



Chinamming的专栏

三维图形, COM技术, VTK重建, CAD/CAE二次开发

目录视图

摘要视图

RSS 订阅

个人资料



Chinamming

访问: 381738次

积分: 5497

等级: 5

排名: 第3664名

原创: 7篇 转载: 315篇

译文: 0篇 评论: 49条

文章分类

[+原创]三维算法实现 (5)

[+原创]VTK三维重建 (8)

算法与数据结构 (19)

编译器编译原理 (24)

跨平台界面开发 (16)

数据库编程/SQL (21)

Lua/Awk/Python (59)

OpenGL/DirectX (1)

COM/COM+/ATL (33)

Win32/MFC (33)

OGRE3D (27)

三维图形引擎 (28)

三维图形建模 (10)

三维网格剖分 (10)

CAD二次开发 (7)

开源软件探索 (28)

阅读排行

VTK与ITK的详细安装指南

(10368)

专注网格剖分 - TetGen, N

(6798)

VTK三维重建(1)-使用VTI (6590)

SQLite C++操作类 (6541)

正则表达式DFA构造方法 (5234)

基于VTK的任意平面切割 (5155)

有限元网格生成程序及软 (4937)

语法分析——Bison介绍! (4646)

常见3DS Max格式概述 (4577)

使用C语言实现二维,三维 (4501)

聚焦行业最佳实践, BDTC 2016完整议程公布 微信小程序实战项目——点餐系统 程序员11月书讯, 评论得书啦 Get IT技能 知识库, 50个领域一键直达

基于Lex 和 Yacc 的 C 语言编译器

2013-11-26 23:07 3117人阅读 评论(0) 收藏 举报

分类: 编译器编译原理 (23) ▾

最近由于项目需要, 看了点关于编译原理和编译器等方面的资料, 特别是词法分析和语法分析部分, 现做一下小结。

一、编译器及其工作流程

编译器, 是将便于人编写, 阅读, 维护的高级计算机语言翻译为计算机能识别, 运行的低级机器语言的程序。编译器将源程序 (Source program) 作为输入, 翻译产生使用目标语言 (Target language) 的等价程序。源程序一般为高级语言 (High-level language), 如Pascal, C++等, 而目标语言则是汇编语言或目标机器的目标代码 (Object code), 有时也称作机器代码 (Machine code)。

一个现代编译器的主要工作流程如下:

源程序 (source code) →预处理器 (preprocessor) →编译器 (compiler) →汇编程序 (assembler) →目标程序 (object code) →连接器 (链接器, Linker) →可执行程序 (executables)

而编译器各阶段的主要工作包括:

1. 词法分析

词法分析器根据词法规则识别出源程序中的各个记号 (token), 每个记号代表一类单词 (lexeme)。源程序中常见的记号可以归为几大类: 关键字、标识符、字面量和特殊符号。词法分析器的输入是源程序, 输出是识别的记号流。词法分析器的任务是把源文件的字符流转换成记号流。本质上它查看连续的字符然后把它们识别为“单词”。

2. 语法分析

语法分析器根据语法规则识别出记号流中的结构 (短语、句子), 并构造一棵能够正确反映该结构的语法树。

3. 语义分析

语义分析器根据语义规则对语法树中的语法单元进行静态语义检查, 如果类型检查和转换等, 其目的在于保证语法正确的结构在语义上也是合法的。

4. 中间代码生成

中间代码生成器根据语义分析器的输出生成中间代码。中间代码可以有若干种形式, 它们的共同特征是与具体机器无关。最常用的一种中间代码是三地址码, 它的一种实现方式是四元式。三地址码的优点是便于阅读、便于优化。

5. 中间代码优化

优化是编译器的一个重要组成部分, 由于编译器将源程序翻译成中间代码的工作是机械的、按固定模式进行的, 因此, 生成的中间代码往往在时间和空间上有很大浪费。当需要生成高效目标代码时, 就必须进行优化。

6. 目标代码生成

目标代码生成是编译器的最后一个阶段。在生成目标代码时要考虑以下几个问题: 计算机的系统结构、指令系统、寄存器的分配以及内存的组织等。编译器生成的目标程序代码可以有多种形式: 汇编语言、可重定位二进制代码、内存形式。

7 符号表管理

符号表的作用是记录源程序中符号的必要信息，并加以合理组织，从而在编译器的各个阶段能对它们进行快速、准确的查找和操作。符号表中的某些内容甚至要保留到程序的运行阶段。

