先级就可以起作用了。

举另一个例子，可以想像在这样的规则中

E -> E + E

是模糊不清的，因为我们既可以从左面又可以人右面递归。为了挽救这个危机，我们可以重写语法规则，或者给 yacc 提供指示以明确操作符的优先顺序。后面的方法比较简单，我们将在练习段中进行示范。

下面的语法存在“归约 - 归约”冲突 (reduce-reduce conflict)。当堆栈中存在 id 是，我们既可以归约为 T，也可以归约为 E。

E -> T

E -> id

T -> id

当存在冲突时，yacc 将执行默认动作。当存在“移进 - 归约”冲突时，yacc 将进行移进。当存在“归约 - 归约”冲突时，yacc 将执行列出的第一条规则。对于任何冲突，它都会显示警告信息。只有通过书写明确的语法规则，才能消灭警告信息。后面的章节中我们将会介绍一些消除模糊性的方法。

练习，第一部分

... 定义 ...

%%

... 规则 ...

%%

... 子程序 ...

yacc 的输入文件分成三段。“定义”段由一组标记声明和括在“%{”和“%}”之间的 C 代码组成。BNF 语法定义放在“规则”段中，而用户子程序添加在“子程序”段中。

构造一个小型的加减法计算器可以最好的说明这个意思。我们要以检验 lex 和 yacc 之间的联系开始我们的学习。下面是 yacc 输入文件的定义段：

```
%token INTEGER
```

上面的定义声明了一个 INTEGER 标记。当我们运行 yacc 时，它会在 y.tab.c 中生成一个剖析器，同时会产生一个包含文件 y.tab.h：

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
```

```
#define INTEGER 258
extern YYSTYPE yylval;
```

lex 文件要包含这个头文件，并且使用其中对标记值的定义。为了获得标记，yacc 会调用 yylex。yylex 的返回值类型是整型，可以用于返回标记。而在变量 yylval 中保存着与返回的标记相对应的值。例如，

```
[0-9]+    {    yylval = atoi(yytext);
                return INTEGER;    }
```

将把整数的值保存在 yylval 中，同时向 yacc 返回标记 INTEGER。yylval 的类型由 YYSTYPE 决定。由于它的默认类型是整型，所以在这个例子中程序运行正常。0-255 之间的标记值约定为字符值。例如，如果你有这样一条规则

```
[-+]        return *yytext;                /* 返回操作符 */
```

减号和加号的字符值将会被返回。注意我们必须把减号放在第一位心避免出现范围指定错误。由于 lex 还保留了像“文件结束”和“错误过程”这样的标记值，生成的标记值通常从 258 左右开始。下面是为我们的计算器设计的完整的 lex 输入文件：

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
[0-9]+    {
                yylval = atoi(yytext);
                return INTEGER;
          }
[-+\n]    return *yytext;
[ \t]     ;       /* skip whitespace */
.         yyerror("invalid character");
%%
int yywrap(void) {
    return 1;
}
```

yacc 在内部维护着两个堆栈；一个分析栈和一个内容栈。分析栈中保存着终结符和非终结符，并且代表当前剖析状态。内容栈是一个 YYSTYPE 元素的数组，对于分析栈中的每一个元素都保存着一个对应的值。例如，当 yylex 返回一个 INTEGER 标记时，yacc 把这个标记移入分析栈。同时，相应的 yylval 值将会被移入内容栈中。分析栈和内容栈的内容总是同步的，因此从栈中找到对应于一个标记的值是很容易实现的。下面是为我的计算器设计的 yacc 输入文件：

```
%{
```

```
int yylex(void);
void yyerror(char *);
%}
%token INTEGER
%%
program:
        program expr '\n'                { printf("%d\n", $2); }
       |
       ;
expr:
        INTEGER                          { $$ = $1; }
       | expr '+' expr                   { $$ = $1 + $3; }
       | expr '-' expr                   { $$ = $1 - $3; }
       ;
%%
void yyerror(char *s) {
      fprintf(stderr, "%s\n", s);
      return 0;
}
int main(void) {
     yyparse();
     return 0;
}
```

规则段的方法类似前面讨论过的 BNF 文法。规则第一条叫 command 规则。其中的左式，或都称为非终结符，从最左而开始，后面紧跟着一个自己的克隆。后面跟着的是右式。与规则相应的动作写在后面的花括号中。

通过利用左递归，我们已经指定一个程序由 0 个或更多个表达式构成。每一个表达式由换行结束。当探测到换行符时，程序就会打印出表达式的结果。当程序应用下面这个规则时

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

在分析栈中我们其实用左式替代了右式。在本例中，我们弹出“expr '+' expr”然后压入“expr”。我们通过弹出三个成员，压入一个成员缩小的堆栈。在我们的 C 代码中可以用通过相对地址访问内容栈中的值，“$1”代表右式中的第一个成员，“$2”代表第二个，后面的以此类推。“$$”表示缩小后的堆栈的顶部。在上面的动作中，把对应两个表达式的值相加，弹出内容栈中的三个成员，然后把造得到的和压入堆栈中。这样，分析栈和内容栈中的内容依然是同步的。

当我们把 INTEGER 归约到 expr 时，数字值开始被输入内容栈中。当 NTEGER 被移分析栈中之后，我们会就应用这条规则

```
expr: INTEGER                        { $$ = $1; }
```

INTEGER 标记被弹出分析栈，然后压入一个 expr。对于内容栈，我们弹出整数值，然后又把它压回去。也可以说，我们什么都没做。事实上，这就是默认动作，不需要专门指定。当遇到换行符时，与 expr 相对应的值就会被打印出来。当遇到语法错误时，yacc 会调用用户提供的 yyerror 函数。如果你需要修改对 yyerror 的调用界面，改变 yacc 包含的外壳文件以适应你的需求。你的 yacc 文件中的最后的函数是 main ... 万一你奇怪它在哪里的话。这个例子仍旧有二义性的语法。yacc 会显示“移进 - 归约”警告，但是依然能够用默认的移进操作处理语法。

练习，第二部分

在本段中我们要扩展前一段中的计算器以便加入一些新功能。新特性包括算术操作乘法和除法。圆括号可以用于改变操作的优先顺序，并且可以在外部定义单字符变量的值。下面举例说明了输入量和计算器的输出：

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5)
user: y = 5
user: x
calc: 27
user: y
calc: 5
user: x + 2*y
calc: 37
```

词汇解释器将返回**VARIABLE**和**INTEGER**标志。对于变量，yylval 指定一个到我们的符号表**sym**中的索引。对于这个程序，**sym**仅仅保存对应变量的值。当返回**INTEGER**标志时，yylval 保存扫描到的数值。这里是 lex 输入文件：

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
%}
%%
    /* variables */
[a-z]   {
            yylval = *yytext - 'a';
            return VARIABLE;
        }
    /* integers */
[0-9]+  {
```

```
            yylval = atoi(yytext);
            return INTEGER;
        }
        /* operators */
[-+()=/*\n]     { return *yytext; }
        /* skip whitespace */
[ \t]           ;
        /* anything else is an error */
.       yyerror("invalid character");
%%
int yywrap(void) {
        return 1;
}
```

接下来是 yacc 的输入文件。yacc 利用 **INTEGER**和 **VARIABLE**的标记在 **y.tab.h**中生成 **#defines** 以便在 lex 中使用。这跟在算术操作符定义之后。我们可以指定**%left**，表示左结合，或者用**%right** 表示右结合。最后列出的定义拥有最高的优先权。因此乘法和除法拥有比加法和减法更高的优先权。所有这四个算术符都是左结合的。运用这个简单的技术，我们可以消除文法的歧义。

```
%token      INTEGER VARIABLE
%left       '+' '-'
%left       '*' '/'
%{
        void yyerror(char *);
        int yylex(void);
        int sym[26];
%}
%%
program:
            program statement '\n'
            |
            ;
statement:
            expr { printf("%d\n", $1); }
            | VARIABLE '=' expr { sym[$1] = $3; }
            ;
expr:
            INTEGER
            | VARIABLE { $$ = sym[$1]; }
```

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

%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main(void) {
    yyparse();
    return 0;
}
```

计算器

描述

这个版本的计算器的复杂度大大超过了前一个版本。主要改变包括像**if-else** 和**while**这样的控制结构。另外，在剖析过程中还构造了一个语法树。剖析完成之后，我们历遍语法树来生成输出。此处提供了两个版本的树历遍程序：

- 一个在历遍树的过程中执行树中的声明的解释器，以及
- 一个为基于堆栈的计算机生成代码的编译器。

为了使我们的解释更加形象，这里有一个例程，

```
x = 0;
while (x < 3) {
    print x;
    x = x + 1;
}
```

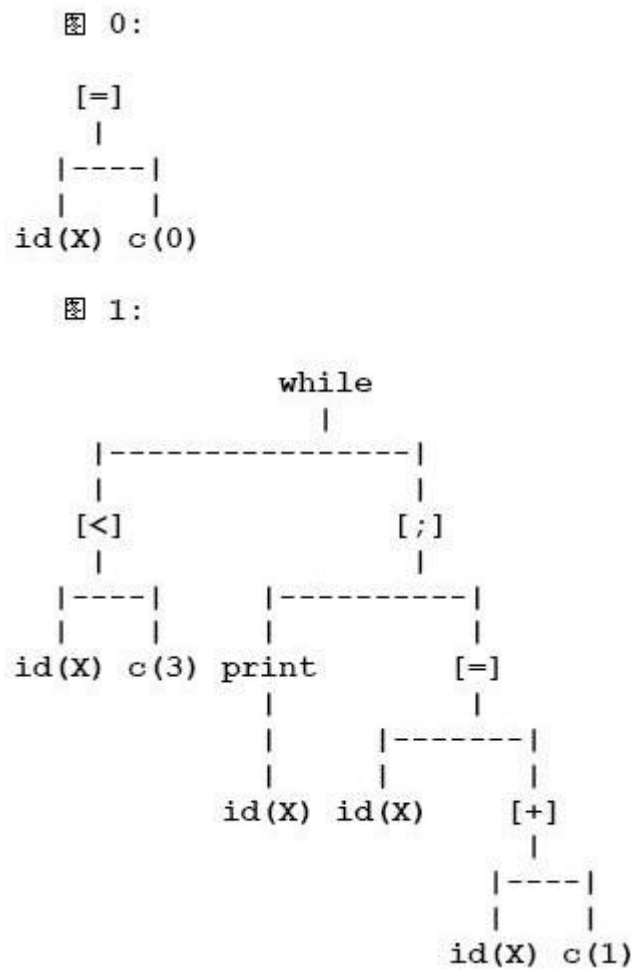
以及解释版本的输出数据，

```
0
1
2
```

和编译版本的输出数据，和

```
    push 0
    pop x
L000:
    push x
    push 3
    compLT
    jz L001
    push x
    print
    push x
    push 1
    add
    pop x
    jmp L000
L001:
```

一个生成语法树的版本。



包含文件中包括了对语法树和符号表的定义。符号表 sym 允许使用单个字符表示变量名。语法树中的每个节点保存一个常量（conNodeType），标识符（idNodeType），或者一个带算子（oprNodeType）的内部节点。所有这三种变量压缩在一个 union 结构中，而节点的具体类型跟据其内部所拥有的结构来判断。

lex 输入文件中包含有返回 VARIABLE 和 INTEGER 标志的正则表达式。另外，也定义了像 EQ 和 NE 这样的双字符算子的标志。对于单字符算子，只需简单地返回其本身。

yacc 输入文件中定义了 YYSTYPE，yylval 的类型，定义如下

```
%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
};
```

这将导致在 y.tab.h 中生成如下代码：

```
typedef union {
    int iValue; /* integer value */
```

```
        char sIndex; /* symbol table index */
        nodeType *nPtr; /* node pointer */
    } YYSTYPE;
    extern YYSTYPE yylval;
```

在剖析器的内容栈中，常量、变量和节点都可以由 yylval 表示。注意下面的定义

```
    %token <iValue> INTEGER
    %type <nPtr> expr
```

这把 expr 和 INTEGER 分别绑定到 union 结构 YYSTYPE 中的 nPtr 和 iValue 成员。这是必须的，只有这样 yacc 才能生成正确的代码。例如，这个规则

```
    expr: INTEGER { $$ = con($1); }
```

可以生成下面的代码。注意，yyvsp[0] 表示内容栈的顶部，或者表示对应于 INTEGER 的值。

```
    yylval.nPtr = con(yyvsp[0].iValue);
```

一元算子的优先级比二元算子要高，如下所示

```
    %left GE LE EQ NE '>' '<'
    %left '+' '-'
    %left '*' '/'
    %nonassoc UMINUS
```

%nonassoc 意味着没有依赖关系。它经常在连接词中和 %prec 一起使用，用于指定一个规则的优先级。因此，我们可以这样

```
    expr: '-' expr %prec UMINUS { $$ = node(UMINUS, 1, $2); }
```

表示这条规则的优先级和标志 UMINUS 相同。而且，如同上面所定义的，UMINUS 的优先级比其它所有算子都高。类似的技术也用于消除 if-else 结构中的二义性（请看 if-else 二义性）。

语法树是从底向上构造的，当变量和整数减少时才分配叶节点。当遇到算子时，就需要分配一个节点，并且把上一个分配的节点作为操作数记录在其中。

构造完语法树之后，调用函数 ex 对此语法树进行第一深度历遍。第一深度历遍按照原先节点分配的顺序访问各节点。

这将导致各算子按照剖析期间的访问顺序被使用。此处含有三个版本的 ex 函数：一个解释版本，一个编译版本，一个用于生成语法树的版本。

包含文件

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;
/* constants */
typedef struct {
    int value; /* value of constant */
} conNodeType;
/* identifiers */
typedef struct {
    int i; /* subscript to sym array */
} idNodeType;
/* operators */
typedef struct {
    int oper; /* operator */
    int nops; /* number of operands */
    struct nodeTypeTag *op[1]; /* operands (expandable) */
} oprNodeType;
typedef struct nodeTypeTag {
    nodeEnum type; /* type of node */
    /* union must be last entry in nodeType */
    /* because operNodeType may dynamically increase */
    union {
        conNodeType con; /* constants */
        idNodeType id; /* identifiers */
        oprNodeType opr; /* operators */
    };
} nodeType;
extern int sym[26];
```

Lex 输入文件

```
%{
#include <stdlib.h>
#include "calc3.h"
#include "y.tab.h"
void yyerror(char *);
%}
%%
[a-z] {
        yylval.sIndex = *yytext - 'a';
        return VARIABLE;
        }
[0-9]+ {
        yylval.iValue = atoi(yytext);
        return INTEGER;
        }
[-()<>=+*/;{}.] {
        return *yytext;
        }
">="    return GE;
"<="    return LE;
"=="    return EQ;
"!="    return NE;
"while"    return WHILE;
"if"    return IF;
"else"    return ELSE;
"print"    return PRINT;
[ \t\n]+    ;          /* ignore whitespace */
.             yyerror("Unknown character");
%%
int yywrap(void) {
        return 1;
}
```

Yacc 输入文件

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"
/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(char *s);
int sym[26]; /* symbol table */
%}
%union {
    int iValue; /* integer value */
    char sIndex; /* symbol table index */
    nodeType *nPtr; /* node pointer */
    };
%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <nPtr> stmt expr stmt_list
%%
program:
        function                { exit(0); }
        ;
function:
```

```
        function stmt               { ex($2); freeNode($2); }
        | /* NULL */
        ;
stmt:
        ';'                         { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'                  { $$ = $1; }
        | PRINT expr ';'            { $$ = opr(PRINT, 1, $2); }
        | VARIABLE '=' expr ';'     { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt   { $$ = opr(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX    { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt    { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}' { $$ = $2; }
        ;
stmt_list:
        stmt                        { $$ = $1; }
        | stmt_list stmt            { $$ = opr(';', 2, $1, $2); }
        ;
expr:
        INTEGER                     { $$ = con($1); }
        | VARIABLE                  { $$ = id($1); }
        | '-' expr %prec UMINUS     { $$ = opr(UMINUS, 1, $2); }
        | expr '+' expr             { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr             { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr             { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr             { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr             { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr             { $$ = opr('>', 2, $1, $3); }
        | expr GE expr              { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr              { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr              { $$ = opr(NE, 2, $1, $3); }
        | expr EQ expr              { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'              { $$ = $2; }
        ;
%%
#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)
nodeType *con(int value) {
    nodeType *p;
```

```
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeCon;
    p->con.value = value;
    return p;
}
nodeType *id(int i) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeId;
    p->id.i = i;
    return p;
}
nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
    (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
```