8 出错处理

用户编写的源程序中往往会有的一些错误，可分为静态错误和动态错误两类。所谓动态错误，是指源程序中的逻辑错误，它们发生在程序运行的时候，也被称作动态语义错误，如变量取值为零时作为除数，数组元素引用时下标出界等。静态错误又可分为语法错误和静态语义错误。语法错误是指有关语言结构上的错误，如单词拼写错、表达式中缺少操作数、begin和end不匹配等。静态语义错误是指分析源程序时可以发现的语言意义上的错误，如加法的两个操作数中一个是整型变量名，而另一个是数组名等。

二、lex 和 yacc 简单入门

Lex(Lexical Analyzer) 词法分析生成器, **Yacc(Yet Another Compiler Compiler)**

编译器代码生成器)是Unix下十分重要的词法分析，语法分析的工具。经常用于语言分析，公式编译等广泛领域。遗憾的是网上中文资料介绍不是过于简单，就是跳跃太大，入门参考意义并不大。本文通过循序渐进的例子，从0开始了解掌握Lex和Yacc的用法。

<本系列文章的地址: http://blog.csdn.net/liwei_cmg/category/207528.aspx>

1.Lex(Lexical Analyzer) 初步示例

先看简单的例子(注：本文所有实例皆在RedHat Linux下完成):

一个简单的Lex文件 exfirst.l 内容：

```
%{
#include "stdio.h"
%
%%
[ln]      ;
[0-9]+     printf("Int   : %s\n",yytext);
[0-9]*.[0-9]+   printf("Float  : %s\n",yytext);
[a-zA-Z][a-zA-Z0-9]* printf("Var   : %s\n",yytext);
[+|-|*|\%]   printf("Op    : %s\n",yytext);
.          printf("Unknown : %c\n",yytext[0]);
%%
```

在命令行下执行命令flex解析，会自动生成lex.yy.c文件：

[root@localhost liweitest]flex exfirst.l

进行编译生成parser可执行程序：

[root@localhost liweitest]cc -o parser lex.yy.c -l

[注意：如果不加-l链结选项，cc编译时会出现以下错误，后面会进一步说明。]

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/http://www.cnblogs.com/..../crt1.o(.text+0x18): In function `__start':
./sysdeps/i386/elf/start.S:77: undefined reference to `main'
/tmp/cciACkbX.o(.text+0x37b): In function `yylex':
: undefined reference to `yywrap'
/tmp/cciACkbX.o(.text+0xabd): In function `input':
: undefined reference to `yywrap'
collect2: ld returned 1 exit status
```

创建待解析的文件 file.txt:

```
title
i=1+3.9;
a3=909/6
```

评论排行

SQLite C++操作类	(10)
VTK与ITK的详细安装指南	(5)
基于VTK的任意平面切割	(4)
专注网格剖分 - TetGen,...	(3)
语法分析——Bison介绍!	(3)
三维图形数据格式 STL的...	(2)
VTK三维重建(1)-使用VTI	(2)
通过CAA在CATIA中创建	(2)
有限元网格生成程序及软...	(2)
使用C语言实现二维,三维	(2)

最新评论

语法分析——Bison介绍以及Flex
帆帆love: 这人写了挺多 可是永远都是一点点莫名其妙的代码就完事。

通过CAA在CATIA中创建自己的牛逼顿: 楼主你好，请问用勾选框addin调用dialog后，如果关闭dialog，怎样同时取消勾选框addi...

SQLite C++操作类
jjcwxw: 请问插入中文乱码是什么

SQLite C++操作类
cannycsy: if(sqlite3_exec(m_db,szSql,pCallBack,NULL,&err...)

有限元网格生成程序及软件
WHJ_HJW_WHJ: 有联系方式吗？交流一下Tetgen

基于VTK的任意平面切割
inter_peng: 如果可以给一个运行后的效果图就好了。

SQLite C++操作类
firejq: 学习了

SQLite C++操作类
Learning1985: 太棒了。正在学习中。

我们为什么需要awk?
sinat_34584629: 楼主写得不错，也可以使用
<http://www.itshouce.com.cn/linux/linux/>

三维体数据分割算法及实现
poiyuhng: 这个代码可以跑么？为什么我调用OctreeSplitVolumeData会出现栈溢出

文章存档

2014年02月	(17)
2013年12月	(97)
2013年11月	(183)
2013年07月	(5)
2013年06月	(6)

展开



```
bcd=4%9-333
```

通过已生成的可执行程序，进行文件解析。

```
[root@localhost liweitest]# ./parser < file.txt
Var    : title
Var    : i
Unknown := 
Int    : 1
Op    : +
Float  : 3.9
Unknown : ;
Var    : a3
Unknown := 
Int    : 909
Op    : /
Int    : 6
Var    : bcd
Unknown := 
Int    : 4
Op    : %
Int    : 9
Op    : -
Int    : 333
```

到此Lex用法会有个直观的了解：

1. 定义Lex描述文件
2. 通过lex, flex工具解析成lex.yy.c文件
3. 使用cc编译lex.yy.c生成可执行程序

再来看一个比较完整的Lex描述文件 exsec.l：

```
%{
#include "stdio.h"
int linenum;
%}
%%
title      showtitle();
[\n]        linenum++;
[0-9]+      printf("Int   : %s\n",yytext);
[0-9]*.[0-9]+  printf("Float  : %s\n",yytext);
[a-zA-Z][a-zA-Z0-9]* printf("Var   : %s\n",yytext);
[+|-|*|\%]    printf("Op    : %s\n",yytext);
.           printf("Unknown : %c\n",yytext[0]);
%%
showtitle()
{
printf("----- Lex Example -----\\n");
}

int main()
{
linenum=0;
yylex(); /* 进行分析 */
printf("\\nLine Count: %d\\n",linenum);
```

```

return 0;
}

int yywrap()
{
    return 1;
}

进行解析编译:
[root@localhost liweitest]flex exsec.l
[root@localhost liweitest]cc -o parser lex.yy.c
[root@localhost liweitest]./parser < file.txt

----- Lex Example -----
Var   : i
Unknown :=

Int   : 1
Op    : +
Float  : 3.9
Unknown :
Var   : a3
Unknown :=

Int   : 909
Op    : /
Int   : 6
Var   : bcd
Unknown :=

Int   : 4
Op    : %
Int   : 9
Op    : -
Int   : 333

Line Count: 4

```

这里就没有加-ll选项，但是可以编译通过。下面开始着重整理下Lex描述文件.l。

2.Lex(Lexical Analyzar) 描述文件的结构介绍

Lex工具是一种词法分析程序生成器，它可以根据词法规则说明书的要求来生成单词识别程序，由该程序识别出输入文本中的各个单词。一般可以分为<定义部分><规则部分><用户子程序部分>。其中规则部分是必须的，定义和用户子程序部分是任选的。

(1) 定义部分

定义部分起始于 %{ 符号，终止于 %} 符号，其间可以是包括include语句、声明语句在内的C语句。这部分跟普通C程序开头没什么区别。

```
%{
#include "stdio.h"
int linenum;
%}
```

(2) 规则部分

规则部分起始于"%%"符号，终止于"%%"符号，其间则是词法规则。词法规则由模式和动作两部分组成。模式部分可以由任意的正则表达式组成，动作部分是由C语言语句组成，这些语句用来对所匹配的模式进行相应处理。需要注意的是，lex将识别出来的单词存放在yytext[]字符数据中，因此该数组的内容就代表了所识别出来的单词的内容。类似yytext这些预定义的变量函数会随着后面内容展开一一介绍。动作部分如果有多

行执行语句，也可以用{}括起来。

```
%%
title      showtitle();
[ln]       linenum++;
[0-9]+    printf("Int   : %s\n",yytext);
[0-9]*.[0-9]+  printf("Float  : %s\n",yytext);
[a-zA-Z][a-zA-Z0-9]* printf("Var   : %s\n",yytext);
[+-\^*\%]   printf("Op    : %s\n",yytext);
.          printf("Unknown : %c\n",yytext[0]);
%%
```

A. 规则部分的正则表达式

规则部分是Lex描述文件中最为复杂的一部分，下面列出一些模式部分的正则表达式字符含义：

A-Z, 0-9, a-z 构成模式部分的字符和数字。

- 指定范围。例如：a-z 指从 a 到 z 之间的所有字符。

**** 转义元字符。用来覆盖字符在此表达式中定义的特殊意义，只取字符的本身。

[] 表示一个字符集合。匹配括号内的任意字符。如果第一个字符是^那么它表示否定模式。例如: [abC] 匹配 a, b, 和C 的任何一个。

^ 表示否定。

***** 匹配0个或者多个上述模式。

+ 匹配1个或者多个上述模式。

? 匹配0个或1个上述模式。

\$ 作为模式的最后一个字符时匹配一行的结尾。

{} 表示一个模式可能出现的次数。例如: A{1,3} 表示 A 可能出现1次或3次。[a-z]{5} 表示长度为5的，由a-z组成的字符串。此外，还可以表示预定义的变量。

. 匹配任意字符，除了 \n。

() 将一系列常规表达式分组。如: {Letter}({Letter}|{Digit})*

| 表达式间的逻辑或。

"一些符号" 字符的字面含义。元字符具有。如: "*" 相当于 [*]。

/ 向前匹配。如果在匹配的模式中的"/"后跟有后续表达式，只匹配模版中"/"前面的部分。如：模式为 ABC/D 输入 ABCD，时ABC会匹配ABC/D，而D会匹配相应的模式。输入ABCE的话，ABCE就不会去匹配ABC/D。

B. 规则部分的优先级

规则部分具有优先级的概念，先举个简单的例子：