```
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}
void freeNode(nodeType *p) {
    int i;
    if (!p) return;
    if (p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}
void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}
int main(void) {
    yyparse();
    return 0;
}
```

解释器

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"
int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
        case typeCon: return p->con.value;
        case typeId: return sym[p->id.i];
        case typeOpr:
            switch(p->opr.oper) {
                case WHILE: while(ex(p->opr.op[0]))   ex(p->opr.op[1]);
                            return 0;
                case IF:  if (ex(p->opr.op[0]))  ex(p->opr.op[1]);
                            else if (p->opr.nops > 2)  ex(p->opr.op[2]);
                            return 0;
                case PRINT: printf("%d\n", ex(p->opr.op[0]));
                            return 0;
                case ';': ex(p->opr.op[0]);
                            return ex(p->opr.op[1]);
                case '=': return sym[p->opr.op[0]->id.i] = ex(p->opr.op[1]);
                case UMINUS: return -ex(p->opr.op[0]);
                case '+': return ex(p->opr.op[0]) + ex(p->opr.op[1]);
                case '-': return ex(p->opr.op[0]) - ex(p->opr.op[1]);
                case '*': return ex(p->opr.op[0]) * ex(p->opr.op[1]);
                case '/': return ex(p->opr.op[0]) / ex(p->opr.op[1]);
                case '<': return ex(p->opr.op[0]) < ex(p->opr.op[1]);
                case '>': return ex(p->opr.op[0]) > ex(p->opr.op[1]);
                case GE: return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
                case LE: return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
                case NE: return ex(p->opr.op[0]) != ex(p->opr.op[1]);
                case EQ: return ex(p->opr.op[0]) == ex(p->opr.op[1]);
            }
    }
    return 0;
}
```

编译器

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"
static int lbl;
int ex(nodeType *p) {
    int lbl1, lbl2;
    if (!p) return 0;
    switch(p->type) {
        case typeCon: printf("\tpush\t%d\n", p->con.value);
            break;
        case typeId:  printf("\tpush\t%c\n", p->id.i + 'a');
            break;
        case typeOpr:
         switch(p->opr.oper) {
            case WHILE:  printf("L%03d:\n", lbl1 = lbl++);
                ex(p->opr.op[0]);
                printf("\tjz\tL%03d\n", lbl2 = lbl++);
                ex(p->opr.op[1]);
                printf("\tjmp\tL%03d\n", lbl1);
                printf("L%03d:\n", lbl2);
                break;
            case IF:  ex(p->opr.op[0]);
            if (p->opr.nops > 2) {
                /* if else */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex(p->opr.op[1]);
                printf("\tjmp\tL%03d\n", lbl2 = lbl++);
                printf("L%03d:\n", lbl1);
                ex(p->opr.op[2]);
                printf("L%03d:\n", lbl2);
            } else {
                /* if */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex(p->opr.op[1]);
                printf("L%03d:\n", lbl1);
```

```
            }
            break;
        case PRINT:  ex(p->opr.op[0]);
            printf("\tprint\n");
            break;
        case '=':  ex(p->opr.op[1]);
            printf("\tpop\t%c\n", p->opr.op[0]->id.i + 'a');
            break;
        case UMINUS:  ex(p->opr.op[0]);
            printf("\tneg\n");
            break;
        default:  ex(p->opr.op[0]);
            ex(p->opr.op[1]);
            switch(p->opr.oper) {
                case '+': printf("\tadd\n"); break;
                case '-': printf("\tsub\n"); break;
                case '*': printf("\tmul\n"); break;
                case '/': printf("\tdiv\n"); break;
                case '<': printf("\tcompLT\n"); break;
                case '>': printf("\tcompGT\n"); break;
                case GE: printf("\tcompGE\n"); break;
                case LE: printf("\tcompLE\n"); break;
                case NE: printf("\tcompNE\n"); break;
                case EQ: printf("\tcompEQ\n"); break;
            }
        }
    }
    return 0;
}
```

图

```
/* source code courtesy of Frank Thomas Braun */
#include <stdio.h>
#include <string.h>
#include "calc3.h"
#include "y.tab.h"
int del = 1; /* distance of graph columns */
int eps = 3; /* distance of graph lines */
/* interface for drawing (can be replaced by "real" graphic using GD or
other) */
void graphInit (void);
void graphFinish();
void graphBox (char *s, int *w, int *h);
void graphDrawBox (char *s, int c, int l);
void graphDrawArrow (int c1, int l1, int c2, int l2);
/* recursive drawing of the syntax tree */
void exNode (nodeType *p, int c, int l, int *ce, int *cm);
/*****************************************************************/
/* main entry point of the manipulation of the syntax tree */
int ex (nodeType *p) {
        int rte, rtm;
        graphInit ();
        exNode (p, 0, 0, &rte, &rtm);
        graphFinish();
        return 0;
}
/*c----cm---ce----> drawing of leaf-nodes
l leaf-info
*/
/*c----------------cm--------------ce----> drawing of non-leaf-nodes
l node-info
* |
* ------------ ...----
* | | |
* v v v
* child1 child2 ... child-n
```

```
* che che che
*cs cs cs cs
*
*/
void exNode
( nodeType *p,
int c, int l, /* start column and line of node */
int *ce, int *cm /* resulting end column and mid of node */
)
{
    int w, h; /* node width and height */
    char *s; /* node text */
    int cbar; /* "real" start column of node (centred above
                subnodes) */
    int k; /* child number */
    int che, chm; /* end column and mid of children */
    int cs; /* start column of children */
    char word[20]; /* extended node text */
    if (!p) return;
    strcpy (word, "???"); /* should never appear */
    s = word;
    switch(p->type) {
        case typeCon: sprintf (word, "c(%d)", p->con.value); break;
        case typeId: sprintf (word, "id(%c)", p->id.i + 'A'); break;
        case typeOpr:
        switch(p->opr.oper){
            case WHILE: s = "while"; break;
            case IF: s = "if"; break;
            case PRINT: s = "print"; break;
            case ';': s = "[;]"; break;
            case '=': s = "[=]"; break;
            case UMINUS: s = "[_]"; break;
            case '+': s = "[+]"; break;
            case '-': s = "[-]"; break;
            case '*': s = "[*]"; break;
            case '/': s = "[/]"; break;
            case '<': s = "[<]"; break;
```

```
            case '>': s = "[>]"; break;
            case GE: s = "[>=]"; break;
            case LE: s = "[<=]"; break;
            case NE: s = "[!=]"; break;
            case EQ: s = "[==]"; break;
        }
        break;
}
/* construct node text box */
graphBox (s, &w, &h);
cbar = c;
*ce = c + w;
*cm = c + w / 2;
/* node is leaf */
if (p->type == typeCon || p->type == typeId || p->opr.nops == 0) {
        graphDrawBox (s, cbar, l);
        return;
}
/* node has children */
cs = c;
for (k = 0; k < p->opr.nops; k++) {
        exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
        cs = che;
}
/* total node width */
if (w < che - c) {
        cbar += (che - c - w) / 2;
        *ce = che;
        *cm = (c + che) / 2;
}
/* draw node */
graphDrawBox (s, cbar, l);
/* draw arrows (not optimal: children are drawn a second time) */
cs = c;
for (k = 0; k < p->opr.nops; k++) {
        exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
        graphDrawArrow (*cm, l+h, chm, l+h+eps-1);
```

```
            cs = che;
        }
}
/* interface for drawing */
#define lmax 200
#define cmax 200
char graph[lmax][cmax]; /* array for ASCII-Graphic */
int graphNumber = 0;
void graphTest (int l, int c)
{   int ok;
    ok = 1;
    if (l < 0) ok = 0;
        if (l >= lmax) ok = 0;
    if (c < 0) ok = 0;
    if (c >= cmax) ok = 0;
    if (ok) return;
    printf ("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ...%d, %d",l, c, lmax, cmax);
    exit (1);
}
void graphInit (void) {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = 0; j < cmax; j++) {
            graph[i][j] = ' ';
        }
    }
}
void graphFinish() {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = cmax-1; j > 0 && graph[i][j] == ' '; j--);
        graph[i][cmax-1] = 0;
        if (j < cmax-1) graph[i][j+1] = 0;
        if (graph[i][j] == ' ') graph[i][j] = 0;
    }
    for (i = lmax-1; i > 0 && graph[i][0] == 0; i--);
    printf ("\n\nGraph %d:\n", graphNumber++);
```

```
        for (j = 0; j <= i; j++) printf ("\n%s", graph[j]);
        printf("\n");
}
void graphBox (char *s, int *w, int *h) {
        *w = strlen (s) + del;
        *h = 1;
}
void graphDrawBox (char *s, int c, int l) {
        int i;
        graphTest (l, c+strlen(s)-1+del);
        for (i = 0; i < strlen (s); i++) {
                graph[l][c+i+del] = s[i];
        }
}
void graphDrawArrow (int c1, int l1, int c2, int l2) {
        int m;
        graphTest (l1, c1);
        graphTest (l2, c2);
        m = (l1 + l2) / 2;
        while (l1 != m) {
                graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
        }
        while (c1 != c2) {
                graph[l1][c1] = '-'; if (c1 < c2) c1++; else c1--;
        }
        while (l1 != l2) {
                graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
        }
        graph[l1][c1] = '|';
}
```

Lex 进阶

字符串

LEX 和 YACC 的使用方法

一词法分析器LEX 的使用

1．1 Lex概述

程序设计语言从机器语言发展到今天的象pascal, C等这样的高级语言，使人们可以摆脱与机器有关的细节进行程序设计。但是用高级语言写程序时程序员必须在程序中详尽地告诉计算机系统怎样去解决某个问题，这在某种程度上说也是一件很复杂的工作。

人们希望有新的语言——非常高级的语言，用这种语言程序员仅仅需要告诉计算机系统要解决什么问题，计算机系统能自动地从这个问题的描述去寻求解决问题的途径，或者说将这个问题的描述自动地转换成用某种高级语言如C、FORTRAN表示的程序。这个程序就可以解决给定的问题，这种希望虽然还没有能够完全变成现实，但是在某些具体的问题领域里已经部分地实现了。

这里要介绍的Lex和下章要介绍的Yacc就是在编译程序设计这个领域里的两种非常高级的语言。用它们可以很方便的描述词法分析器和语法分析器，并自动产生出相应的高级语言（C）的程序。

Lex是一个词法分析器（扫描器）的自动产生系统，它的示意图如图 1.1。

Lex源程序是用一种面向问题的语言写成的。这个语言的核心是正规表达式（正规式），用它描述输入串的词法结构。在这个语言中用户还可以描述当某一个词形被识别出来时要完成的动作，例如在高级语言的词法分析器中，当识别出一个关键字时，它应该向语法分析器返回该关键字的内部编码。Lex并不是一个完整的语言，它只是某种高级语言（称为lex的宿主语言）的扩充，因此lex没有为描述动作设计新的语言，而是借助其宿主语言来描述动作。我们只介绍C作为lex的宿主语言时的使用方法，在Unix系统中，FORTRAN语言的一种改进形式Ratfor也可以做lex的宿主语言。

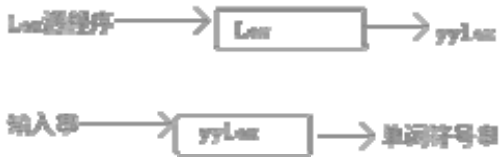


图 1.1 Lex 示意图

Lex自动地表示把输入串词法结构的正规式及相应的动作转换成一个宿主语言的程序，即词法分析程序，它有一个固定的名字yyler，在这里yyler是一个C语言的程序。

Yylex将识别出输入串中的词形，并且在识别出某词形时完成指定的动作。

看一个简单的例子：写一个lex源程序，将输入串中的小写字母转换成相应的大定字母。

程序如下：

```
%%
  [a-z]printf("%c".yytext[0]+'A'-'a');
```

上述程序中的第一行%%是一个分界符，表示识别规则的开始。第二行就是识别规则。左边是识别小写字母的正规式。右边就是识别出小写字母时采取的动作:将小写字母转换成相应的大写字母。

Lex的工作原理是将源程序中的正规式转换成相应的确定有限自动机，而相应的动作则插入到yylox中适当的地方，控制流由该确定有限自动机的解释器掌握，不同的源程序，这个解释器是相同的。关于lex工作原理的详细情况请参考[3]，这里不多介绍。

1．2 lex源程序的格式

lex源程序的一般格式是：

{辅助定义的部分}

%%

{识别规则部分}

%%

{用户子程序部分}

其中用花括号起来的各部分都不是必须有的。当没有“用户子程序部分”时，第二个%%也可以省去。第一个%%是必须的，因为它标志着识别规则部分的开始，最短的合法的lex源程序是：

%%

它的作用是将输入串照原样抄到输出文件中。

识别规则都分是Lex源程序的核心。它是一张表，左边一列是正规式，右边一列是相应的动作。下面是一条典型的识别规则：

integer printf（"found keywcrd INT"）;

这条规则的意思是在输入串中寻找词形“integer”，每当与之匹配成功时，就打印出“foundkeyword INT”这句话。

注意在识别规则中，正规式与动作之间必须用空格分隔开。动作部分如果只是一个简单的C表达式，则可以写在正规式右边同一行中，如果动作需要占两行以上，则须用花括号括起来，否则会出错。上倒也可以写成：

integer {printf（"found keyword INT"）;}

下面先介绍识别规则部分的写法，再介绍其余部分。

1.3 Lex用的正规式

一个正规式表示一个字符串的集合。正规式由正文字符与正规式运算符组成．正文字符组成基本的正规式，表示某一个符号串；

正规式运算符则将基本的正规式组合成为复杂的正规式，表示字符串的集合。

例如：

ab

仅表示字符串ab，而

（a b）+

表示字符串的集合：

｛ab，abab，ababab，…）。

Lex中的正规式运算符有下列十六种：

”\[ ]∧ -？ •*+| （）/${} %<>

上述运算符需要作为正文字符出现在正规式中时，必须借助于双引号”或反斜线＼，具体用法是；

xyz“++”或xyz＼+＼+

表示字符串xyz++

为避免死记上述十多个运算符，建议在使用非数字或字母字符时都用双引号或反斜线。

要表示双引号本身可用＼”，要表示反外线用”＼”或\\

前面说过，在识别规则中空格表示正规式的结束，因此要在正规式中引进空格必须借助双引号或反斜线，但出现在方括号[]之内的空格是例外。

几个特殊符号：

＼n是回车换行（newline）

＼t是tab

\b是退格（back space）

下面按照运算符的功能分别介绍上述正规式运算符。

1．字符的集合

用方括号对可以表示字符的集合。正规式

[a b c]

与单个字符a或b或c匹配

在方括号中大多数的运算符都不起作用，只有＼-和∧例外。

运算符----表示字符的范围，例如

［a-z 0-9 ＜＞-]

表示由所有小写字母，所有数字、尖括号及下划线组成的字符集合。

如果某字符集合中包括-在内，则必须把它写在第一个或最后一个位置上，如

[-+0-9]

与所有数字和正负号匹配

在字符集合中，运算符∧必须写在第一个位置即紧接在左方括号之后，它的作用是求方括号中除∧之外的字符组成的字符集合相对于计算机的字符集的补集，例如 [∧abc]与除去a、b和c以外的任何符号匹配。

运算符＼在方括号中同样发挥解除运算符作用的功能。

2．与任意字符匹配的正规式

运算符。形成的正规式与除回车换行符以外的任意字符匹配。

在lex的正规式中，也可以用八进制数字与＼一起表示字符，如

[＼40-＼176]

与ASCII字符集中所有在八进制 40（空格）到八进制 176（～）之间的可打印字符匹配。

3．可有可无的表达式

运算得？指出正规式中可有可无的子式，例如

ab?c

与ac或abc匹配，即b是可有可无的。

4．闭包运算

运算符*和十是 Lex正规式中的闭包运算符，它们表示正规式中某子式的重复，例如"a*"表示由 0 个或多个a组成的字符串的集合，而"a+"表示由 1 个或多个a组成的字符串的集合，下面两个正规式是常用的：

[a-z]+

[A-Za-z][A-Za-z 0-9]*

第一个是所有由小写字母组成的字符串的集合，第二个是由字母开头的字母数字串组成的集合。

5、选择和字符组

运算符|表示选择：

（ab|cd）
与ab或cd匹配
运算符（）表示一组字符，注意（）与[ ]的区别。（ab）表示字符串ab，而[ab]则表示单个字符a或b。
圆括号（）用于表示复杂的正规式，例如：
（ab|cd+）？（ef）*
与abefef, efef, cdef, cddd匹配，但不与abc, abcd或abcdef匹配。
6、上下文相关性
lex可以识别一定范围的上下文，因此可在一定程度上表示上下文相关性。

若某正规式的第一个字符是∧，则仅当该正规出现在一行的开始处时才被匹配，一行的开始处是指整个输入串的开始或者紧接在一个回车换行之后，注意∧还有另一个作作即求补，∧的这两种用法不可能发生矛盾。

若某正规式的最后一个字符是$，则仅当该表达式出现在一行的结尾处时才被匹配，一行的结尾处是指该表达式之后紧接一个回车换行。

运算符/指出某正规式是否被匹配取决于它的后文，例如: ab/cd，仅在ab之后紧接cd的情况下才与ab匹配。$其实是/的一个特殊情形，例如下面两个正规式等价：ab$,ab/\ n

某正规式是否被匹配，或者匹配后执行什么样的动作也可能取决于该表达式的前文，前文相关性的处理方法在后面专门讨论，将用到运算符"<>"
7、重复和辅助定义
当被{}括起来的是数字对时，{}表示重复；当它括起来的是一个名字时，则表示辅助定义的展开。例如：a{1，5} ，表示集合{a.aa.aaa.aaaa.aaaaa}.{digit}则与预先定义的名叫dight的串匹配，并将有定义插入到它在正规式中出现的位置上，辅助定义在后面专门讨论。

最后，符号%的作用是作为lex源程序的段间分隔符。

1．4 Lex源程序中的动作

前面说过当Lex识别出一个词形时，要完成相应的动作。这一节叙述Lex为描述动作提供的帮助。

首先应指出，输入串中那些不与任何识别规则中的正规式匹配的字符串将被原样照望抄到输出文件中去。因此如果用户不仅仅是希望照抄输出，就必须为每一个可能的词形提供识别规则，并在其中提供相应的动作。用lex为工具写程序语言的词法分析器时尤其要注意。最简单的一种动作是滤掉输入中的某些字符串，这种动作用C的空语句“；”来实现。
例：滤掉输入申中所有空格、tab和回车换行符，相应的识别规则如下：
［\t\n]；

如果相邻的几条规则的动作都相同，则可以用|表示动作部分，它指出该规则的动作与下一条规则的动作相同。例如上倒也可以写成：
“ ”|
“\t”|
“\n”；
注意\t和\n中的双引号可以去掉。

外部字符数组yytext的内容是当前被某规则匹配的字符串，例如正规式[a-z]+与所有由小写字母组成的字符串匹配，要想知道与它匹配的具体字符串是什么，可用下述规则：
[a－z]＋ printf（“％ s”， yytext）；
动作printf（“%s”，yytext）就是将字符数组yytext的内容打印出来，这个动作用得很频繁，Lex

提供了一个宏ECHO来表示它，因此上述识别规则可以写成：
［a－z]+ECHO；
请注意，上面说过缺省的动作就是将输入串原样抄到输出文件中，那么上述规则起什么作用呢？这一点将在“规则的二义性”一节中解释。

有时有必要知道被匹配的字符串中的字符个数，外部变量yyleng就表示当前yytext中字符的个数。例如要对输入串中单词的个数和字符的个数进行计数（单词假定是由大写或小写字母组成的字符串），可用下述规则：