```
%{
#include "stdio.h"
```

```
%}
%%

[\n]      ;
A        {printf("ONE\n");}
AA       {printf("TWO\n");}
AAAA     {printf("THREE\n");}
%%
```

此时，如果输入内容：

```
[root@localhost liweitest]# cat file1.txt
AAAAAAA

[root@localhost liweitest]# ./parser < file1.txt
THREE
TWO
ONE
```

Lex分析词法时，是逐个字符进行读取，自上而下进行规则匹配的，读取到第一个A字符时，遍历后发现三个规则皆匹配成功，Lex会继续分析下去，读至第五个字符时，发现"AAAA"只有一个规则可用，即按行为进行处理，以此类推。可见Lex会选择最长的字符串匹配规则。

如果将规则

```
AAAA     {printf("THREE\n");}
改为
AAAAA    {printf("THREE\n");}
```

./parser < file1.txt 输出结果为：

```
THREE
TWO
```

再来一个特殊的例子：

```
%%
title      showtitle();
[a-zA-Z][a-zA-Z0-9]* printf("Var : %s\n",yytext);
%%
```

并输入title，Lex解析完后发现，仍然存在两个规则，这时Lex只会选择第一个规则，下面的则被忽略的。这里就体现了Lex的顺序优先级。把这个例子稍微改一下：

```
%%
[a-zA-Z][a-zA-Z0-9]* printf("Var : %s\n",yytext);
title      showtitle();
%%
```

Lex编译时会提示： warning, rule cannot be matched. 这时处理title字符时，匹配到第一个规则后，第二个规则就无效了。

再把刚才第一个例子修改下，加深印象！

```
%{
#include "stdio.h"
%
%%

[\n]      ;
A        {printf("ONE\n");}
AA       {printf("TWO\n");}
```

```
AAAA      {printf("THREE\n");}
AAAA      {printf("Cannot be executed!");}
```

`./parser < file1.txt` 显示效果是一样的，最后一项规则肯定是会忽略掉的。

C. 规则部分的使用变量

且看下面示例：

```
%{
#include "stdio.h"
int linenum;
%}
int      [0-9] +
float    [0-9]*.[0-9] +
%%%
{int}    printf("Int : %s\n",yytext);
{float}  printf("Float : %s\n",yytext);
.        printf("Unknown : %c\n",yytext[0]);
%%%
```

在`%`和`%%`之间，加入了一些类似变量的东西，注意是没有`;`的，这表示`int`, `float`分别代指特定的含义，在两个`%%`之间，可以通过`{int}{float}`进行直接引用，简化模式定义。

(3) 用户子程序部分

最后一个`%%`后面的内容是用户子程序部分，可以包含用C语言编写的子程序，而这些子程序可以用在前面的动作中，这样就可以达到简化编程的目的。这里需要注意的是，当编译时不带`-l`选项时，是必须加入`main`函数和`yywrap`(`yywrap`将下后面说明)。如：

```
...
%%%
showtitle()
{
printf("---- Lex Example ----\n");
}

int main()
{
linenum=0;
yylex(); /* 进行Lex分析 */
printf("\nLine Count: %d\n",linenum);
return 0;
}

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

3.Lex(Lexical Analyzaz) 一些的内部变量和函数

内部预定义变量：

```
yytext  char * 当前匹配的字符串
yyleng  int   当前匹配的字符串长度
yyin    FILE * lex当前的解析文件，默认为标准输出
yyout   FILE * lex解析后的输出文件，默认为标准输入
```

```
yylineno int    当前的行数信息
```

内部预定义宏:

```
ECHO #define ECHO fwrite(yytext, yylen, 1, yyout) 也是未匹配字符的
默认动作
```

内部预定义的函数:

```
int yylex(void) 调用Lex进行词法分析
int yywrap(void) 在文件(或输入)的末尾调用。如果函数的返回值是1, 就停止解
析。因此它可以用来解析多个文件。代码可以写在第三段, 这
样可以解析多个文件。方法是使用 yyin 文件指针指向不同的
文件, 直到所有的文件都被解析。最后, yywrap() 可以返回1
来表示解析的结束。
```

lex和**flex**都是解析**Lex**文件的工具, 用法相近, **flex**意为fast lexical analyzer generator。可以看成**lex**的升级版本。

相关更多内容就需要参考**flex**的man手册了, 十分详尽。

4.Lex理论

Lex使用正则表达式从输入代码中扫描和匹配字符串。每一个字符串会对应一个动作。通常动作返回一个标记给后面的剖析器使用, 代表被匹配的字符串。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。

我们以上文《Lex和Yacc应用方法(一).初识Lex》第一个例子详细说明:

exfirst.l

```
%{
#include "stdio.h"
%
%%
[\\n]      ;
[0-9]+     printf("Int : %s\\n",yytext);  B
[0-9]*.[0-9]+   printf("Float : %s\\n",yytext);  C
[a-zA-Z][a-zA-Z0-9]* printf("Var : %s\\n",yytext);  D
[+|-|*|\\%]    printf("Op : %s\\n",yytext);  E
.          printf("Unknown : %c\\n",yytext[0]); F
%
%
```

这里使用一相对简单的输入文件 file.txt

```
i=1.344+39;
bcd=4%9-333
```

我们假定,

Lex 系统创建一动态列表: 内容+规则+状态

Lex 状态: 1 接受 2 结束

接受表示该元素可以做为模式匹配

结束表示该元素已完成模式匹配

读入“r”

[查找元素]查找相邻且状态为1的元素，无元素，

[匹配规则]D,

[新增列表<元素1>并置数据](存在则覆盖)状态为1，规则为D，内容为“i”。

[操作顺序符] 1

读入“=”

[查找元素]查找相邻且状态为1的元素，“i=”寻找匹配规则，无规则

[置上一元素]<元素1>状态为2

[匹配规则]F,

[新增列表<元素2>并置数据](存在则覆盖)状态为1，规则为F，内容为“=”

[操作顺序符] 2

读入“1”，

[查找元素]查找相邻且状态为1的元素，“=1”寻找匹配规则，无规则

[置上一元素]<元素2>状态为2

[匹配规则]B,

[新增列表<元素3>并置数据](存在则覆盖)状态为1，规则为B，内容为“1”

[操作顺序符] 3

读入“.”

[查找元素]查找相邻且状态为1的元素，“1.”寻找匹配规则，无规则，但有潜在规则C

[匹配规则]F,

[新增列表<元素4>并置数据](存在则覆盖)状态为1，规则为F，内容为“.”

[置上一元素]<元素3>状态为1

[操作顺序符] 4

读入“3”

[查找元素]查找相邻且状态为1的元素，“1.3”寻找匹配规则，有规则

[置起始元素]状态为1，规则为C，内容为“1.3”

[操作顺序符] 3 组合元素的起始操作符

读入“4”

[查找元素]查找相邻且状态为1的元素，“1.34”寻找匹配规则，有规则

[置起始元素]状态为1，规则为C，内容为“1.34”

[操作顺序符] 3 组合元素的起始操作符

读入“4”

[查找元素]查找相邻且状态为1的元素，“1.344”寻找匹配规则，有规则

[置起始元素]状态为1，规则为C，内容为“1.344”

[操作顺序符] 3 组合元素的起始操作符

读入“+”

[查找元素]查找相邻且状态为1的元素，“1.344+”寻找匹配规则，无规则

[匹配规则]E,

[新增列表<元素4>并置数据](存在则覆盖)状态为1，规则为E，内容为“+”

[置上一元素]<元素3>状态为2

[操作顺序符] 4

...

最后解析结果为

内容	规则	状态
<元素1>	i	D
<元素2>	=	F
<元素3>	1,344	C
<元素4>	+	E

...

上面列出的**算法**是仅属于个人的分析，是相对直观且便于理解的，也可以参照这个算法用C语言模拟出lex的效果。不过真正的Lex算法肯定是更为复杂的理论体系，这个没有接触过，有兴趣可以参看相关资料。

5. 关于Lex的一些综述

Lex其实就是词法分析器，通过配置文件*.l，依据正则表达式逐字符去顺序解析文件，并动态更新内存的数据解析状态。不过Lex只有状态和状态转换能力。因为它没有堆栈，它不适合用于剖析外壳结构。而yacc增加了一个堆栈，并且能够轻易处理像括号这样的结构。Lex擅长于模式匹配，如果有更多的运算要求就需要yacc了。

6、yacc的BNF文件

个人认为lex理论比较容易理解的， yacc要复杂一些。

我们先从yacc的文法说起。 yacc文法采用BNF(Backus-Naur Form)的变量规则描述。BNF文法最初由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的语言。现代程序语言大多数结构能够用BNF来描述。

举个加减乘除例子来说明：

1+2/3+4*6-3

BNF文法：

优先级

```
E = num    规约a  0
E = E / E  规约b  1
E = E * E  规约c  1
E = E + E  规约d  2
E = E - E  规约e  2
```

这里像（E表达式）这样出现在左边的结构叫做非终结符(**nonterminal**)。像（num标识符）这样的结构叫终结符 (**terminal**，读了后面内容就会发现，其实是由lex返回的标记），它们只出现在右边。