```
[a-zA－Z]+ {words++；
Chars+=yyleng；}
```

注意被匹配的字符串的第一个字符和最后一个字符分别是
yytext[0]和yytext[yyleng-1]

下面介绍三个Lex提供的在写动作时可能用到的C函数

I．yymore（）

当需下一次被匹配的字符串被添加在当前识别出的字符串后面，即不使下一次的输入替换yytext中已有的内容而是接在它的内容之后，必须在当前的动作中调用yymore( )

例：假设一个语言规定它的字符串括在两个双引号之间，如果某字符串中含有双引号，则在它前面加上反斜线＼。用一个正规式来表达该字符串的定义很不容易，不如用下面较简明的正规式与yymore（）配合来识别：

```
\" [∧"]*{
if（yytext[yyleng－1]
= ='\\'yymore( );
else
…normal user processing
}
```

当输入串为”abc＼"def”时，上述规则首先与前五个字符”abc＼匹配，然后调用yymore（ ）使余下部分”def被添加在前一部分之后，注意作为字符串结尾标志的那个双引号由”normal user proessing”部分负责处理

2．yyless（n）

如果当前匹配的字符串的末尾部分需要重新处理，那么可以调用 yyless（n）将这部分子串“退回”给输入串，下次再匹配处理。yyless(n）中的n是不退回的字符个数，即退回的字符个数是yyleng-n。

例；在C语言中串“=-a”具有二义性，假定要把它解释为“=-a”同时给出信息，可用下面的识别规则：

```
=-［a-zA-Z]{
printf（“Operator（=-）
ambiguous\n”）；
yyless（yyleng-1）；
…action for＝-…
}
```

上面的规则先打印出一条说明出现二义性的信息，将运算符后面的字母返回给输入串，最后将运算符按“=－”处理．另外，如果希望把“=- a”解释为”＝- a”，这只需要把负号与字母一起退回给输入串等候下次处理，用下面的规则即可：

```
=-[a-zA-Z］｛
printf（“Operator（=-）
ambiguous\n”）；
```

yyless (yyleng-1);
…action for = …
}
3. yywrap ( )

当Lex处理到输入串的文件尾时，自动地调用yywrap（），如果 yywrap（）返回值是 1，那么Lex就认为对输入的处理完全结束，如果yywrap（）返回的值是 0,Lex就认为有新的输入串等待处理。

Lex自动提供一个yywrap（），它总是返回 1，如果用户希望有一个返回 0 的yywrap（ ），那么就可以在"用户子程序部分"自己写一个 yywrap（），它将取代Lex自动提供的那个yywrap（），在用户自己写的ywrap（）中，用户还可以作其他的一些希望在输入文件结束处要作的动作，如打印表格、输出统计结果等，使用yywrap（）的例子在后面举出。

1. 5 识别规则的二义性

有时Lex的程序中可能有多于一条规则与同一个字符串匹配，这就是规则的二义性，在这种情况下，Lex有两个处理原则：

1）能匹配最多字符的规则优先

2）在能匹配相同数目的字符的规则中，先给出的规则优先

例：设有两规则按下面次序给出：

integer kegword action…

[a-z]+ identifier action…

如果输入是integers，则它将被当成标识符处理，因为规则integer只能匹配 7 个字符，而[a-z]+能匹配 8 个字符；如果输入串是integer，那么它将被当作关键字处理，因为两条规则都能与之匹配，但规则integer先给出。

1.6 lex源程序中的辅助定义部分

Lex源程序的第一部分是辅助定义，到目前为止我们只涉及到怎样写第二部分，即识别规则部分的写法，现在来看第一部分的写法。在Lex源程序中，用户为方便起见，需要一些辅助定义，如用一个名字代表一个复杂的正规式。辅助定义必须在第一个％％之前结出，并且必须从第一列开始写，辅助定义的语法是：

name translation

例如用名字IDENT来代表标识符的正规式的辅助定义为

IDENT [a-zA-Z][a-zA-Z0-9]*

辅助定义在识别规则中的使用方式是用运算符｛ ｝将 name括起来，Lex自动地用 translation去替换它，例如上述标识符的辅助定义的使用为：

{IDENT｝action for identifer…

下面我们用辅助定义的手段来写一段识别FORTRAN语言中整数和实数的Lex源程序：

```
D       [0 一 9]
E       [DEde] [-+]? ｛ D｝+
％％
{D}+ printf（“integer”）；
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E} printf( "real" );
```

请注意在辅助定义部分中可以使用前面的辅助定义。例如：定义E时使用了D，但所用的辅助定义必须是事先已定义过的，不能出现循环定义。上面的规则只是说明辅助定义的用法，并不是识别FORTRAN中数的全部规则，因为它不能处理类似 35.EQ．I这样的问题，即会把 35.EQ．I中的 35.E当作实数，怎么解决这种问题请读者思考。

除了上面介绍的辅助定义之外，用户还需要在Lex源程序中使用变量，还需要有一些自己写的子程序。前面已经见过两个常用的变量即yytext和yylong，也介绍过几个Lex提供的子程序yymore,yyless和yywrap,现在介绍用户如何自己定义变量和写子程序。

Lex是把用户写的Lex源程序转换成一个C语言的程序yylex，在转换过程中，Lex是把用户自己的变量定义和子程序照抄到yylex中去,lex规定属于下面三种情况之一的内容就照抄过去；

1）以一个空格或tab起头的行，若不是 Lex的识别规则的一部分，则被照抄到 Lex产生的程序中去。如果这样的行出现在第一个％％之前，它所含的定义就是全局的，即Lex产生的程序中的所有函数都可使用它。如果这样的行紧接在第一个％％之后但在所有识别规则之前，它们就是局部的，将被抄到涉及它的动作相应的代码中去。注意这些行必须符合C语言的语法，并且必须出现在所有识别规则之前。

这一规定的一个附带的作用是使用户可以为Lex源程序或Lex产生的词法分析器提供住解，当然注解必须符合C语言文法。

2）所有被界于两行％{和％}之间的行，无论出现在哪里也无论是什么内容都被照抄过去，要注意 ％｛和％｝必须分别单独占据一行．例如；

```
％｛
# defineENDOFFILE 0
#include “head.h”
int flag
％｝
```

提供上面的措施主要因为在C语言中有一些行如上例中的宏定义或文件蕴含行必须从第一列开始写。

3）出现在第二个％％之后的任何内容，不论其格式如何，均被照抄过去。

1.7 怎样在Unix系统中使用Lex假定已经写好了一个Lex源程序。

怎样在Unix系统中从它得到一个词法分析器呢？

Lex自动地把Lex源程序转换成一个C语言的可运行的程序，这个可运行的程序放在叫Lex.yy.c的文件中，这个C语言程序再经过C编译，即可运行。

例，有一名叫source的Lex源程序，第一步用下面的命令将它转换成lex.yy.c：

```
$ lex source
```

（$是 Unix的提示符）。Lex.yy.c再用下面的命令编译即得到可运行的目标代码 a.out：

```
$cc lex.yy.c－ll
```

上面的命令行中的一 11 是调用Lex的库，是必须使用的，请参看[1]。

这一节内容请读者参看[4]中的lex（1）

Lex可以很方便地与Yacc配合使用，这将在下一章中介绍。

$1.8 例子

这一节举两个例子看看Lex源程序的写法

1．将输入串中所有能被 7 整除的整数加 3，其余部分照原样输出，先看下面的Lex源程序：

```
%%
int k;
［0-9］+{
scanf（-1， yytext，“%d”，＆k）；
if（k ％ 7 = =0）
printf（“%d”，k+3）；
else
printf（“ ％ d”, k）；
}
```

上面的程序还有不足的地方，如对负整数，只是将其绝对值加上 3，而且象X7,49.63 这样的项也做了修改，下面对上面的源程序稍作修改就避免了这些问题。

```
%%
int k;
-?[0-9]+{
scanf（－1，yytext，“%d”，&k）；
printf（“%d”，k%7= =0？k+3;k）；
}
-?[0-9]+ ECHO;
[A-Za-z][A-Za-z0-9]+ ECHO;
```

2．下一个例子统计输人串中各种不同长度的单词的个数，统计结果在数组lengs中，单词定义为由字母组成的串，源程序如下；

```
int lengs [100];
%%
[a-z]+ lengs[yyleng]++;
•|
\n ;
%%
yywrap ( )
｛
int i ；
printf（“Length No.words \n”） ；
for（i＝0；i＜100；i++)
if（lengs［i］＞0）
ptintf（“%5d ％ 10d＼n”，i， lengs[i];
return (1);
}
```

在上面的流程序中，当Lex读入输入串时，它只统计而不输出，到输入串读入完毕后，才在调用yywrap（）时输出统计结果，为此用户自己提供了yywrap（），注意yywrap（）的最后一个语句是返回值 1。

1.9 再谈上下文相关性的处理

在$3 中介绍Lex用的正规式时提到了上下文相关性的表示，这里再详细介绍Lex提供的处理上下文相关的措施。要处理的问题是某些规则在不同的上下文中要采取不同的动作，或者说同样

的字符串在不同的上下文中有不同的解释。例如在程序设计语言中，同一个等号“=”，在说明部分表示为变量赋初值，这时的动作应是修改符号表内容；而在语句部分等号就是赋值语句的赋值号，这时又应该产生相应于赋值语句的代码。因此要依据等号所处的上下文来判断它的含义。
Lex提供了两种主要的方法，
1）使用标志来区分不同的上下文。

标志是用户定义的变量，用户在不同的上下文中为它置不同的值，以区分它在哪个上下文中，这样识别规则就可以根据标志当前值决定在哪个上下文中并采取相应的动作。
例：将输入串照原样输出，但对magic这个词，当它出现在以字母a开头的行中，将其改为first，出现在以b开头的行中将其改为second，出现在以c开头的行中则改为third。
使用标志flag的Lex源程序如下；

```
int flag ;
％％
∧a {flag='a';ECHO;}
∧b {flag='b'; ECHO;}
∧c {flag='c'; ECHO;}
\n {flag=o; ECHO;}
magic{
switch (flag)
{
case 'a' : printf (“first”); break;
case 'b': printf (“second”); break;
case 'c' : printf (“third”); break;
default; ECHO; break;
}
}
```

2）使用开始条件来区分不同上下文

在Lex源程序中用户可以用名字定义不同的开始条件。当把某个开始条件立置于某条识别规则之前时，只有在Lex处于这个开始条件下这条规则才起使用，否则等于没有这条规则。Lex当前所处的开始条件可以随时由用户程序（即Lex动作）改变。
开始条件由用户在Lex源程序的“辅助定义部分”定义，语法是
％Start name1 name2 name3…
其中Start可以缩写成S或s。开始条件名字的顺序可以任意给出，有很多开始条件时也可以由多个％Start行来定义它们。
开始条件在识别规则中的使用方法是把它用尖括号括起来放在识别规则的正规式左边:
＜name1＞expression
要进入开始条件如Name1，在动作中用语句
BEGIN name1
它将Lex所处的当前开始条件改成 name1
要恢复正常状态，用语句
BEGIN 0
它将 Lex恢复到 Lex解释器的初始条件
一条规则也可以在几个开始条件下都起作用，如
<name1 ，name2，name3> rule
使rule在三个不同的开始条件下都起作用。要使一条规则在所有开始条件下都起作用，就不在它

前面附加任何开始条件。
例：解决 1）中的问题，这次用开始条件，Lex源程序如下：
％start AA BB CC
％％
∧a {ECHO； BEGIN AA；}
∧b {ECHO； BEGIN BB；}
∧c {ECHO； BEGIN CC；}
\n {ECHO； BEGIN 0；}
＜AA＞magic printf （“first”）；
＜BB＞magic Printf（“second”）；
＜CC＞magic Printf（“third”）i

1.10 Lex源程序格式总结

为使用方便起见，将Lex源程序的格式，Lex的正规式的格式等总录于此.
Lex源程序的一般格式为：
｛definitions｝
％％
｛rules｝
％％
｛user subroutines｝
辅助定义部分包括以下项目；
1）辅助定义，格式为：
name translation
2）直接按照抄的代码，格式为：
空格 代码
3）直接照抄的代码，格式为：
％{
代码
％｝
4）开始条件，格式为：
％S namel name2…
还有几个其他项目，不常使用故略去。
识别规则部分的格式是
expression action
其中expression必须与action用空格分开，动作如果多于一行，要用花括号括起来。
Lex用的正规式用的运算符有以下一些：
x 字符x
“x”字符x，若为运算符，则不起运算符作用
\x 同上
［xy］ 字符x或y
[x-z] 字符x，或y，或z
[∧x] 除x以外的所有字符
. 除回车换行外的所有字符
∧x 出现在一行开始处的x

＜y＞x 当 Lex处于开始条件 y时， x
X＄ 出现在一行末尾处的x
x？ 可有可无的 X
x* 0 个或多个x
x+ 1 个或多个x
x|y X或y
（x）字符x
x/y 字符x但仅当其后紧随y
{xx} 辅助定义XX的展开
x（m,n）m到n个x

二法程器　　yacc器

2.1 yacc概述

形式语言都有严格定义的语法结构，我们对它们进行处理时首先要分析其语法结构。yace是一个语法分析程序的自动产生器，严格地说Lex也是一个形式语言的语法分析程序的自动产生器。不过Lex所能处理的语言仅限于正规语言，而高级语言的词法结构恰好可用正规式表示，因此Lex只是一个词法分析程序的产生器。yace可以处理能用LALR（1）文法表示的上下文无关语言。而且我们将会看到yace具有一定的解决语法的二义性的功能。

yacc的用途很广，但主要用于程序设计语言的编译程序的自动构造上。例如可移植的C语言的编译程序就是用yacc来写的。还有许多数据库查询语言是用yacc实现的。因此，yacc又叫做“编译程序的编译程序（"A Compiler ComPiler"）。

yacc的工作示意图如下；

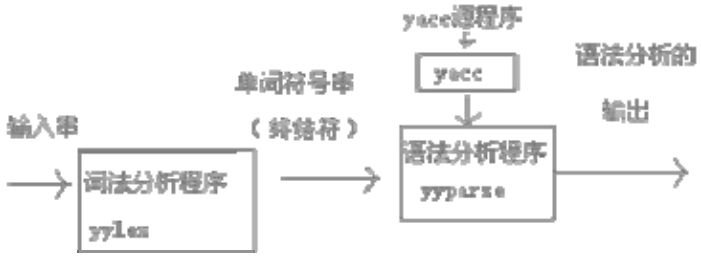


图 2.1 yacc 示意图

在图 2.1 中，“yacc源程序”是用户用yacc提供的一种类似BNF的语言写的要处理的语言的语法描述。yacc会自动地将这个源程序转换成用LR方法进行语法分析的语法分析程序yyparse，同Lex一样，yacc的宿主语言也是C，因此yyParse是一个C语言的程序，用户在主程序中通过调用yyparse进行语法分析。

语法分析必须建立在词法分析的基础之上，所以生成的语法分析程序还需要有一个词法分析程序与它配合工作。yyparse要求这个词法分析程序的名字为yylex。用户写yylex时可以借助于Lex。因为Lex产生的词法分析程序的名字正好是yylex，所以 Lex与yacc配合使用是很方便的，这将在 2.5 的 2.5.3 中详细介绍，请注意词法分析程序也是可以包含在yacc源程序中的。

在yacc源程序中除了语法规则外，还要包括当这些语法规则被识别出来时，即用它们进行归约时要完成的语义动作，语义动作是用C语言写的程序段。语法分析的输出可能是一棵语法树，或生成的目标代码，或者就是关于输入串是否符合语法的信息。需要什么样的输出都是由语义动作和程序部分的程序段来实现的。

下面分节介绍yacc源程序的写法以及在Unix系统中使用yacc的有关命令。

2.2 yacc源程序的一般格式

一个yacc源程序一般包括三部分：说明部分；语法规则部分；程序段部分，这三部分内容依次按下面的格式组织在一起：

说明部分

%%

语法规则部分

%%

程序段部分

上述三部分中说明部分和程序段部分不必要时可省去，当没有程序段部分时，第二个%%也可以省去。但是第一

个%%是必须有的。下面详细介绍各部分的组成及写法。

2.3 yacc源程序说明部分的写法

yacc源程序的说明部分定义语法规则中要用的终结符号，语义动作中使用的数据类型、变量、语义值的联合类

型以及语法规则中运算符的优先级等。这些内容的组织方式如下：

%{
头文件表
宏定义
数据类型定义
全局变量定义
%}
语法开始符定义
语义值类型定义
终结符定义
运算符优先级及结合性定义

2.3.1 头文件表

yacc 直接把这部分定义抄到所生成的 C 语言程序 y.tab.c 中去的，所以要按 C 语言的语法规定来写。头文件表是一

系列 C 语言的＃include 语句，要从每行的第一列开始写，
例如：