我们将“1+2/3+4*6-3-2”逐个字符移进堆栈，如下所示：

```
.1+2/3+4*6-3
1  1.+2/3+4*6-3  移进
2  E.+2/3+4*6-3  规约a
3  E+.2/3+4*6-3  移进
4  E+2./3+4*6-3  移进
5  E+E./3+4*6-3  规约a
6  E+E/.3+4*6-3  移进
7  E+E/3.+4*6-3  移进
8  E+E/E.+4*6-3  规约a
9  E+E/E+.4*6-3  移进
10 E+E/E+4.*6-3  移进
11 E+E/E+E.*6-3  规约a
12 E+E/E+E*.6-3  移进
13 E+E/E+E*6.-3  移进
14 E+E/E+E*E.-3  规约a
15 E+E/E+E*E-.3  移进
16 E+E/E+E*E-3.  移进
17 E+E/E+E*E-E.  规约a
```

```

18 E+E+E*E-E.    规约b
19 E+E+E-E.    规约c
20 E+E-E.    规约d
21 E-E.    规约d
22 E.    规约e

```

我们在实际运算操作中是把一个表达式逐步简化成一个非终结符。称之为“自底向上”或者“移进归约”的分析法。

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`1`压入堆栈中。第2步对应规则a，把`1`转换成`E`。然后继续压入和归约，直到第5步。此时堆栈中剩下`E+E`，按照规则d，可以进行`E=E+E`的合并，然而输入信息并没有结束，这就产生了“移进-归约”冲突(shift-reduce conflict)。在yacc中产生这种冲突时，会继续移进。

在第17步，`E+E/E`，即可以采用`E+E`规则d，也可以采用`E/E`规则b，如果使用`E=E+E`规约，显然从算法角度是错误的，这就有了运算符的优先级概念。这种情况称为“归约-归约”冲突(reduce-reduce conflict)。此时yacc会采用第一条规则，即`E=E/E`。这个内容会在后面的实例做进一步深化。

7、十分典型的利用lex和yacc模拟的简单+*/计算器。

A.示例

最有效的方法是示例学习，这样首先给出全部示例文件。

lex文件: `lexya_a.l`

```

%{
#include <stdlib.h>
void yyerror(char *);
#include "lexya_a.tab.h"
%}
%%
[0-9]+ { yyval = atoi(yytext); return INTEGER; }
[-+/*\n]  return *yytext;
[\t]    /* 去除空格 */
.     yyerror("无效字符");
%%
int yywrap(void) {
return 1;
}

```

yacc文件: `lexya_a.y`

```

%{
#include <stdlib.h>
int yylex(void);
void yyerror(char *);
%}
%token INTEGER
%left '+' '-'
%left '*' '/'

```

```

%%

program:
program expr '\n' { printf("%d\n", $2); }
|
;

expr:
INTEGER { = $1; } | expr '*' expr { = $1 * $3; }
| expr '/' expr { = $1 / $3; } | expr '+' expr { = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
;
%%

void yyerror(char *s) {
printf("%s\n", s);
}

int main(void) {
yparse();
return 0;
}

进行编译:
bison -d lexya_a.y
lex lexya_a.l
cc -o parser lex.y.c lexya_a.tab.c -l

运行:
./parser

输入计算式, 回车会显示运算结果

```

如:

```

1+2*5+10/5
13
9+8/3
11
10+2-2/2-2*5
1

```

这里有两个文件lexya_a.y和lexya_a.l。lexya_a.y是yacc文件, bison -d lexya_a.y
编译后会产生lexya_a.tab.c lexya_a.tab.h。lex文件lexya_a.l中头声明已包括了
lexya_a.tab.h。这两个文件是典型的互相协作的示例。

B.分析

(1) 定义段和预定义标记部分

yacc文件定义与lex十分相似, 分别以%{ }% %% %%分界。

```

%{
#include <stdlib.h>
int yylex(void);
void yyerror(char *);
%}

```

这一段十分容易理解, 只是头文件一些引用说明。称为“定义”段。

```
%}
```

```
%token INTEGER
```

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%%
```

%}和%%这一段可以看作预定义标记部分。%token INTEGER 定义声明了一个标记。

当我们编译后，它会在lexya_a.tab.c中生成一个剖析器，同时会在lexya_a.tab.h

产生包含信息：

```
# define INTEGER 257
```

其中0-255之间的标记值约定为字符值，是系统保留的后定义的token。

lexya_a.tab.h其它部分是默认生成的，与token INTEGER无关。

```
# ifndef YYSTYPE
```

```
# define YYSTYPE int
```

```
# define YYSTYPE_IS_TRIVIAL 1
```

```
# endif
```

```
extern YYSTYPE yylval;
```

lex文件需要包含这个头文件，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yylval变量中保存着与返回的标记相对应的值。

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，并且记录了当前剖析状态。而内容栈是一个YYSTYPE类型的元素数组，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，把这个标记移入分析栈。同时，相应的yacc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的标记值是很容易的。

比如lex文件中下面这一段：

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

这是将把整数的值保存在yylval中，同时向yacc返回标记INTEGER。即内容栈存在了整数的值，对应的分析栈就为INTEGER标记了。yylval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。

lex文件还有一段：

```
[-+*/\n] return *yytext;
```

这里显然只是向yacc的分析栈返回运算符标记，系统保留的0-255此时便有了作用，内容栈为空。把“-”放在第一位是防止正则表达式发现类似a-z的歧义。

再看下面的：

```
%left '+' '-'
```

```
%left '*' '/'
```

%left 表示左结合，%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。

注：关于结合性，各运算符的结合性分为两种，即左结合性(自左至右)和右结合性(自右至左)。例如算术运算符的结合性是自左至右，即先左后右。如有表达式x-y+z则y应先与“-”号结合，执行x-y运算，然后再执行+z的运算。这种自左至右的结合方向就称为“左结合性”。而自右至左的结合方向称为“右结合性”。最典型的右结合性运算符是赋值运算符。如x=y=z,由于“=”的右结合性，应先执行y=z再执行x=(y=z)运算。

(2)规则部分

```
%%
program:
program expr '\n' { printf("%d\n", $2); }
|
;
expr:
INTEGER { = $1; } | expr '*' expr { = $1 * $3; }
| expr '/' expr { = $1 / $3; } | expr '+' expr { = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
;
%%
```

这个规则乍看起来的确有点晕，关键一点就是要理解yacc的递归解析方式。

`program`和`expr`是规则标记，但是作为一个整体描述表达式。

先看`expr`，可以由单个`INTEGER`值组成，也可以有多个`INTEGER`和运算符组合组成。

以表达式“1+4/2*3-0”为例，1 4 2 3 都是`expr`，就是`expr+expr/expr*expr-expr`说到底最后还是个`expr`。递归思想正好与之相反，逆推下去会发现`expr`这个规则标记能表示所有的数值运算表达式。

了解了`expr`后，再看`program`，首先`program`可以为空，也可以用单单的`expr`加下“\n”回车符组成，结合起来看`program`定义的就是多个表达式组成的文件内容。

回过头，创建如下文件`input`:

```
[root@localhost yacc]# cat input
1+5/5+4*5
3+9+2*10-9
2/2
3-9
```

运行则结果如下：

```
[root@localhost yacc]# ./parser < input
22
23
1
-6
```

粗略有了概念之后，再看lex如何执行相应的行为。

以 `expr: expr '+' expr { = $1 + $3; }`为例： 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，压入一个成员来缩小堆栈。在我们的代码中可以看到用相对地址访问内容栈中的值。如`$1`, `$2`, 这样都是yacc预定义可以直接使用的标记。“`$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，然后把得到的和压入堆栈中。这样，保持分析栈和内容栈中的内容依然同步。

而

`program:`

```
program expr '\n' { printf("%d\n", $2); }
```

说明每当一行表达式结束时，打印出第二个栈值，即`expr`的值，完成字符运算。

三、 使用Lex 和 Yacc 开发C编译器

(1) 从网站： 下载C语言的语法文件：

最新 The ANSI C grammar ([Yacc](#) and [Lex](#))

<http://www.quut.com/c/ANSI-C-grammar-l-1998.html>

<http://www.quut.com/c/ANSI-C-grammar-y-1998.html>

(2) 编译词法文件： > lex c.l

(3) 编译语法文件： > yacc -dv c.y说明： -d： 产生头文件y.tab.h， -v： 产生分析表y.output。针对else产生的移进规约冲突，采用了yacc的默认动作“移进”解决。

(4) 编译语法分析器：

> cc lex.yy.c y.tab.c -ll

(5) 测试： 编写测试程序test.c

```
#include "stdio.h"

int main(){
    int a = 0;
    for(; a < 10; a++){
        printf("hello from sun! ");
    }
}
```

运行： > ./a.out < test.c结果如下：

```
include "stdio.h"

int main(){
    int a = 0;
    for(; a < 10; a++){
        printf("hello from sun! ");
    }
}
```

没看到预期到效果。

存在一个移进/规约冲突。以bison的"-v"选项生成状态机描述文件（《Lex与Yacc》很好地描述了如何理解此文
件）。

[kenny@kenny ser-0.9.4]\$ bison -v c.y

cfg.y: conflicts: 1 shift/reduce

查看状态机描述文件c.output可知如下文法片段存在典型的“if-then-else”冲突：