```
%{
#include＜stdio.h＞
#include ＜math.h＞
#include＜ctype.h＞
$include “header.h”
%}
.
```

```
.
.
%}
```

2.3.2 宏定义

这部分用 C 语言的 # define 语句定义程序中要用的宏。例如

```
%{
.
.
.
#define EOF O
#dffine max（x,y）((x>y)？x:y)
.
.
.
%}
```

2.3.3 数据类型定义

这部分定义语义动作中或程序段部分中要用到的数据类型，例如：

```
%{
.
.
.
typedef struct interval｛
double lo， hi；
｝INTERVAL；
.
.
.
%}
```

2.3.4 全局变量定义

外部变量（external variable）和 yacc 源程序中要用到的全局变量都在这部分定义，例如：

```
%{
.
.
.
extern int nfg;
douhle dreg[ 26]；
INTERVAL Vreg[26]；
.
.
.
```

%}

另外非整型函数的类型声明也包含在这部分中，请参看 2.6 例 2。

重申一遍，上述四部分括在 %{和%}之间的内容是由 yacc 原样照抄到 y.tab.c 中去，所以必须完全符合 C 语言文法，另外，界符%｛和%｝最好各自独占一行，即最好不写成

```
%{ int x; %}
```

2.3.5 语法开始符定义

上下文无关文法的开始符号是一个特殊的非终结符，所有的推导都从这个非终结符开始，在 yacc 中，语法开始符定义语句是：

```
% start 非终结符……
```

如果没有上面的说明，yacc 自动将语法规则部分中第一条语法规则左部的非终结符作为语法开始符。

2.3.6 语义值类型定义

yycc 生成的语法分析程序 yyparse 用的是 LR 分析方法，它在作语法分析时除了有一个状态钱外，还有一个语义值钱，存放它所分析到的非经结符和终结符的语义值，这些语义值有的是从词法分析程序传回的，有的是在语义动作中赋与的，这些在介绍语义动作时再详细说明。如果没有对语义值的类型做定义，那么 yacc 认为它是整型（int）的，即所有语法符号如果赋与了语义值，则必须是整型的，否则会出类型错，但是用户经常会希望语义值的类型比较复杂，如双精度浮点数，字符串或树结点的指针．这时就可以用语义值类型定义进行说明。因为不同的语法符号的语义值类型可能不同，所以语义值类型说明就是将语义值的类型定义为一个联合（Union），这个联合包括所有可能用到的类型（各自对应一个成员名），为了使用户不必在存取语义值时每次都指出成员名，在语义值类型定义部分还要求用户说明每一个语法符号（终结符和非终结符）的语义值是哪一个联合成员类型。下面举例说明并请参看 2.6 例 2。

```
% union {
int ival
double dval
INTERVAL VVal;
}
%token ＜ival＞ DREG VREG
%token ＜dval＞ CONST
%type ＜dyal＞dexp
%type ＜vval＞vexP
. . .
```

在上述定义中，以%union 开始的行定义了语义值的联合类型，共有三个成员类型分别取名为 ival, dval, vval。

以%token 开始的行定义的是终结符（见 2.3.7）所以 DREG,VREG 和 CONST 都是终结符，尖括号中的名字就是这些终结符的语义值的具体类型。如 DREG 和 VREG 这两个终结符的语义值将是整型（int）的，成员名是 ival。

以%tyPe 开始的行是说明非终结符语义值的类型。如非终结符 dexP 的语义值将是双精度浮点类型，请注意，在 yacc 中非终结符不必特别声明，但是当说明部分有对语义值类型的定义，而且某非终结符的语义值将被存取，就必须用上面的方法定义它的类型。

2.3.7 终结符定义

在 yacc 源程序语法规则部分出现的所有终结符（文字字符 literal 除外）必须在这部分定义，定义方法如下例：

％ token DIGIT LETTER

每个终结符定义行以％token 开头，注意％与 token 之间没有空格，一行中可以定义多个终结符，它们之间用空格分开，终结符名可以由字母，数字，下划线组成，但必须用字母于头。非终结符名的组成规则与此相同。终结符定义行可多于一个。

yacc 规定每个终结符都有一个唯一的编号（token number）。当我们用上面的方式定义经结符时，终结符的编号由 yacc 内部决定，其编号规则是从 257 开始依次递增，每次加 1。但这个规则不适用于文字字符（literal）的终结符。例如在下面的语法规则中，'＋'，'；'就是文字字符终结符:

stats： stats'；' stat；

expr: expr'+' expr;

文字字符终结符在规则中出现时用单引号括起来。它们不需要用％token 语句定义，yacc 对它们的编号就采用该字符在其字符集（如 ASCII）中的值。注意上面两条语法规则末尾的分号是 yacc 元语言的标点符号，不是文字字符终结符。

yacc 也允许用户自己定义终结符的编号。如果这样，那么终结符定义的格式就是：

％token 终结符名 整数

其中“终结符名”就是要定义的终结符，“整数”就是该终结符的编号，每一个这样的行定义一个终结符。特别注意不同终结符的编号不能相同。例如

```
%token BEGIN 100
％token END 101
%token IF 105
％token THEN 200
...
```

在 3.6 中我们说过如果用户定义了语义值的类型，那么那些具有有意义的语义值的终结符其语义值的类型要用 Union 中的成员名来说明，除了在 3.6 段中介绍的定义方法外，还可以把对终结符的定义和其语义值的类型说明分开，例如：

```
%token DREG VREG CONST
％type ＜ival＞ DREG VREG
％type ＜dval＞ CONST
```

2.3.8 运算符优先级及结合性定义

请看下面的关于表达式的文法：

```
%token NAME
expr: expr'+' expr
|expr '–' expr
|expr'*'expr
|NAME
;
```

这个文法有二义性，例如句子：a＋b－c ，可以解释成（a＋ b）一 c 也可以解译成 a＋

（b－ c），虽然这两种解释都合理但造成了二义性，如果将句子 a＋b*C 解释为（a＋b）*c 就在语义上错了。

yacc 允许用户规定运算符的优先级和结合性，这样就可以消除上述文法的二义性。例如规定'＋"一'具有相同的优先级，而且都是左结合的，这样。a＋b－c 就唯一地解释为(a＋ b)一 c。再规定'*'的优先级大于'＋"一'，则 a＋ b* c 就正确地解释为 a＋（b*c） 了，因此上述文法的正确形式应是：

```
%token NAME
%left '+"-'
%left '*'
%%
expr：expr'+' expr
|expr'-' exPr
|expr'*' expr
|NAME
;
```

在说明部分中以%left 开头的行就是定义算符的结合性的行。%left 表示其后的算符是遵循左结合的；%right 表示右结合性，而%nonassoc 则表示其后的算符没有结合性。优先级是隐含的，在说明部分中，排在前面行的算符较后面行的算符的优先级低；排在同一行的算符优先级相同，因此在上述文法中，'＋'和'一'优先级相同，而它们的优先级都小于'*'，三个其符都是左结合的。

在表达式中有时要用到一元运算符，而且它可能与某个二元运算符是同一个符号，例如一元运算符负号"-"就与减号'-'相同，显然一元运算符的优先级应该比相应的二元运算符的优先级高。至少应该与'*'的优先级相同，这可以用 yacc 的%Prec 子句来定义，请看下面的文法：

```
%token NAME
%left '-"+'
%left '*" / '
%%
expr；expr'+' expr
|expr'+' expr
|expr'-'expr
|expr'*' expr
|expr' / ' expr
|'-'expr %prec'*'
|NAME
;
```

在上述文法中，为使一元'一'的优先级与'*'相同，我们使用了子句 %prec'*', 它说明它所在的语法规则中最右边的运算符或终结符的优先级与%Prec 后面的符号的优先级相同，注意%Prec 子句必须出现在某语法规则结尾处分号之前，%prec 子句并不改变'一'作为二元运算符时的优先级。

上面介绍的八项定义，没有必要的部分都可以省去。

2.4. yacc 源程序中语法规则部分的写法

语法规则部分是 yacc 源程序的核心部分，这一部分定义了要处理的语言的语法及要采用的语义动作。下面介绍语法规则的书写格式、语义动作的写法以及 yacc 解决二义性和冲突的具体措施。最后介绍错误处理。

2.4.1 语法规则的书写格式

每条语法规则包括一个左部和一个右部，左右部之间用冒号':'来分隔，规则结尾处要用分号";"标记，所以一条语法规则的格式如下：

nonterminal : BODY；

或

nonterminal：BODY

其中 nonterminal 是一个非终结符，右部的 BODY 是一个由终结符和非终结符组成的串、可以为空，请看几个例子：

```
stat: WHILE bexp DO Stat
;
stat: IF bexp THEN stat
;
stat：/* empty* /
;
```

上面的第三条语法规则的右部为空，用' / *'和'* / '括起来的部分是注解．可以把左部非终结符相同的语法规则集中在一起，规则间用短线'|'分隔，最后一条规则之后才用分号，例如：

```
stat： WHILE bexp DO stat
| IF bexp THEN stat
| / * empty* /
;
```

对语法规则部分的书写有几条建议；

1. 用小写字母串表示非终结符，用大写字母串表示终结符。

2．将左部相同的产生式集中在一起，象上例一样。

3．各条规则的右部尽量对齐，例如都从第一个 tab 处开始。按这样的风格写 yacc 源程序清晰可读性强而且易修改和检查错误。

4．如果产生式（语法规则）需要递归，尽可能使用左递闭方式．例如：

```
seq: item
| seq',' item
;
```

因为用左速归方式可以使语法分析器尽可能早地进行归约，不致使状态栈溢出。

2.4.2 语义动作

当语法分析程序识别出某个句型时，它即用相应的语法规则进行归约，yscc 在进行归约之前，先完成用户提供的语义动作，这些语义动作可以是返回语法符号的语义值，也可以是求某些语法符号的语义值，或者是其他适当的动作如建立语法树，产生目标代玛，打印有关信息等。终结符的语义值是通过词法分析程序返回的，这个值由全局变量（yacc 自动定义的） yylval 带回，如果用户在词法分析程序识别出某终结符时，给 yylval 赋与相应的值，这个值就自动地作为该终结符的语义值。当语义值的类型不是 int 时，要注意 yylval 的值的类型须与相应的终结符的语义值类型一致。语义动作是用 C 语言的语句写成的，跟在相应的语法规则后面，用花括号括起来. 例如：

```
A：'('B')'
{hello（I，“abc”）；}
```

XXX：YYY ZZZ
{printf（“a message＼n”）；
flag＝25；
｝
：

要存取语法符号的语义值，用户要在语义动作中使用以＄开头的伪变量，这些伪变量是 yacc 内部提供的，用户不用定义。伪变量＄＄代表产生式左部非终结符的语义值，产生式右部各语法符号的语义值按从左到右的次序为＄1，＄ 2，… 例如在下面的产生式中：

A :B C D
;

A 的语义值为＄＄，B、C、D 的语义值依次为＄ 1，＄2，＄3。

为说明伪变量的作用，请看下例：有产生式

expr:'('expr')'
;

左的边的 exPr 的值应该等于右连的 expr 的值，表示这个要求的语义动作为，

expr: '('expr')'
{$$=$2;}
;

如果在产生式后面的语义动作中没有为伪变量＄＄赋值， yaCC 自动把它置为产生式右部第一个语法符号的值（即＄1）有较复杂的应用中，往往需要在产生式右部的语法符号之间插入语义动作．这意味着使语法分析器在识别出句型的一部分时就完成这些动作。请看下例：

A：B
｛＄＄＝1；｝
C
{X=$2; y=$3;}

例中 x 的值最后为 1 而 y 的值量为符号 C 的语义值，注意 B 后面的语义动作＄＄＝1 并非将符号 A 的语义值置为 1，这是因为上面的例子是按下面的方式实现的。

＄ACT： /*empty。 /
｛＄＄＝1；｝
;
A：B＄ACTC
{X＝＄2；y=$3；｝
;

即 yacc 自动设置一个非终结符＄ ACT 及一个空产生式用以完成上述语义动作。关于语义动作的实例请读者详细阅读 6 中的两个例子。

2.4.3 yacc 解决二义性和冲突的方法

在 2.3.8 中已涉及到二义性和冲突的问题，这里再集中介绍一下，这在写 Yacc 源程序时会经常碰到。二义性会带来冲突。在 2.3.8 中我们介绍了 yacc 可以用为算符确定优先级和结合规则解决由二义性造成的冲突，但是有一些由二义性造成的冲突不易通过优先级方法解决，

如有名的例子：

stat：IF bexp THEN stat
|IF bexp THEN stat ELSE
stat

;

对于这样的二义性造成的冲突和一些不是由二义性造成的冲突，Yacc 提供了下面两条消除二义性的规则：

A1．出现移进／归约冲突时，进行移进；

A2．出现归约／归约冲突时，按照产生式在 yacc 源程序中出现的次序，用先出现的产生式归约。

我们可以看出用这两条规则解决上面的 IF 语句二义性问题是合乎我们需要的。所以用户不必将上述文法改造成无二义性的。当 Yacc 用上述两条规则消除了二义性，它将给出相应信息。

下面再稍微严格地介绍一下 Yacc 如何利用优先级和结合性来解决冲突的。

Yacc 源程序中的产生式也有一个优先级和结合性．这个优先级和结合性就是该产生式右部最后一个终结符或文字字符的优先级和结合性，当使用了％Prec 子句时，该产生式的优先级和结合性由％Prec 子句决定。当然如果产生式右部最后一个终结符或文字字符没有优先级或结合性，则该产生式也没有优先级或结合性。

根据终结符（或文字字符）和产生式的优先级和结合性，Yacc 又有两个解决冲突的规则：

P1．当出现移进/归约冲突或归约／归约冲突，而当时输入符号和语法规则（产生式）均没有优先级和结合性，就用 AI 和 A2 来解决这些冲突。

P2．当出现移进／归约冲突时，如果输入符号和语法规则（产生式）都有优先级和结合性，那么如果输入符号的优先级大于产生式的优先级就移进如果输入符号的优先级小于产生式的优先级就归约。如果二者优先级相等，则由结合性决定动作，左结合则归约，右结合则移进，无结合性则出错。

用优先级和结合性能解决的冲突，yacc 不报告给用户。

2.4.4 语法分析中的错误处理

当进行语法分析时发现输入串有语法错误，最好能在报告出错信息以后继续进行语法分析，以便发现更多的错误。

yacc 处理错误的方法是：当发现语法错误时，yacc 丢掉那些导致错误的符号适当调整状态栈。然后从出错处的后一个符号处或跳过若干符号直到遇到用户指定的某个符号时开始继续分析。

Yacc 内部有一个保留的终结符 error，把它写在某个产生式的右部，则 Yacc 就认为这个地方可能发生错误，当语法分析的确在这里发生错误时，Yacc 就用上面介绍的方法处理，如果没有用到 error 的产生式，则 Yacc 打印出“Syntax error”，就终止语法分析。

下面看两个使用 error 的简单例子：

1．下面的产生式

```
stat: error
;
```

使 yacc 在分析 stat 推导出的句型时，遇到语法错误时跳过出错的部分，继续分析（也会打印语法错信息）

2．下面的产生式

```
stat: error ';'
;
```

使 yacc 碰到语法错时，跳过输入串直到碰到下一个分号才继续开始语法分析。

如果语法分析的输入串是从键盘上输入的（即交互式），那么某一行出错后，希望重新输入这一行，并使 yacc 立即开始继续分析，这只要在语义动作中使用语句 yyerror 即可，如下例：

```
input: error'\n'
{yyerror;
```

```
printf (“Reenter last line:”);}
input
{$ $ = $4；}
;
```

关于错误处理请参看[2]和 6 的例子。

2.5 程序段部分

程序段部分主要包括以下内容：主程序 main（）；错误信息执行程序 yyerror（s）；词法分析程序yylex（）；用户在语义动作中用到的子程序，下面分别介绍。

2.5.l主程序

主程序的主要作用是调用语法分析程序yyparse（），yyparse（）是yacc 从用户写的yacc源程序自动生成的，在调用语法分析程序yyparse（）之前或之后用户往往需要做一些其他处理，这些也在main（）中完成，如果用户只需要在main()中调用yyparse（），则也可以使用Unix的yacc库（一ly）中提供的main（）而不必自己写。库里的main（）如下：

```
main（）  {
return（yyparse（））；
}
```

2.5.2 错误信息报告程序

yacc 的库也提供了一个错误信息报告程序，其源程序如下:

```
＃include <stodio. h＞
yyerror (s) char * s{
fprintf (stderr, “%s\n”,s);
}
```

如果用户觉得这个 yyerror（s）太简单。也可以自己提供一个，如在其中记住输入串的行号并当 yyerror（s）被调用时，可以报告出错行号。

2.5.3 词法分析程序

词法分析程序必须由用户提供，其名字必须是 yylex，调法分析程序向语法分析程序提供当前输入的单词符号。yylex 提供给 yyparse 的不是终结符本身，而是终结符的编号，即 token number,如果当前的终结符有语义值，yylex 必须把它赋给 yylval。

下面是一个词法分析程序例子的一部分。

```
yylex（）{
extern int yylval
int c；
…
c＝ getchar（）；
…
switch（c）{
…
case ‘0’：
case ‘1’：
```

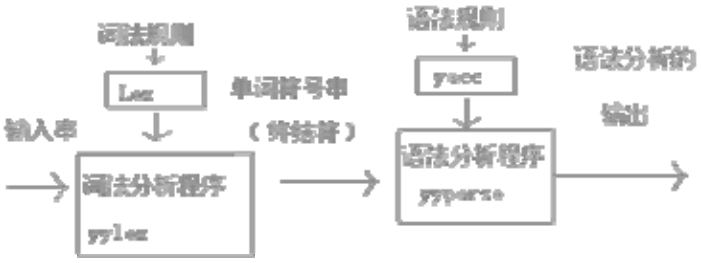
```
...
case “9”
yylval=c-'0'
return (DIGIT);
…
}
…
```

上述词法分析程序碰到数字时将与其相应的数值赋给 yylval,并返回 DIGIT 的终结符编号，注意 DIGIT 代表它的编号（如可以通过宏来定义）。

用户也可以用 Lex 为工具编写记号词法分析程序，如果这样，在 yacc 源程序的程序段部分就只需要用下面的语句来代替词法分析程序：

```
#include "lex.yy.c"
```

为了清楚 lex 与 yacc 的关系，我们用下图表示 lex 与 yacc 配合使用的情况；



在 Unix 系统中，假设 lex 源程序名叫 plo.l.yace 源程序名叫 plo.y,则从这些源程序得到可用的词分析程序中语法分析程序依次使用下述三个命令：

```
Lex plo.l

yacc plo,y

cc y,tab,c -ly-ll
```

第一条命令从lex源程序plo.l产生词法分析程序，文件名为lex.yy.c第二命令从yacc源程序plo.y产生语法分析程序，文件名为y.tab.c;第三条命令将y.tab.c这个c语言的程序进行编译得到可运行的目标程序。

第三条命令中-11 是调用lex库，-ly是调用yacc库，如果用户在yacc源程序的程序段部分自己提供了main（）和yyerror（s）这两个程序，则不必使用-ly．另外如果在第二条命令中使用选择项-v，例如：

```
yacc －v plo.y
```

则yacc除产生y.tab.c外，还产生一个名叫y.output的文件，其内容是被处理语言的LR状态转换表,这个文件对检查语法分析器的工作过程很有用。”

请参看[4]的 Lex（1）和 yacc（1）

2.5.4 其他程序段

语义动作部分可能需要使用一些子程序，这些子程序都必须遵守C语言的语法规定，这里不

多讲了。

2.6 yacc源程序例子说明

例 1．用yacc描述一个交互式的计算器，该计算器有 26 个寄存器，分别用小写字母a到z表示，它能接受由运算符+、-、*、／、％（取模）、&（按位求与）、|（按位求或）组成的表达式，能为寄存器赋值，如果计算器接受的是一个赋值语句，就不打印出结果，其他情况下都给出结果，操作数为整数，若以 0（零）开头，则作为八进制数处理。

例 1 的yacc源程序见附录F

读者从例 1 中可以看出用优先关系和二义性文法能使源程序简洁，还可看到错误处理方法，但例 1 不足之处是它的词法分析程序太简单，还有对八进制与十进制数的区分也最好在词法分析中处理.

例 2．这个例子是例 1 的改进，读者能看到语义值联合类型的定义及使用方法和如何模拟语法错误并进行处理，该树也是描述一个交互式的计算器，比例 1 的计算器功能强，它可以处理浮点数和浮点数的区间的运算，它接受浮点常数，以及+、-、*、/、一元-和=（赋值）组成的表达式，它有 26 个浮点变量，用小写字母a到z表示，浮点数区间用一对浮点数表示：(x,y)

其中x小于或等于y，该计算器有 26 个浮点数区间变量，用大写字母A到Z表示。和例 1 相似，赋值语句不打印出结果，其他表达式均打印出结果，当发生错误时给出相应的信息。

下面简单总结一个例 2 的一些特点。

1．语义值联合类型的定义

区间用一个结构表示，其成员给出区间的左右边界点，该结构用c语言的typedef语 句定义，并赋与类型名INTERVAL。 yacc的语义值经过％union定义后，可以存放整型，浮点及区间变量的值，还有一些函数（如 hil,vmul,vdiv）都返回结构类型的值。

2．yacc的出错处理

源程序中用到了YYERROR来处理除数区间中含有 0 或除数区间端点次序倒置的错误，当碰到上述错误时，YYERROR使yacc调用其错误处理机构，丢掉出错的输入行，继续处理。

3．使用有冲突的文法

如果读者在机器上试试这个例子，就会发现它包含 18 个移进／归约冲突，26 个归约／归约冲突，请看下面两个输入行：

2.5+(3.5-4.0)

2.5+(3.5,4.0)

在第二行中，2.5 用在区间表达式中，所以应把它当作区间处理，即要把它的类型由标量转换成区间量，但yacc只有当读到后面的','时才知道是否应该进行类型转换，此时改变主意为时已晚，当然也可以在读到 2.5 时再向前看几个符号来决定 2.5 的类型，但这样实现较困难，因为yacc本身不支持，该例是通过增加语法规则和充分利用yacc内部的二义性消除机构来解决问题的。在上述文法中，每一个区间二元运算都对应两条规则，其中一条左操作数是区间，另一条左操作数是标量，yacc可以根据上下文自动地进行类型转换。除了这种情况外，还存在着其他要求决定是否进行类型转换的情形，本例将标量表达式的语法规则放在区间表达式语法规则的前面，使运算量的类型先为标量。直到必要时再转换成区间，这样就导致了那些冲突。有兴趣的读者不妨仔细看一看这个源程序和yacc处理它时产生的y.output文件分析一下yacc解决冲突的具体方法。要注意上述解决类型问题的方法带有很强的技巧性，对更复杂的问题就难以施展了。

（1）解释程序

任何一种高级语言（如C）都精确规定了数据结构和程序的执行顺序，即定义了一台计算机（称作A）。 A的存储结构是C的数据结构，A的控制器控制C程序的执行，A的运算器完成C的语句操作，A的机器语言即是C，每个C程序都规定了计算机A从初始状态到终止状态的转换规则。

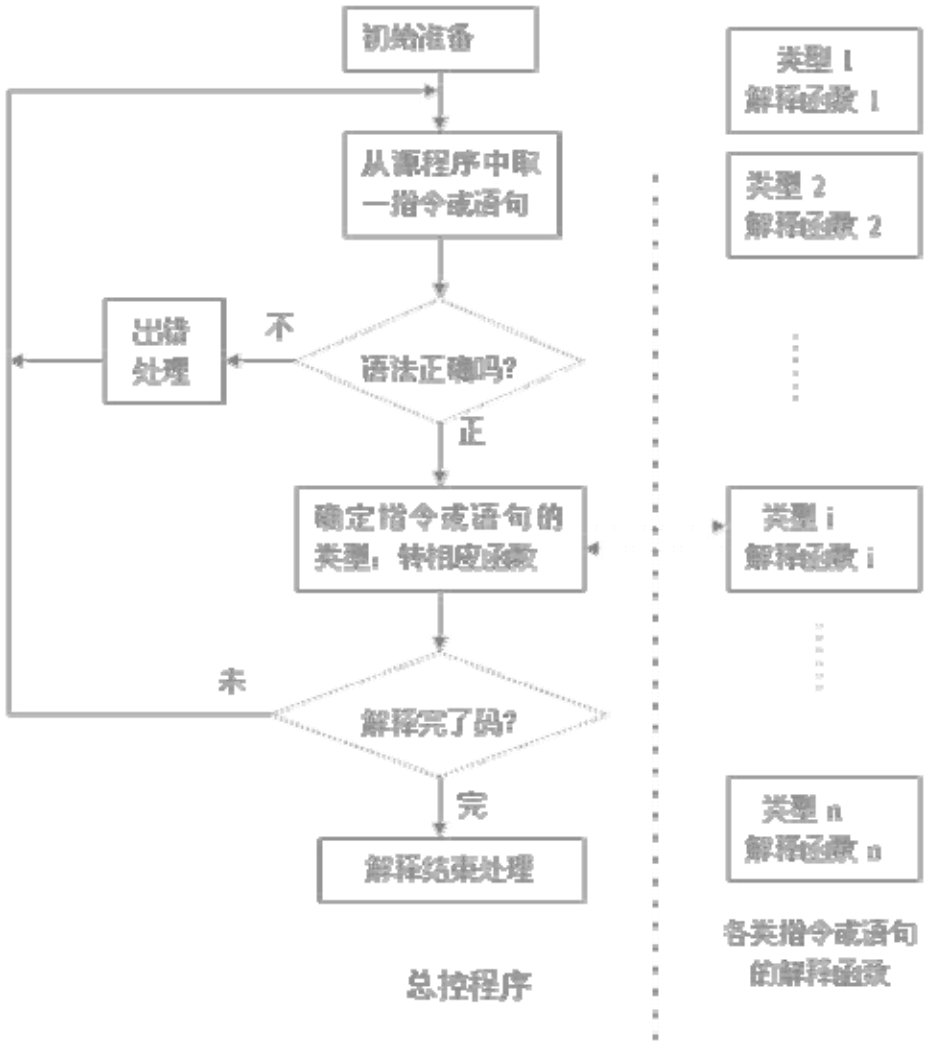
我们用一台通用计算机B来模拟计算机A的执行，为达到这个目的，必须用计算机B的机器语言构造一组程序以支持用A的机器语言C编写的程序的执行。换句话说，我们在一台通用计算机B上用软件构造了一台高级语言计算机A。这个过程称为软件模拟（或软件解释）。

（2）编译方式和解释方式的主要区别

解释程序由总控程序和各类指令或语句的解释函数组成，它按照源程序的逻辑流程，直接解释执行源程序或源程序的内部形式。对于循环中的语句，每次循环执行要进行相当复杂的分析过程，因此，用解释方式执行是低效的；而用编译方式（参见 2.1.4）仅在产生目标代码时分析一遍，最后执行目标代码，这是它们之间的主要区别。

（1） 解释程序基本原理

模拟程序的解释算法如图 2.6，它按输入程序的执行顺序解释执行，产生程序规定的输.



2.6 图 解释程序的结构和工作流程示意图

（2） 程序语言的通常实现方式

纯碎的解释和纯碎的编译是两个极端情况，很少使用它们。例如，在处理汇编语言时才使用纯碎的编译方式；在处理操作系统的控制语言和交互语言时才使用纯碎的解释方式。程序语言的通常实现方式如图 2.7 所示，是编译技术和解释技术的结合。

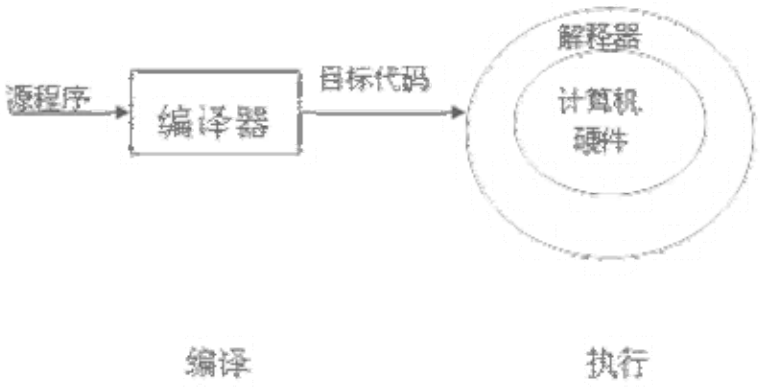


图 2.7 通常的语言实现示意图

采用哪一种处理方式，是由被实现的语言和实现环境（在什么计算机系统上实现）两者决定的，而被实现的语言，它的数据结构和控制结构更起着重要作用。程序语言被分为编译型和解释型，为了提高效率并尽早检查出源程序中的错误，尽量采用编译方式,既使采用解释方式，也尽量生成接进目标代码的中间代码。

C、C++、FORTRAN、PASCAL 和 ADA 通常采用编译方式实现，称为编译型语言。编译器把源程序翻译成目标计算机语言程序，解释器只提供一个运行库，以支持目标语言程序中计算机语言没有提供的操作。一般来说，编译器比较复杂庞大，它的侧重点是产生尽可能高效运行的目标语言程序。

LISP、ML、Prolog 和 Smalltalk 通常采用解释方式实现,称为解释型语言。在实现中，编译器仅产生易于解释的中间代码，中间代码不能在硬件机上直接执行，而由解释器解释执行。这种实现的编译器相对简单，实现的复杂工作在于构造解释器。

Java 不象 LISP，而更象 C++，但由于 Java 运行在网络环境上，Java 通常被当作解释型语言，编译器产生一种字节码的中间语言，被各个终端上的浏览器创建的解释器解释执行。

从 lex&yacc 说到编译器

作者:tangl_99

学过编译原理的朋友肯定都接触过 LEX 这个小型的词法扫描工具. 但是却很少有人真正把 LEX 用在自己的程序里. 在构造专业的编译器的时候,常常需要使用到 lex 和 yacc. 正是因为这两个工具,使得我们编写编译器,解释器等工具的时候工作变得非常简单.不过话说回来,会使用 lex 和 yacc 的人也确实不简单. Lex 和 yacc 里面牵涉到一系列的编译原理的理论知识,不是简单地看看书就能搞懂的. 本文只是简单地介绍一下 lex 和 yacc 的使用方法.相关编译理请查看本科教材.

国内大学教材里面对于 lex 和 yacc 的介绍很少,有些根本就没有,不过在国外的编译原理教材介绍了很多. 按照学科的分类,国内大学本科里面开的<<编译原理>>教程只是讲解编译的原理,并不讲解实践. 而对于实践方面则是另外一门学科<<编译技术>>. 关于编译技术的书籍在国内是少之又少. 前不久, 听说上海交大的计科内部出版过编译技术的教材.可惜我们这些人就无法得见了. 还好,机械工业出版社引进了美国 Kenneth C.Louden 所著的经典著作<<编译原理及实践>>中,比较详细地介绍 lex 和 yacc 的使用.

Lex 属于 GNU 内部的工具,它通常都是 gcc 的附带工具. 如果你使用的 Linux 操作系统,那么肯定系统本身就有 lex 和 yacc,不过 yacc 的名字变成了 bison. 如果你使用的 Windows 操作系统,那么可以到 cygwin 或者 GNUPro 里面找得到. 网上也有 windows 版本 lex 和 yacc,大家可以自己去找一找.

本文一共有两篇,一篇是介绍 lex,另一篇是介绍 yacc. Lex 和 yacc 搭配使用, 我们构造自己的编译器或者解释器就如同儿戏. 所以我把本文的名字叫做黄金组合.

本文以 flex(Fase Lex)为例,两讲解如何构造扫描程序.

Flex 可以通过一个输入文件,然后生成扫描器的 C 源代码.

其实扫描程序并不只用于编译器 .比如编写游戏的脚本引擎的时候,我看到很多开发者都是自己写的扫描器,其算法相当落后(完全没有 DFA 的概念化), 甚至很多脚本引擎开发者的词法扫描器都没有编写,而是在运行过程中寻找 token(单词). 在现代的计算机速度确实可以上小型的脚本引擎在运行中进行词法扫描, 但是作为一个合格的程序员, 或者说一个合格的计算机本科毕业生而来说, 能够运用编译原理与技术实践,应该是个基本要求.

如果要说到词法分析的扫描器源代码编写, 其实也很简单, 会 C 语言的人都会写. 可是 Kenneth Louden 在<<编译原理及技术>里面,花了 50 多页,原因就是从理论角度,介绍标准的,可扩展的,高效的词法扫描器的编写. 里面从正则表达式介绍到 DFA(有穷自动机),再到 NFA(非确定性有穷自动机),最后才到代码的编写. 以自动机原理编译扫描器的方法基本上就是现在词法扫描器的标准方法, 也就是 Lex 使用的方法. 在 Lex 中,我们甚至不需要自己构造词法的 DFA, 我们只需要把相应的正则表达式输入, 然后 lex 能够为我们自己生成 DFA,然后生成源代码,可谓方便之极.

本文不讲 DFA, lex 的输入是正则表达式, 我们直接先看看正则表达式方面知识就可以了.

1.正则表达式(regular expression):

对于学过编译原理的朋友来说,这一节完全可以不看.不过有些东西还是得注意一下,因为在 flex 中的正则表达式的使用有些具体的问题是在我们的课本上没有说明的.

先看看例子:

例 1.1

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 1.2

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 1.3

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 1.4

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 1.5

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 1.6

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.

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.

2.flex 的使用

看了第一篇的关于正则表达式的说明后,下面我们就来通过它,使用 flex 这个词法分析工具来构造我们的编译器的词法分析器.

关于 lex 的教程应该是很多,这里我就简单地介绍一下,然后着重后面的 lex 和 yacc 的配合使用以及其技巧.所以,如果你不看了后还是不太明白 lex 或者 yacc 的使用,请你自己上网去查查,这方面的教程是很多的.我知道的一篇常见的就是

Yacc 与 Lex 快速入门
Lex 与 Yacc 介绍

它的作者就是 Ashish Bansal.

Flex 就是 fast lex 的意思.而 lex 就是 Lexical Analyzar 的意思.flex 可以在 cygwin 或者 gnupro 中找到.它是 unix 的一个工具,属于 GNU 组织产品.网上也可以找到单独可以在 windows 下用的版本.我们一般把我们的词法扫描程序要扫描的一些单词(token) 用正则表达式写好,然后作为 lex 的输入文件,输入命令 flex xxx.l(xxx.l 就是输入文件),lex 经过处理后,就能得到一个名字叫 lex.yy.c 的 C 源代码.这个 C 源代码文件,就是我们的词法扫描程序. 通常 lex 为我们生成的词法分析器的 C 源代码都是十分复杂而且庞大的,我们一般根本不会去查看里面的代码(放心好了,flex 这个东西不会出错的)

下面让我们看看几个我已经使用过的几个 lex 输入文件.
这是一个前段时间我为 GBA 上的一个 RPG 游戏写的脚本引擎所使用的 lex 输入文件(部分)
例 2.1

```
%{
/* need this for the call to atof() below */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "globals.h"

%}
digit        [0-9]
number       ("-"|"+")?{digit}+
hexnumber    "0x"({digit}|[a-fA-F])+
letter       [a-zA-Z]
identifier   ({letter}|_)({number}|{letter}|_)*
newline      [\n]
whitespace   [ \t]+
string       \"[^"]*\"
comment      "#"[^#]*"#"
```

```
%%

{string}      { return VM_STRING;        }
"Logo"        { return VMIN_LOGO; }
"FaceIn"      { return VMIN_FACEIN; }
"FaceOut"     { return VMIN_FACEOUT; }
"LoadTile"    { return VMIN_LOAD_TILE;    }
"CreateRole" { return VMIN_CREATE_ROLE; }
"ReleaseRole" { return VMIN_RELEASE_ROLE;}
"CreateMap"   { return VMIN_CREATE_MAP;   }
"ReleaseMAP" { return VMIN_RELEASE_MAP;}
"ShowBitmap" { return VMIN_SHOWBITMAP;   }
"CreateDialog" { return VMIN_CREATE_DIALOG; }
"ReleaseDialog" { return VMIN_RELEASE_DIALOG;}
"Fight"       { return VMIN_FIGHT;       }
"Delay"       { return VMIN_DELAY;       }
"PressA"      { return VMIN_PRESS_A;     }
"PressB"      { return VMIN_PRESS_B;     }
"PressR"      { return VMIN_PRESS_R;     }
"PressL"      { return VMIN_PRESS_L;     }
"PressStart" { return VMIN_PRESS_START; }
"PressSelect" { return VMIN_PRESS_SELECT;}
{number}      { return VM_NUMBER;        }
{whitespace} { /* skip whitespace */     }
{identifier} { return VM_ID;             }
{newline}     ;
.             ;
%%
int yywrap()
{
     return 1;
}
```

这里的 lex 输入文件一共有三个部分,用%%分开.第一部分中的%{和}%中的内容就是直接放在 lex 输出 C 代码中的顶部.我们通过它可以来定义一些所需要的宏,函数 和 include 一些头文件等等.我的这个 lex 输入文件中也没什么特别的东西,就是常规的 C 源文件的 include 头文件

```
%{
/* need this for the call to atof() below */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "globals.h"
%}
```

第一部分中,除了前面的%{和}%包含的部分,下面的就是正则表达式的定义.
看了第一篇的正则表达式,这样你就能够在这里派上用场了.
让我们来看看我这里定义的正则表达式:

```
digit        [0-9]
number       ("-"|"+")?{digit}+
hexnumber    "0x"({digit}|[a-fA-F])+
letter       [a-zA-Z]
identifier   ({letter}|_)({number}|{letter}|_)*
newline      [\n]
whitespace   [ \t]+
string       \"[^"]*\"
comment      "#"[^#]*"#"
```

digit 就不用说了,就是 0-9 的阿拉伯数字定义,第一篇文章中也举了这个例子.number 就是 digit 的 1 到无限次的重复,再在其前面加上”+”和”-“符号.
注意:
“a”: 即使 a 是元字符,它仍是字符 a
\a: 当 a 是元字符时候,为字符 a
a?: 一个可选的 a,也就是说可以是 a,也可以没有 a
a|b: a 或 b
(a): a 本身
[abc]: 字符 a,b 或 c 中的任一个
[a-d]: a,b,d 或者 d 中的任一个
[^ab]: 除了 a 或 b 外的任何一个字符
.: 除了新行之外的任一个字符
{xxx}: 名字 xxx 表示的正则表达式

这里需要特别说明的就是

```
newline      [\n]
```

newline 就是新行,这里我使用了[]把\n 换行号括起来.因为如果我直接用\n 表示的话,那么按照上面的规则,那就会看成\和 n 两个字符,所以我使用了[\n]. 有些时候 newline 也被写成[\n]|[\r\n].因为在文本文件中,一般换行一次,那么就是一个\n(0xA),可是在二进制文件中,换行有时候 又是\r\n(0xD,0xA)一共两个字符号.

第二部分就是定义扫描到正则表达式的动作.
这些动作其实就是 C 代码,它们将会被镶嵌在 lex 输出的 C 文件中的 yylex()函数中.
上面的例子的动作其实十分平常,就是返回一个值.
我们在外部使用这个 lex 为我们生成 C 代码的时候,只需要使用它的 int yylex()函数.当我们使用一次 yylex(),那么就会自动去扫描一个匹配的正则表达式,然后完成它相应的动作.这里的动作都是返回一值,那么 yylex 就会返回这个值.通常默认 yylex 返回 0 时候,表示文件扫描结束,所以你的动作中最好不要返回 0,以免发生冲突.当然,动作中也可以不返回一 值,那么 yylex 就会完成这个动作后自动扫描下一个可以被匹配的字符串,一直到扫描到文件结束.

当扫描到一个可以被匹配的字符串,那么这个时候,全局变量 yytext 就等于这个字符串

请大家一定记住这些正则表达式的顺序.
如果出现一个字符串,可以同时匹配多个正则表达式,那么它将会被定义在前面的正则表达式匹配.
所以我一般把字符串 string 定义在最前面.
如果文件中的字符没有被 lex 输入文件中任何一个字符匹配,那么它会自动地被标准输出.所以大家一定要记住在每个正则表达式处理完毕后,一定要加上{newline}和.这两个正则表达式的动作.

好,让我们看看 lex 为我们输出 C 文件中提供一些常量

Lex 变量

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。）

例 2.2
这是<<编译原理与实践>>书中配套的源代码的 lex 输入文件.大家可以参考一下,作者为它自己定义的一个 Tiny C 编译所做的词法扫描器.

```
/****************************************************/
/* File: tiny.l                                     */
/* Lex specification for TINY                       */
/* Compiler Construction: Principles and Practice   */
/* Kenneth C. Louden                                */
/****************************************************/

%{
#include "globals.h"
#include "util.h"
#include "scan.h"
/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN+1];
%}

digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+
```

```
%%

"if"            {return IF;}
"then"          {return THEN;}
"else"          {return ELSE;}
"end"           {return END;}
"repeat"        {return REPEAT;}
"until"         {return UNTIL;}
"read"          {return READ;}
"write"         {return WRITE;}
":="            {return ASSIGN;}
"="             {return EQ;}
"<"             {return LT;}
"+"             {return PLUS;}
"-"             {return MINUS;}
"*"             {return TIMES;}
"/"             {return OVER;}
"("             {return LPAREN;}
")"             {return RPAREN;}
";"             {return SEMI;}
{number}        {return NUM;}
{identifier}    {return ID;}
{newline}       {lineno++;}
{whitespace}    {/* skip whitespace */}
"{"             { char c;
                do
                { c = input();
                  if (c == EOF) break;
                  if (c == '\n') lineno++;
                } while (c != '}');
               }
.               {return ERROR;}

%%

TokenType getToken(void)
{ static int firstTime = TRUE;
  TokenType currentToken;
  if (firstTime)
  { firstTime = FALSE;
    lineno++;
    yyin = source;
    yyout = listing;
```

```
  }
  currentToken = yylex();
  strncpy(tokenString,yytext,MAXTOKENLEN);
  if (TraceScan) {
    fprintf(listing,"\t%d: ",lineno);
    printToken(currentToken,tokenString);
  }
  return currentToken;
}
```

这里有点不同的就是, 作者用了另外一个 getToken 函数来代替 yylex 作为外部输出函数.其中 getToken 里面也使用了 lex 默认的输出函数 yylex(),同时还 做了一些其它的事情.不过我建议大家不要像作者那样另外写自己的结果输出函数,因为在后面,需要和 yacc 搭配工作的时候,yacc 生成的语法分析程序只 认名字叫 yylex()的词法结果输出函数.

```
if (firstTime)
  { firstTime = FALSE;
    lineno++;
    yyin = source;
    yyout = listing;
  }
```

其中的 yyin,yyout,source,listing 都是 FILE*类型.yyin 就是要 lex 生成的词法扫描程序要扫描的文件,yyout 就是基本输出文件(其实我们通常都不用 yyout,即使要生成一些输出信息,我们都是自己通过 fprintf 来输出).

```
"{"             { char c;
                  do
                  { c = input();
                    if (c == EOF) break;
                    if (c == '\n') lineno++;
                  } while (c != '}');
                }
```

其中, 作者的这个 Tiny C 是以{}来包括注释信息.作者并没有写出注释信息的正则表达式,但是它可以通过检索“{”,然后用 lex 内部函数 input()一一检查 { 后面的字符是不是 } 来跳过注释文字.(C 语言的/* */注释文字正则表达式十分难写,所以很多时候我们都用这种方法直接把它的 DFA(扫描自动机)写出来).

本文就是通过简单地举出两个比较实际的例子来讲解 flex 输入文件的.再次说明,如果你是第一次接触 lex,那么请看看前面我推荐的文章,你可以在 IBM 的开发者网上查到.下一篇关于 yacc 于 BNF 文法的说明也是如此.请大家先参考一下其它标准的教程.

3.范式文法

从这一节开始, 我们就算进入编译器构造的正题了.不得不说,前面的词法扫描器在整个编译器部分只是个很小很小的组成,而这两节讲述的语言构造器才能真正为我们的编译工作 起到重要的作用.这些东西相信大家在大学的编译原理的课程已经学了不少,那么本文我也只是大致地带过,让大家回忆起大学的知识,重要的 yacc 使用技巧等 等,我将在后面的内容讲出.

例 3.1

exp -> *exp op exp* | ***(exp)*** | ***number***

op -> + | - | ***

这里就是一个定义的带有加法,减法,乘法的简单整数算术表达式的文法.其中粗体表示的是终结符号,也就是不能有产生式生成的符号.而 exp,op 就是非终结符,它们都是由一个”->”符号来产生的.

比如 100 + 222 *123123 －(888+11)就是符合上述文法的具体的表达式.

注意,在文法定义中,是可以递归的.所以 exp 产生式右边的式子中可以再次出现 exp.

这里的|和正则表达式一样,表示的选择的意思,也就是说,exp 可以是 exp op exp 或者(exp)再或者 number.

下面让我们看看<<编译原理及实践>>书中的一个关于 BNF 文法的介绍.

比如说我们有个数学表达式(34-3)*42,然后我们来看看上面的 exp 文法怎么来推导识别它.

(1) exp => exp op exp [exp ->exp op exp]
(2) => exp op number [exp ->number]
(3) => exp * number [op -> *]
(4) => (exp) * number [exp ->(exp)]
(5) => (exp op exp) * number [exp ->exp op exp]
(6) => (exp op number)* number [exp -> number]
(7) => (exp － number) * number [op -> -]
(8) => (number－number)* number [exp -> number]

最终,exp 里面全部的非终结符号全部变成了终结符号.那么推导完成.

这种推导十分像我们在离散数学中讲到的命题推理.其实形式语言的推导的数学基础就是我们离散数学的命题推理.

在推导过程中,其实就是把原来的文法中的递归展开.那么我们在推导的过程,也就很容易实现分析树的生成.而分析树就是我们编译程序中十分重要的信息源.我们之所以前面又做词法分析,又做语法分析的目标就是为了生成分析树.有了它,我们编译程序在后面的代码生成过程中将变得容易百倍.

请看:

例 3.2

同样是<<编译原理及实践>>书上的例子.

设 E -> E+a | a 表示的文法为 G,那么考虑它生成的表达 L(G)

如果由标准的数学定义,那么我们用公式 L(G)={s | exp =>* s }表示一种文法 G.

s 代表记号符号的任意数组串,也就是我们的终结符号.而 exp 代表非终结符号,=>*表示一系列的从非终结符到终结符号的推导过程.这里*有点像我们在讲述正则表达式中的*符号一样,它表示 0 到无限次的重复.所以=>*就是表示 0 次到无限次的推导过程.

L(G) = {a,a+a,a+a+a,a+a+a+a,…}

E => E+a => E+a+a => E+a+a+a

同时,在我们的编译课本上,又经常讲述另一种数学表达方式来阐述文法的定义.

G=(T,N,P,S)

注意,这里的 T,N,P,S 都是集合.

T 表示终结符号(terminal),也就是这里{a,+}

N 表示非终结符号(nonterminal),也就是这里{E},但是 N 不能与 T 相交.

P 表示产生式(production)或者文法规则(grammar rule)的集合,这里它只有一个元素: E -> E+a

S 表示集合 N 的开始符号(start symbol).关于 S,本人也搞不清楚它的用处,所以很抱歉!

例 3.3

这是我们 C 程序语言中经常使用 if else 文法

statement -> if-stmt | **other**

if-stmt -> **if (**exp**)** statement | **if (**exp**)** statement **else** statement

exp -> **0**|**1**

statement 就是我们 C 语言中使用语句,它的产生式包括了两种可能,一是 if-stmt 语句,二是 other.然后我们又另外定义 if-stmt 语句的产生式.这里有 两种情况,一是没有 else 的,另外就是有 else 的.里面我们又使用了递归.if-stmt 本来是包含在 statement 里面的,可是我们又在 if- stmt 的产生式中使用 statement.正是因为文法中允许递归,所以它比起我们前面讲的正则表达式有更广泛的表示能力,但同时,文法的推导识别也更 加法复杂.按照编译原理的书籍,一般讲完 BNF 文法后,就要重点讲解文法的推导算法.一共有两种,一是 LL 算法,自顶向下的算法,二是 LR 算法,自底向上 的算法.LL 算法比较简单,其中还有一种特殊的情况,就是我们下一节要讲的递归下降的算法.由于 C 语言中的函数本来就可以递归,那么实现这中递归下降的算 法是十分简单的,而且对于我们一般的程序设计语言来说,虽然它的算法能力很弱,但是已经是足够用了.而关于LR的算法,那么就是一个大难题了.它的算法能 力最强,但是实现起来十分困难,还好,已经有科学家为我们提供了 yacc(或者叫bison)这个工具,可以来自动生成LR的文法推导算法.这就是我们一 直在提到的yacc工具了.

回过头来,我们看看下面的程序

if(0) other else other

的分析树

思考: 为什么要把文法最终分析成树结构?
因为文法本身是递归的,而表示的递归的最好数据结构就是树,所以我们把文法弄成树结构后,后面在处理代码生成等问题上,也可以用递归来很容易地完成.

例 3.4
这里我给出 microsoft 在 msdn 中对于 C 语言的 statement 的文法
注意,这里用:符号替代了我们前面产生式的->

statement :
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement
try-except-statement /* Microsoft Specific */
try-finally-statement /* Microsoft Specific */
jump-statement :
goto *identifier* **;**
continue;
break;
return *expression* $_{opt}$ **;**
compound-statement :
{ *declaration-list* $_{opt}$ *statement-list* $_{opt}$ **}**
declaration-list :
declaration
declaration-list declaration
statement-list :
statement
statement-list statement
expression-statement :
expression $_{opt}$ **;**
iteration-statement :
while (*expression* **)** *statement*
do *statement* **while (** *expression* **);**
for (*expression* $_{opt}$ **;** *expression* $_{opt}$ **;** *expression* $_{opt}$ **)** *statement*
selection-statement :
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **)** *statement*
labeled-statement :
identifier **:** *statement*

case *constant-expression* **:** *statement*

default : *statement*

try-except-statement : /* Microsoft Specific */

__try *compound-statement*

__except (*expression* **)** *compound-statement*

try-finally-statement : /* Microsoft Specific */

__try *compound-statement*

__finally *compound-statement*

4.文法识别

没想到这一系列文件能得到 csdn 和大家的这么看好,首先要感谢大家的赏识和 csdn 的推荐.那么我就更没有理由不写好这下面的几篇文章了.本来我的计划是简单把 lex 和 yacc 介绍完后 就直接进入编译器的构造的技术细节问题讨论,但是最近看了一些国外经典教材后,发现文法的识别问题在编译原理和技术中是个绝不能忽视的问题.即使现在有了 yacc 工具来帮助我来识别文法,但是有些时候还是需要我们自己来写简单的语法分析器.

1.什么是文法识别(语法分析)

首先要告诉大家的是,这里的文法识别是指的上下文无关的文法,也就是上一节我们一直在讨论的那些 BNF 式.

比如说,我写了一句

if (a>6+5) printf(“OK!”); else printf(“No!”);

那么它匹配的文法也就是

if-stmt -> ***if*** *expr stmt*

| ***if*** *expr stmt* ***else*** *stmt*

我们通常要为一个程序语言写出很多 BNF 式的文法,怎么知道这句话是匹配的哪个文法,这就是语法分析器(或者叫文法分析器要做的工作).知道了是那句文法后,我们才能对这句话做出正确的解释,所以文法识别是个不可忽视的工作.下面我来看看我们常使用的文法识别的算法.

2.自顶向下的算法(LL 算法)

自顶向下的语法分析算法是十分简单的.自顶向下的算法也叫 LL 算法.LL(k)就是向前预测 k 个符号的自顶向下的算法.不过无论是我们国内的编译教程还是国外的经典教程都是只讨论了 LL(1)算法.因为一般的程序语言,只使用 LL(1)算法就已经足够了.这里我们同样也只是讨论 LL(1)算法.

其中有种特殊的算法叫做递归下降的算法,在 C 语言中,由于函数本身是可以递归的,所以实现这种算法就只需要写简单的几个函数的递归过程就是了.

为什么叫自顶向下呢?因为在分析过程中,我们是从语法树的树顶逐步向树底分析的,所以叫自顶向下的算法.

为了方便说明自顶向下算法的简单性,我们来看一下<<Compilers Principles,Techniques,and Tools>>中的一个例子.(本系列文章经常要引用国外经典著作的范例,希望大家不要告我抄袭,我实在找不到比大师的范例更经典的范例了)

例 4.1

考虑一个 Pascal 中定义变量的文法.

特别说明,这里的 **dotdot** 表示”..”

type -> *simple* | **id** | **array** [*simple*] **of** *type*

simple -> **integer** | **char** | **num dotdot num**

在为 **array[num dotdot num] of integer** 构造一个分析数的时候,该算法就是从根结点开始.

下面我们通过其中主要的三个步骤来看看算法的实现原理.

第一步分析:

首先分析的是输入的字符串第一个串”array”,判断出它属于 type 的 **First** 集合.所以在图中的分析树部分,我们的当前分析就是树根结点 type.(图中标上箭头,就表示是当前正在分析的部分).
这里遇到一个新名词:**First** 集合.在大学里的编译课程肯定是讲过 First 集合的吧.不过我还是要在这里重复一次了.

名词解释 **First** 集合:

在对文法产生式进行判断的时候,每个产生式都是由好几个终结符和非终结符构成.比如
本例中的文法
type -> *simple*
| **id**
| **array** [*simple*] **of** *type*
simple -> **integer**
| **char**
| **num dotdot num**

判断 type 的产生式的时候,如果我们把每个产生式里面的 simple,id,array, [,simple ,] , of , type 这些终结符和非终结符都进行判断的话,那么就会涉及到”试验和错误”的问题.当一个文法产生式分析到最后,发现并不匹配,就必然会产生回溯的问 题,就要回到头,从新开始对第二个产生式逐步进行判断分析.我们知道,回溯的算法效率肯定是十分低效率的.但是实际上我们完全可以避免这种回溯算法,而完 成同样工作的文法分析.这就产生了计算 **First** 集合的理论和以及后面的**左提公因式**的问题.

First 集合简单地说,就是一个非终结符的最开头的字符串(终结符号)的集合.比如说.
First(simple) = { **integer, char, num** }
First(type) = First(simple) U { **id**, **array** }
这里的 type 的一个产生式中有个 simple 非终结符在其开头,那么 simple 的开头字符串同时也可以是 simple,所以 First(simple)也是 First(type)的一部分.
为什么我们只计算每个非终结符的最开头的终结符? 因为我们这里是考虑的 LL(1)算法,LL(1)算法只向前预测一个字符号,所以我们只考虑一个 First 集合就可以判断出是哪个文法产生式了.

这里听起来似乎有些不太可能,一个产生式有那么千百万化,如果单单只看第一个非终结符号,如果就能说明一个输入串到底是哪个产生式呢? 如果有两个产生式的最开头一样怎么办,比如像 if 语句,那怎么办? 但其实我们几乎所有的程序语言的文法都可以通过 LL(1)来分析出来.原因是我们可以通过**左提公因式**来把最开头的相同的产生式的公共终结符号提取出来,留下两个子产生式,而他们的最开头的非终结符号不相同.

左提公因式:

例 4.2

考虑文法

A -> **ab**

|**ac**

这里,A 的两个产生式中最开头的终结符号都是’**a**’,那么就无法通过这个最开头的终结符号来判断一个输入串到底该是哪个产生式了.那么我们可以修改文法成

A -> **a***A*’

A’ -> **b** | **c**

这样一来,一个文法变成两个,但是无论 A 还是 A’,它们的每个产生式的 **First 集合都是不相交的**.所以,他们能够只通过最开头的终结符号来判断是哪个产生式.

这个变化过程有点想我们的代数里面的 ab + ac = a(b+c),所以叫它左提公因式.

这只是个简单的左提公因式的例子,实际当中还会遇到一些复杂的问题.但是无论是哪个编译教材,都会给出针对一切文法的左提公因式的算法.同样,计算 First 集合的算法也在教材中详细讲解了.我就不在这里再描述了.

第二步分析:

经过第一步的考察输入串中的第一个串为”array”属于非终结符号 type 第三个产生式的 First 集合,那么就可以确定这个串确实为 type 文法第三个产生式的串.所以在第二步中,展开出 type 的第三个产生式出来. *type* -> **array [** *simple* **] of integer**

那么接下来就是继续分析构造出来的 type -> array[simple] of integer 产生式中的每个结点.

所以箭头又放到了分析树中 type 的第一个孩结点 **array** 上.因为 array 是终结符号,如果它和输入中的当前箭头所指的终结符号相同,那么箭头都往下移动一结点到’**[**‘符号.同样地,由于分析树中的’**[**‘是终结符号,那么只要看输入中的串是否是’**[**‘就可以了.如果是,那么继续往下分析.分析到分析数中的 *simple* 的时候,由于 simple 是非终结符号,那么就需要考虑 simple 的产生式了.

第三步分析:

在第二步中,分析到分析数中的 simple 子结点的时候,由于 simple 是非终结符号,那么就需要考虑 simple 的产生式.simple 一共有三个产生式.通过输入串当前的串是”**num**”,是属于 simple 产生式中第 3 个产生式的 First 集合,所以 simple 在分析数中就按第三个产生式 *simple* -> **num dotdot num** 来展开.那么分析箭头同样,也自动移动到 simple 的第一个子结点 **num** 上继续分析.

总体说来,这中自顶向下的分析原理就基本上是上面的过程.通过计算产生式的 First 集合,来逐步产生非终结符的产生式.最后的分析树都会划归到终结符来进行判断(非终结符号是无法进行直接判断的,一定要展开过后才行).

看了原理,我们再看实现的伪代码.代码很简单.

```
void match(char token)
{
    if lookahead == token)
          lookahead = token;
    else
     error(0);
}

void type()
{
    if( lookahead == integer || lookeahead == char || lookahead == num)
       simple();
    else if( lookahead == id)
       match(id);
    else if( lookahead == array)
    {
       match(array); match(')'); simple(); match(')'); match(of); type();
    }
    else
       error(0);
}

void simple()
{
    if( lookahead == integar) match(integer);
    else if( lookahead == char) match(char);
    else if( lookahead == num)
    {
       match(num); match(dotdot); match(num);
    }
    else
       error(0);
}
```

注意:这里的代码都是纯的语法分析代码,实际执行过程中并没有什么用处,但是我们构造语法树 parse-tree 的代码就是镶嵌在这些纯的语法分析代码中.

5.实用 javacc

前言

本系列的文章的宗旨是让大家能够写出自己的编译器,解释器或者脚本引擎,所以每到理论介绍到一个程度后,我都会来讨论实践问题.理论方面,编译原理的教材已经是够多了,而实践的问题却很少讨论.

前几节文章只讨论到了词法分析和 LL 文法分析,关键的 LR 文法分析这里却还没有讲,我们先不要管复杂的 LR 文法和算法,让我们使用 LL 算法来实际做一些东西后再说.本文将介绍一个在 JAVA 上广泛使用的 LL 算法分析工具 Javacc.(这是我唯一能找到的使用 LL 算法的语法分析器构造工具).这一节的文章并非只针对 JAVA 开发者,如果你是 C/C++开发者,那么也请你来看看这个 JAVA 下的优秀工具,或许你将来也用得着它.

Lex 和 yacc 这两个工具是经典的词法分析和语法分析工具,但是它们都是基于 C 语言下面的工具,而使用 JAVA 的朋友们就用不上了.但是 JAVA 下已经有了 lex 和 yacc 的替代品 javacc(Java Compiler Compiler).同时 javacc 也是使用 LL 算法的工具,我们也可以实践一下前面学的 LL 算法.

首先声明我不是一个 JAVA 专家,我也是刚刚才接触 JAVA.Java 里面或许有很多类似 javacc 一样的工具,但是据我所知,javacc 还是最广泛,最标准的 JAVA 下的词法语法分析器.

Javacc 的获取

同 lex 和 yacc 一样,javacc 也是一个免费可以获取的通用工具,它可以在很多 JAVA 相关的工具下载网站下载,当然,javacc 所占的磁盘空间比起 lex 和 yacc 更大一些,里面有标准的文档和 examples.相对 lex 和 yacc 来说,javacc 做得更人性化,更容易一些.如果你实在找不到 javacc,还是可以联系我,我这里有.现在最新的就是 javacc 3.2 版本.

Javacc 的原理

Javacc 可以同时完成对 text 的词法分析和语法分析的工作,使用起来相当方便.同样,它和 lex 和 yacc 一样,先输入一个按照它规定的格式的文件,然后 javacc 根据你输入的文件来生成相应的词法分析于语法分析程序.同时,新版本的 Javacc 除了常规的词法分析和语法分析以外,还提供 JJTree 等 工具来帮助我们建立语法树.总之,Javacc 在很多地方做得都比 lex 和 yacc 要人性化,这个在后面的输入文件格式中也能体现出来.

Javacc 的输入文件

Javacc 的输入文件格式做得比较简单.每个非终结符产生式对应一个 Class 中的函数,函数中可以嵌入相应的识别出该终结符文法时候的处理代码(也叫动作).这个与 YACC 中是一致的.
Javacc 的输入文件中,有一系列的系统参数,比如其中 lookahead 可以设置成大于 1 的整数,那么就是说,它可以为我们生成 LL(k)算法(k>= 1),而不是简单的递归下降那样的 LL(1)算法了.要知道,LL(2)文法比起前面讨论的 LL(1)文法判断每个非终结符时候需要看前面两个记号而不是 一个,那么对于文法形式的限制就更少.不过 LL(2)的算法当然也比 LL(1)算法慢了不少.作为一般的计算机程序设计语言,LL(1)算法已经是足够 了.就算不是 LL(1)算法,我们也可以通过前面讲的左提公因式把它变成一个 LL(1)文法来处理.不过既然 javacc 都把 lookahead 选择做出 来了,那么在某些特定的情况下,我们可以直接调整一个 lookahead 的参数就可以,而不必纠正我们的文法.

下面我们来看看 Javacc 中自带的 example 中的例子.

例 5.1

这个例子可以在 javacc-3.2/doc/examples/SimpleExamples/Simple1.jj 看到

```
PARSER_BEGIN(Simple1)
public class Simple1 {
  public static void main(String args[]) throws ParseException {
    Simple1 parser = new Simple1(System.in);
    parser.Input();
  }
}
PARSER_END(Simple1)
void Input() :
{}
{
  MatchedBraces() ("\n"|"\r")* <EOF>
}
void MatchedBraces() :
{}
{
  "{" [ MatchedBraces() ] "}"
}
```

设置好 javacc 的 bin 目录后,在命令提示符下输入

javacc Simple1.jj

然后 javacc 就会为你生成下面几个 java 源代码文件

Simple1.java

Simple1TokenManager.java

Simple1Constants.java

SimpleCharStream.java

Token.java

TokenMgrError.java

其中 Simple1 就是你的语法分析器的对象,它的构造函数参数就是要分析的输入流,这里的是 System.in.

class Simple1 就定义在标记 PARSER_BEGIN(Simple1)

PARSER_END(Simple1)之间.

但是必须清楚的是,PARSER_BEGIN 和 PARSER_END 中的名字必须是词法分析器的名字(这里是 Simple1).

PARSER_END 下面的定义就是文法非终结符号的定义了.

Simple1 的文法基本就是:

Input -> MatchedBraces ("\n"|"\r") <EOF>*

MatchedBraces -> “{“ MatchedBraces “}”

从它的定义我们可以看到,每个非终结符号对于一个过程.
比如 Input 的过程

```
void Input() :
{}
{
  MatchedBraces() ("\n"|"\r")* <EOF>
}
```

在定义 void Input 后面记住需要加上一个冒号”:”,然后接下来是两个块{}的定义.
第一个{}中的代码是定义数据,初试化数据的代码.第二个{}中的部分就是真正定义 Input 的产生式了.
每个产生式之间用”|”符号连接.
注意: 这里的产生式并非需要严格 BNF 范式文法,它的文法既可以是 BNF,同时还可以是混合了正则表达式中的定义方法.比如上面的
Input -> MatchedBraces ("\n"|"\r") <EOF>*
中(“\n”|”\r”)* 就是个正则表达式,表示的是\n 或者\r 的 0 个到无限个的重复的记号.
而<EOF>是 javacc 系统定义的记号(TOKEN),表示文件结束符号.
除了<EOF>,无论是系统定义的 TOKEN,还是自定义的 TOKEN, 里面的 TOKEN 都是以<token’s name>的方式表示.

每个非终结符号(Input 和 MatchedBraces)都会在 javacc 生成的 Simple1.java 中形成 Class Simple1 的成员函数.当你在外部调用 Simple1 的 Input,那么语法分析器就会开始进行语法分析了.

例 5.2
在 javacc 提供的 example 里面没有.javacc 提供的 example 里面提供的例子中 SimpleExamples 过于简单,而其它例子又过于庞大.下面我以我们最常见的数学四则混合运算的文法来构造一个 javacc 的文法识别器.这个例子是我自己写的,十分简单,.其中还包括了文法识别同时嵌入的构建语法树 Parse-Tree 的代码.不过由于篇幅的原因,我并没有给出全部的代码,这里只给了 javacc 输入部分相关的代码.而 Parse-tree 就是一个普通的 4 叉树,3 个 child,1 个 next(平行结点),相信大家在学习数据结构的时候应该都是学过的.所以这里就省略过去了.

在大家看这些输入代码之前,我先给出它所使用的文法定义,好让大家有个清楚的框架.

```
Expression -> Term { Addop Term }
Addop -> "+" | "-"
Term -> Factor { Mulop Factor }
Mulop -> "*" | "/"
Factor -> ID | NUM | "(" Expression ")"
```

这里的文法可能和BNF范式有点不同. {}的意思就是0次到无限次重复, 它跟我们在学习正则表达式的时候的”*”符号相同, 所以，在 Javacc 中的文法表示的时候，{…}部分的就是用(…)*来表示.

为了让词法分析做得更简单,我们通常都不会在文法分析的时候,使用”(”,”)“等字符号串来表示终结符号,而需要转而使用 LPAREN, RPAREN 这样的整型符号来表示.

```
PARSER_BEGIN(Grammar)
public class Grammar implements NodeType {
  public ParseTreeNode GetParseTree(InputStream in) throws ParseException
  {
      Grammar parser =new Grammar(in);
      return parser.Expression();
  }

}
PARSER_END(Grammar)
SKIP :
{
  " " | "\t" | "\n" | "\r"
}
TOKEN :
{
  < ID: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )* >
|  < NUM: ( ["0"-"9"] )+ >
|  < PLUS:    "+" >
|  < MINUS:   "-" >
|  < TIMERS: "*" >
|  < OVER:    "/" >
|  < LPAREN: "(" >
|  < RPAREN: ")" >
}

ParseTreeNode Expression() :
{
       ParseTreeNode ParseTree = null;
       ParseTreeNode node;
}
{
 ( node=Simple_Expression()
 {
    if(ParseTree == null)
          ParseTree =node;
```

```
    else
    {
                ParseTreeNode t;
           t= ParseTree;
                while(t.next != null)
                 t=t.next;
               t.next = node;
    }
 }
   )*
  { return ParseTree;}
  <EOF>
}
ParseTreeNode Simple_Expression() :
{
        ParseTreeNode node;
        ParseTreeNode t;
        int op;
}
{
  node=Term(){}
  (
  op=addop() t=Term()
   {
                ParseTreeNode newNode = new ParseTreeNode();
                newNode.nodetype = op;
                newNode.child[0] = node;
                newNode.child[1] = t;
                switch(op)
                {
                        case PlusOP:
                        newNode.name = "Operator: +";
                        break;
                        case MinusOP:
                        newNode.name = "Operator: -";
                        break;
                }
                node = newNode;
        }
  )*
  { return node; }
}
int addop() : {}
{
```

```
        <PLUS> { return PlusOP; }
|   <MINUS> { return MinusOP; }
}
ParseTreeNode Term() :
{
        ParseTreeNode node;
        ParseTreeNode t;
        int op;
}
{
  node=Factor(){}
  (
  op=mulop() t=Factor()
   {
                ParseTreeNode newNode = new ParseTreeNode();
                newNode.nodetype = op;
                newNode.child[0] = node;
                newNode.child[1] = t;
                switch(op)
                {
                        case TimersOP:
                        newNode.name = "Operator: *";
                        break;
                        case OverOP:
                        newNode.name = "Operator: /";
                        break;
                }
                node = newNode;
        }
  )*
  {
      return node;
  }
}
int mulop() :{}
{
        <TIMERS> { return TimersOP; }
        | <OVER> { return OverOP;    }
}
ParseTreeNode Factor() :
{
        ParseTreeNode node;
        Token t;
}
```

```
{
  t=<ID>
   {
          node=new ParseTreeNode();
          node.nodetype= IDstmt;
          node.name = t.image;
          return node;
        }
  |
  t=<NUM>
  {
     node=new ParseTreeNode();
          node.nodetype= NUMstmt;
          node.name = t.image;
          node.value= Integer.parseInt(t.image);
          return node;
    }
  |
  <LPAREN> node=Simple_Expression() <RPAREN>
 {
        return node;
  }
}
```

其中 SKIP 中的定义就是在进行词法分析的同时,忽略掉的记号.TOKEN 中的,就是需要在做词法分析的时候,识别的词法记号.当然,这一切都是以正则表达式来表示的.
这个例子就有多个非终结符号,可以看出,我们需要为每个非终结符号写出一个过程.不同的非终结符号的识别过程中可以互相调用.

以 Simple_Expression()过程为例,它的产生式是 Expression -> Term { addop Term },而在 javacc 的输入文件格式是，它的识别是这样写的 node=Term(){} (op=addop() t=Term(){ … })* 前面说过，这里的"*"符号和正则表达式是一样的，就是 0 次到无限次的重复.那么 Simple_Expression 等于文法 Term Addop Term Addop Term Addop Term … 而 Addop 也就相当于 PLUS 和 MINUS 两个运算符号.这里我们在写 Expression 的文法的时候，同时还使用了赋值表达式，因为这个和 Yacc 不同的时候，Javacc 把文法识别完全地做到了函数过程中，那么如果我们要识别 Simple_Expression 的文法，就相当于按顺序识别 Term 和 Addop 两个文法,而识别那个文法，就相当于调用那两个非终结符的识别函数.正是这一点，我觉得 Javacc 的文法识别处理上就很接近程序的操作过程,我们不需要像 YACC 那样使用严格的文法表示格式，复杂的系统参数了.
关于 Yacc 的使用，其实比 Javacc 要复杂，还需要考虑到和词法分析器接口的问题，这个我会在以后细细讲到.

至于其它的文法操作解释我就不再多说了,如果要说,就是再写上十篇这样的文章也写不完.本文只能给读者们一个方向,至于深入的研究,还是请大家看 javacc 提供的官方文档资料.

最后

由于国外使用JAVA做项目的程序员比国内多,那么讨论JAVA技术的人员也比较多.可能来这里读我的文章的人都是C/C++程序员,但是关注其它领域同方向的技术也是可以让我们的知识领域更加宽广.关于JavaCC的讨论主要是在国际新闻组**comp.compilers.tools.javacc**如果大家在使用JavaCC做实际问题的时候遇到什么问题,不妨上去找找专家.

6.数学表达式

前言

文法分析中最重要算法是 LL 自顶向下和 LR 自底向上算法.前面几篇文章主要讲解的是 LL 算法的理论和一个 LL 算法的文法分析器 javacc.本文以 LL(1)算法中最简单的一种形式 递归下降算法来分析常规算法问题中的数学表达式问题.同时,本文也介绍手工构造 EBNF 文法的分析器代码普遍方法.希望本文的实践能对大家实现自己的语法 分析器带来帮助.

数学表达式问题

在学习算法的时候,四则混合运算的表达式处理是个很经典的算法问题.
比如这里有个数学表达式”122+2*(11-1)/(3-(2-0))”.我们需要根据这个字符串的描述,然后计算出其结果.

Input:
122+2*(11-1)/(3-(2-0))
Output:
142

四则混合运算中还需要考虑到括号,乘除号与加减号的优先运算问题,通常的解决办法就是使用堆栈.那种常规的算法和 LL 算法有异曲同工之处,更或者说,那么的算法其实是一样的.

传统数学表达式处理算法简介

这个传统算法其实不知不觉地使用 LL(1)算法的精髓.它就是主要依靠栈式的数据结构分别保存数和符号,然后根据运算符号的优先级别进行数学计算,并将结果保存在栈里面.
传统算法中使用了两个栈.一个是保存数值,暂时就叫值栈. 另一个是保存符号的,叫符号栈.我们规定一个记号#,来表示栈底.下面我们就来看看如何计算一个简单的表达式 11+2-8*(5-3).
为了显示整个计算过程,我们以下面这个栈的变化图来表示.

符号栈和值栈的变化是根据输入串来进行的.基本上栈的操作可以简单用下面几句话来说.
Start:

1. 如果当前输入串中得到的是数字,则直接压入值栈.然后转到 Start.
2. 如果当前输入串中得到的是符号,那么对符号进行判断.
   1)如果符号是’+’或者’-‘,则依次弹出符号栈的符号,计算栈中数值,直到弹出的符号不是*,/,+,-.
   2)如果符号是’*’或者’/’,则压入符号栈
   3)如果符号是’(‘,则直接压’(‘入符号栈
   4)如果符号是’)’,则依照符号栈的顺序弹出符号,计算栈中数值,把结果压入值栈,直到符号栈顶是’(‘,最后再弹出’(‘ .
   最后转到 Start.
3. 如果当前输入串得到的是 EOF(字符串结束符号),则计算栈中数值,知道符号栈没有符号.

语法分析数学表达式

或者可能你以前运用过自己的办法来解决过这个程序问题,不过下面我们将通过编译原理建立的一套文法分析理论,来十分精彩地解决这个算法问题.

首先是建立数学表达式的文法 EBNF.EBNF 文法可以更灵活地表示 BNF,是 BNF 范式文法的一种扩展.下面是上一篇 javacc 的介绍中使用到的计算表达式的文法.

Expression -> Term { Addop Term }

Addop -> "+" | "-"

Term -> Factor { Mulop Factor }

Mulop -> "*" | "/"

Factor -> ID | NUM | "(" Expression ")"

我们来看看如何根据这个 EBNF 文法实现一个递归下降的分析程序.大致上来说要分那么几步来实现.(注意,下面的几个步骤不光是针对本节的数学表达式问题,而是包含所有通常的递归下降文法分析器的实现)

语法分析实现

1. Step 建立词法分析

本系列文章开篇第一节就是讲的词法分析相关问题.因为词法分析是语法分析的前提,那么我们在实现递归下降算法的时候,同样应该把词法分析的实现考虑进去.

本文要处理只是个数学表达式的问题, 那么通过上面的文法,可以看到需要识别的词法无非就是 2 个 ID,NUM 和 4 个运算符号’+’’-‘‘*’’/’以及 2 个括号’(‘‘(‘.本文没有对词法 分析的自动机原理进行讲解,这部分内容应该在编译原理中讲得比较透彻.所谓自动机,就是按一定步骤识别每个字符的算法.可以用下面的几个图来表示 ID 和 NUM 的识别自动机(识别步骤或算法)

NUM:

基本算法就是,如果输入的字符是 digit(‘0’-‘9’),那么进入 check 循环,如果输入还是 digit,那么再跳回循环查看,如果输入是 other(不是’0’-‘9’),那么就直接 accept,接收这个串为 NUM 类型的 TOKEN.

ID:

同 NUM 一样,当输入的是 letter,那么进入 ID 的有限自动机.只是在进入 check 循环后,有两种可能都继续留在循环,那就是 digit 和 letter (‘a’-‘Z’).当输入既不是 digit,也不是 letter 的时候,就跳出 check 循环,进入 accept,把接收到的字符归结成 ID 类型的 TOKEN.

通过这个有限自动机的图示,我们就很容易写出词法分析程序.

不过在此之前,我们得写出识别 letter 和 digit 的代码.我们建立两个函数 IsLetter 和 IsDigit 来完成这个功能.

```
int IsLetter(char ch)
{
    if(ch >= 'A' && ch <= 'Z')
        return 1;
    if(ch >='a' && ch <='z')
        return 1;
    return 0;
}

int IsDigit(char ch)
{
    if(ch >= '0' && ch <='9')
        return 1;
    return 0;
}
```

有个这两个辅助函数,那么接下来,我们就直接写 gettoken 词法分析函数,它的功能就是从输入中分析,得到一个一个的 token.我们首先定义 token 的类型.

```
#define ID   1
#define NUM 2
#define PLUS    3 //   '+'
#define MINUS   4 //   '-'
#define TIMERS 5 //   '*'
#define OVER    6 //   '/'
#define LPAREN 7 //   '('
#define RPAREN 8 //   ')'
#define ERROR 255
```

上面注释已经说符号 token 代表的意思,我也不再多说.不过需要注意的是,这里我们定义了个 ERROR 的常量,但是我们这里并没有 ERROR 的 token,它只是为我们后面处理结果时候的一个错误处理信息的定义.

```
char token[10];
char *nextchar;
const char g_strCalculate[]="122+2*(11-1)/(3-(2-0))";
```

我们需要定义 token 记号和一个指到输入串的指针.token 记录的就是当前 gettoken()得到的 token 的 text(字符串).nextchar 是当前指到输入串的指针.最后,我们随便定义一个要分析的数学表达式的输入串 g_strCalculate.

```
int gettoken()
{
    char *ptoken =token;
    while(*nextchar == ' ' || *nextchar=='\n' || *nextchar=='\t')
```

```
        nextchar++;
    switch(*nextchar)
    {
    case '+': nextchar++; return PLUS;
    case '-': nextchar++; return MINUS;
    case '*': nextchar++; return TIMERS;
    case '/': nextchar++; return OVER;
    case '(': nextchar++; return LPAREN;
    case ')': nextchar++; return RPAREN;
    default: break;
    }
    // ID 的词法识别分析
    if(IsLetter(*nextchar))
    {
        while(IsLetter(*nextchar) || IsDigit(*nextchar))
        {
                *ptoken = *nextchar;
                nextchar++;
                ptoken++;
        }
        *ptoken ='\0';
        printf("gettoken: token = %s\n",token);
        return ID;
      }
      // NUM 的词法识别分析
    if(IsDigit(*nextchar))
    {
        while(IsDigit(*nextchar))
        {
              *ptoken = *nextchar;
              nextchar++;
              ptoken++;
        }
        *ptoken ='\0';
        printf("gettoken: token = %s\n",token);
        return NUM;
    }
    return ERROR;
}
```

代码很简单，我没有写多少任何注释.函数中,首先使用了 char *ptoken 记录 token 的首地址,它为后面的字符串复制(构造 token)所用.同时,在处理代码的第一部分是过滤掉空格,制表符和换行符.然后是 计算符号的词法分析.计算符号就是一个固定的字符号,所以它的识别很简单,直接用 switch 来判断*nextchar.而后面的 ID,NUM 的识别就是 完全按照前面的有限自动机表示图表来进行编写的.以 ID 的图表来说,ID 的自动机首先是要识别出第一个字符是 letter,

那么我就写了第一行 if(IsLetter(*nextchar)),如果满足,则进入 check 循环,也就是 while(IsLetter(*nextchar) || IsDigit(*nextchar))循环.循环中我们记录了*nextchar 到 token 符号中.最后跳出 check 循环,进入 accept,在代码中 return ID.对于 NUM 的词法识别也是如此的,我就不多说了.

2. 根据 EBNF 文法建立文法识别函数

首先看到第一条非终结产生式

Expression -> Term { Addop Term }

Expression 也是我们总的输入结果函数.我们先定义函数 int Expression(),其返回值就是我们要处理的表达式的值.右边的产生式中,第一个是 Term,我们就直接调用 Term 函数完成.然后是 0 到无限次的 Addop Term,那么用一个循环即可.文法中使用非终结符号 Addop.程序代码中我没有特别为此非终结符号创建函数.我们直接在代码以'+' || '-'代替 Addop. 代码如下.

```
int expression()
{
    int temp = term(); // 对应文法中的第一个 Term
       int tokentype;
    while(*nextchar == '+' || *nextchar == '-') // 对应文法中的{ Addop Term }
    {
        tokentype = gettoken();
        switch(tokentype)
        {
        case PLUS:
                temp +=term();
                break;
        case MINUS:
                temp -=term();
                break;
        default:
                break;
        }
    }
    return temp;
}
```

然后是第二条文法.同样,我也没有特别为 Mulop 特别写一个文法函数,而是直接以'*' || '\'代替.

Term -> Factor { Mulop Factor }

同理,建立如下函数

```
int term()
{
    int temp;
    int tokentype;
    temp = factor(); // 对应文法中的 Factor
    while(*nextchar == '*' || *nextchar == '\\') // 对应文法中的 {Mulop Factor}
    {
```

```
        tokentype =gettoken();
        switch(tokentype)
        {
        case TIMERS:
          temp *= factor();
          break;
        case OVER:
          temp /= factor();
          break;
        default:
          break;
        }
    }
    return temp;
}
```

最后是 Factor 文法 Factor -> ID | NUM | "(" Expression ")"

这个文法涉及到文法中的产生式的选择.由 LL(1)文法的理论我们可以知道,这完全可以是通过 ID,NUM, “(“Expression”)”三个产生式的第一个终结符的不同来判断的.ID 的第一个字符号肯定是 letter.而 NUM 第一个字符号肯定是 digit. "(" Expression ")"第一个字符号肯定是”(“.而 ID,NUM 的判断我们已经在词法分析的时候做好了(int gettoken()函数中).下面列出 Factor 文法函数的代码.

```
int factor()
{
    int number;
    switch(gettoken())
    {
    case ID: break;
    case NUM:
        number = atoi(token);
        break;
    case LPAREN:
        number = expression();
        if(gettoken() != RPAREN)
         printf("lost ')' in the expression \n");
        break;
    default:
        break;
    }
    return number;
}
```

好了,把上面出现的函数都写到程序文件中,加上个 main 函数,就可以编译运行了.

```
int main(int argc, char *argv[])
{
```

```
  nextchar = g_strCalculate;
  printf("result = %d\n",expression());
  system("PAUSE");
  return 0;
}
```

整个数学表达式的源程序大家可以在这里下载.
http://member.netease.com/~qinj/tangl_99/my doc/calculate/main.c

3. 总结

从上面三个 EBNF 文法实现我们可以容易得出一些机械性规律出来.

1. 对于 EBNF 文法中的非终结符都可以写成了一个单独的文法函数出来,比如前面 Expression(),Term(),Factor().
2. 文法中的产生式的选择可以根据第一个字符号来进行识别,这就是 LL(1)算法中提到的 First 集合.比如上面的 Factor 是直接通过 gettoken 得到下一个 token 的类型,然后根据类型的不同的 token 来 switch 处理不同产生式的代码.
3. 文法中的{} (0 到无限次循环),比如{Addop Term},{ Mulop Factor}可以直接通过循环语句完成.不过循环的条件还是需要判断下一 token 是不是 Addop 或者 Mulop 的 First 集合的元素.比如上面的 代码中,Addop 就是’+’和’-‘,Mulop 无非就是’*’和’/’,所以判断很容易,直接通过*nextchar 也可以判断.如果下一个 token 不是 Addop 或者 Mulop 的 First 集合元素,那么就应该跳出循环.返回文法函数.

虽然 EBNF 文法构造递归下降文法分析器代码是如此的简单,但是正如<<编译原理及实践>>书上提到的,它有它的特殊性.很多时候,仅仅是把 BNF 文法转换成 EBNF 文法本身就是一件十分困难的事情.这就需要我们前面提到的 LL(1)文法的消除左递归和提取左因式的问题.

与传统算法的比较

当我们明白了 EBNF 文法和递归下降的分析后,构造的数学表达式处理代码比传统算法要简单,要容易.原因前面也提到过,首先种东西的 EBNF 文法十分容易写,其次,从 EBNF 文法到代码的创建也十分机械化,也十分容易,最后,递归下降算法没有接触到栈的操作,利用了程序语言本身的递归本身特性,让程序语言去处理栈的操作(并不 是说递归下降的算法一点都没有接触到栈,可以说数据结构中递归处理就等于栈处理).
递归下降的算法也和容易扩展表达式的计算操作.比如说,我们想把函数(如 sin,cos)加进去,而且是加到所有运算的最高级别.那么我们修改 Factor 文法即可

Factor -> ID | NUM | "(" Expression ")"
| “sin”” (“ Expression “)”
| “cos””)” Expression “)”

至于代码的实现,我们也只需要在 int Factor 函数中的 switch 中增加几个 case 语句就可以了.