```

stm:      cmd
| if_cmd
| LBRACE actions RBRACE
;
if_cmd:   IF exp stm
| IF exp stm ELSE stm
;

```

进来解决这些冲突,除非有其它的操作符优先级的指导。冲突存在的原因是由于语法本身有歧义:任一种简单的if语句嵌套的分析都是合法的.已经建立的惯例是通过将else从句依附到最里面的if语句来解决歧义;这就是Bison为什么选择移进而不是归约的原因。”

“Bison被设计成选择移以下这段话摘自Bison info手册:

“移进/规约”冲突问题解决方法:

有以下几个解决此冲突的办法(参考Bison info手册以及《Lex与Yacc》):

(1)改写if语句语法,优点是彻底,缺点是导致语法复杂化。

(2)为冲突的两规则指定优先级以隐藏这个你知道并理解的冲突,但是特别要注意不要隐藏其他任何冲突。

我按照办法2这样做:

在c.y序幕部分加如下两个的操作符:

```
%nonassoc LOWER_THAN_ELSE
```

```
%nonassoc ELSE
```

修改不带else分支的那个规则,使其优先级低于带else分支的那个规则:

```
selection_statement
```

```
: IF '(' expression ')' statement %prec LOWER_THAN_ELSE
```

```
| IF '(' expression ')' statement ELSE statement
```

```
| SWITCH '(' expression ')' statement
```

```
;
```

五、参考资料

1.编译器知识拾零

2.草木瓜 Lex和Yacc应用

3.使用Lex 和 Yacc 开发 C 语言编译器

顶
1
踩
0

[上一篇 浅谈SQLite——浅析Lemon](#)

[下一篇 20050620 GNU Bison 中文手册翻译完成](#)

我的同类文章

[编译器编译原理 \(23\)](#)

- 正则表达式引擎的构建——... 2013-12-06 阅读 3294
- 状态机之C++解析 2013-12-06 阅读 767
- Bison生成文件分析 2013-11-26 阅读 2414
- Yacc介绍与使用 2013-11-26 阅读 761
- GNU Flex与Bison结合使用 2013-11-26 阅读 4123
- 正则表达式引擎的构建——... 2013-12-06 阅读 1059
- 正则表达式DFA构造方法 2013-12-06 阅读 5236
- 语法分析——Bison介绍以... 2013-11-26 阅读 4647
- 在VC6.0/VC2008中高效地... 2013-11-26 阅读 891
- 20050620 GNU Bison 中文... 2013-11-26 阅读 1540

[更多文章](#)

参考知识库



.NET知识库

2203 关注 | 815 收录



Linux知识库

7499 关注 | 3407 收录



C语言知识库

5743 关注 | 3439 收录



软件测试知识库

2681 关注 | 310 收录



算法与数据结构知识库

10877 关注 | 2291 收录

猜你在找

[C语言系列之 数据结构栈的运用](#)[跟我一起写 Makefile](#)[Linux环境C语言编程基础](#)[MakeFile浅谈](#)[C语言及程序设计提高](#)[跟我一起写 Makefile----- 作者陈皓](#)[《C语言/C++学习指南》语法篇（从入门到精通）](#)[转载跟我一起写 Makefile](#)[ArcGIS for javascript 项目实战（环境监测系统）](#)[Makefile教程](#)

童装连帽卫衣 卡通小单...

¥99.00

特卖中式超薄四屉二门...

¥2400.00

淘 广告

[查看评论](#)

暂无评论

[发表评论](#)

用户名: lixianghai2010

评论内容:

[提交](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- [全部主题](#)
- [Hadoop](#)
- [AWS](#)
- [移动游戏](#)
- [Java](#)
- [Android](#)
- [iOS](#)
- [Swift](#)
- [智能硬件](#)
- [Docker](#)
- [OpenStack](#)
- [VPN](#)
- [Spark](#)
- [ERP](#)
- [IE10](#)
- [Eclipse](#)
- [CRM](#)
- [JavaScript](#)
- [数据库](#)
- [Ubuntu](#)
- [NFC](#)
- [WAP](#)
- [jQuery](#)
- [BI](#)
- [HTML5](#)
- [Spring](#)
- [Apache](#)
- [.NET](#)
- [API](#)
- [HTML](#)
- [SDK](#)
- [IIS](#)
- [Fedora](#)
- [XML](#)
- [LBS](#)
- [Unity](#)
- [Splashtop](#)
- [UML](#)
- [components](#)
- [Windows Mobile](#)
- [Rails](#)
- [QEMU](#)
- [KDE](#)
- [Cassandra](#)
- [CloudStack](#)
- [FTC](#)
- [coremail](#)
- [OPhone](#)
- [CouchBase](#)
- [云计算](#)
- [iOS6](#)
- [Rackspace](#)
- [Web App](#)
- [SpringSide](#)
- [Maemo](#)

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

网站客服 | 杂志客服 | 微博客服 | webmaster@csdn.net | 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved

