YACC（BISON）使用指南

YACC（Yet Another Compile-Compiler）是语法分析器生成工具，它生成的是 LALR 分析器。Yacc 于上世纪 70 年代产生，是美国贝尔实验室的产品，已经用于帮助实现了几百个编译器。

Yacc 是 linux 下的工具，本实验使用的编译工具是 linux 环境（或者在 windows 下模拟 linux 环境）下的 bison，它与 Yacc 的使用方法基本相同，只有很少的差别。

一．YACC 的使用方法：

1. 用户按照 Yacc 规定的规则写出文法说明文件，该文件一般以.y 为扩展名（有的系统以.grm 为扩展名。）
2. Yacc 编译器将此文法说明文件（假设该文件为 filename.y）转换成用 C 编写的语法分析器文件 filename.tab.c（如果在编译时加上某些参数还可以生成头文件 filename.tab.h）。这个文件里至少应该包含语法分析驱动程序 yyparse()以及 LALR 分析表。在这个文件里，语法分析驱动程序调用 yylex()这个函数获取输入记号，每次调用 yylex()都能获取一个输入记号。yylex()可以由 lex 生成，也可以自己用 c 语言写一个 yylex()函数，只要每次调用该函数时返回一个记号即可。
3. 用 C 编译器（本实验所给工具为 linux 下的 gcc）将 filename.tab.c 编译为可执行文件 a.out（cygwin 下默认为 a.exe 或无后缀名）。
4. a.out(a.exe)即为可执行的语法分析器。

二．文法说明文件（即 Yacc 源程序）的写法：

文法说明文件，顾名思义，就是将一个语言的文法说清楚。在 Yacc 中，这个说明文件可以分为三部分，以符号%%分开：

[第一部分：定义段]
%%
第二部分：规则段
[%%
第三部分：辅助函数段]

其中，第一部分及第三部分和第三部分之上的%%都可以省略（即上述方括号括起的部分可以省略）。以%开头的符号和关键字，或者是规则段的各个规则一般顶着行首来写，前面没有空格。

1. 第一部分定义段的写法：

定义段可以分为两部分：

第一部分以符号%{和%}包裹，里面为以 C 语法写的一些定义和声明：例如，文件包含，宏定义，全局变量定义，函数声明等。

第二部分主要是对文法的终结符和非终结符做一些相关声明。这些声明主要有如下一些：%token，%left，%right，%nonassoc，%union，%type，%start。下面分别说明它们的用法。

%token 定义文法中使用了哪些终结符，定义形式为：

```
%token TOKEN1 TOKEN2 TOKEN3 ….
```

其中 TOKEN1，TOKEN2 等为终结符，终结符一般全大写。

%left，%right，%nonassoc 也是定义文法中使用的终结符，定义形式与%token 类似，但是他们定义的终结符具有某种优先级和结合性，%left 表示左结合，%right 表示右结合，%nonassoc 表示不可结合（即它定义的终结符不能连续出现：例如<，如果文法中不允许出现形如 a<b<c 的句子，则<就是不可结合的）。而优先级关系则是以他们定义出现的顺序决定的，先定义的优先级低，最后定义的优先级最高，同时定义的优先级相同。例如，如果有如下定义：

```
%left A B
%nonassoc C
%right D
```

则表示优先级关系为： A=B < C < D，而结合性关系为：A,B 左结合，C 不可结合，D 右结合。

%start 指定文法的开始符号（非终结符），定义形式为： **`%start startsym`** ，其中 startsym 为文法的开始符号。如果不使用%start 定义文法开始符号，则默认在第二部分规则段中定义的第一条产生式规则的左部非终结符为开始符号。

%union 和%type 用来处理文法中各符号所带的属性。在词法分析的学习中，我们知道记号是由记号名和记号的属性值两部分组成的，文法中的终结符就是记号，他们有属性值，同样，非终结符也是可以有属性值的。

Yacc 维护一个栈来保存文法符号的“属性值”，这个栈与移进-归约分析中的文法符号栈是对应的：即，如果移进归约分析栈中某个位置存放文法符号 X，则对应的 Yacc“属性值”栈中就存放 X 的属性值。

Yacc 将这个属性值栈的栈内元素的类型定义为 YYSTYPE（这是一个宏），默认状态下，YYSTYPE 定义为 int 类型，你可以在文法说明文件第一部分定义段中重新定义 YYSTYPE 为其他类型，例如，用 **`#define YYSTYPE double`** 将其定义为 double 类型。

如果你想让属性值栈可以存放多种类型的属性值，例如整型和字符串型等（这在很多情况下是需要的，比如你希望标识符 ID 的属性是字符串而整型数 NUM 的属性是整型值），你最好将属性值栈元素的类型定义为一种 union 类型，此时，你可以用%union 来定义它。例如，如下这样的定义会将 Yacc 属性值栈元素的类型定义为包含 num 和 id 两个域的联合体，其中 num 域的类型为 int 而 id 域的类型为 char*。

```
%union {
  int num;
  char * id;
}
```

如果你在写文法说明文件时，需要用到一些终结符和非终结符的属性值，那么你可以先将这些属性值放入对应的属性值栈中，并在需要使用的时候从属性值栈中取出。不过，在你这么做之前，你首先需要定义这些终结符和非终结符的属性值的类型。

对于终结符的类型，你可以这样定义：

```
%token <num> TOKEN1
%token <id> TOKEN2 TOKEN3
```

上述定义会将终结符 TOKEN1 定义为属性值栈 num 域所具有的类型（即 int 类型），而将 TOKEN2 和 TOKEN3 定义为 id 域所具有的类型（char*类型）。

对于非终结符，可以这样用%type 定义其属性值的类型，例如：

```
%type <id> sym1 sym2
%type <num> sym3
```

上述定义将非终结符 sym1 和 sym2（非终结符一般用小写单词表示）定义为 char*类型，而将 sym3 定义为 int 类型。

2. 第二部分规则段的写法：

规则段实际上定义了文法的非终结符及产生式集合，以及当归约整个产生式时应执行的操作。假如产生式为 expr → expr plus term | term，则在规则段应该写成：

```
expr : expr PLUS term      {语义动作}
     | term                {语义动作}
     ;
```

其中，产生式左部的非终结符 expr 应写在冒号左边，冒号表示产生符号→，产生式右部每个用 | 分隔的部分单独写一行并用 | 分隔，整组产生式写完后，结尾处应加分号;，分号表示同一左部的一组产生式结束。

每一行产生式后面大括号括起的语义动作表示该条产生式归约时应该执行的操作，语义动作使用 C 语言的语法来写，他们将会被直接拷贝到 yacc 编译器生成的 c 语言源程序文件中。在语义动作中，可以引用存放在属性值栈中的文法符号的属性值，$$符号可引用产生式左部非终结符的属性值，而$i 则可以引用产生式右部第 i 个文法符号的属性值，当然，前提是你必须为这些文法符号的属性定义了类型，而且在你使用这些属性值之前，已经将这些属性值放入了属性值栈对应的地方。例如：

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

上述例子中，归约 expr PLUS term 为 expr 时，将会做如下操作：从属性值栈中取出产生式右部 expr 的属性值（$1）和 term 的属性值（$3），将两者的和存入属性值栈中对应产生式左部的 expr 的属性值的位置（$$）。归约 term 为 expr 时，则将 term 的属性（$1）存入产生式左部 expr 的属性值（$$）中。

注意，在这个例子中，PLUS 是终结符，而 expr 和 term 都是非终结符（能出现在产生式左部的符号都已定义为非终结符）。由于用到了 expr 和 term 的属性值，因此必须先用%type 对 expr 和 term 的属性值类型进行定义（除非属性值栈只能存放默认类型 int 的元素，并且 expr 和 term 的属性值类型都为 int，这种情况下可以不必定义他们的属性值类型）。本例中 $$=$1; 这一语义动作实际上可以省略不写，因为将 term 归约为 expr 的过程在移进归约分析中实际上是将栈顶的 term 退栈，而将 expr 入栈，对应的属性值栈中的操作实际上是将 term 的属性值退栈，再将 term 的属性值作为 expr 的属性值入栈（因为 term 的属性值要赋给 expr 的属性值），最终的结果是 term 的属性值仍然留在属性值栈上原来的位置，只不过现在这个位置我们认为是 expr 的属性值。

3. 第三部分辅助函数段的写法：

辅助函数段用 C 语言语法来写，辅助函数一般指在规则段中用到或者在语法分析器的其他部分用到的函数。这一部分一般会被直接拷贝到 yacc 编译器产生的 c 源文件中。

一般来说，除规则段用到的函数外，辅助函数段一般包括如下一些例程：yylex()，yyerror()，main()。

int yylex()是词法分析程序，它返回记号。语法分析驱动程序 yyparse()将会调用 yylex()

获取记号。如果不使用 lex 生成这个函数，则必须在辅助函数段用 C 语言写这个程序。记号由记号名和属性值构成，记号名一般作为 yylex 的返回值（注意，记号名是由%token 等定义的终结符名，这些终结符名在 yacc 内部会被宏定义成一些常数。），而属性值则由 yacc 内部定义的变量 yylval 来传递。yylval 的类型与属性值栈元素的类型相同，即，默认状态下，yylval 为 int 类型，若使用**#define YYSTYPE double** 将属性值栈元素定义为 double 类型，则 yylval 就是 double 类型，若用%union 将属性值栈元素定义为联合类型，则 yylval 也是联合类型。注意，当 yylval 是联合类型时，对它的引用要注意。例如，若属性值栈定义为

```
%union {
  int num;
  char * id;
}
```

而 yylex 返回记号 ID 的属性值为"myid"（类型为 char *）时，yylex 在返回之前，应使用如下语句将属性值传递给语法分析器：

```
yylval.id = "myid";
```

main 是主程序，主程序的主要作用是调用 yyparse()函数（至少应有一条调用 yyparse()的语句）。int yyparse()是 yacc 生成的语法分析驱动函数，语法分析成功结束时，yyparse 返回 0，而发现错误时，则返回 1，并且调用 yyerror()函数输出错误信息。yyerror 是错误报告例程。如果辅助函数部分不包含 yyerror 和 main 函数，也可以通过链接时将例程库中标准的 yyerror 和 main 链接进来，链接方法是在链接命令尾端加-ly 参数。

4. YACC 中的注释：

Yacc 文法说明文件中，凡是可以使用 C 语法写的部分，都可以用 C 语言的注释/**/和//，其它部分的注释可以用/**/，但是注意，/*之前必须有先导空格，不能顶着行首来写。

三．使用 LEX 定义词法分析器并把它和 YACC 写的语法分析器链接起来

假设用 yacc 写的文法说明文件为 filename.y，由于使用 LEX 生成 yylex()，因此 filename.y 的辅助函数段不需要定义 yylex()，只需在定义段加上函数声明 **int yylex();** 即可。用 yacc 编译器对 filename.y 进行编译，编译时带上参数-d，即在 linux 下使用如下命令：

```
bison –d filename.y
```

此时编译器除生成 filename.tab.c 以外，还将生成名为 filename.tab.h 的头文件。该头文件中包含 filename.y 中定义的所有终结符的常量定义，属性值栈的类型定义，以及变量 yylval 的外部引用定义。

假设用 Lex 写的词法分析规则文件为 filename.l，则在 filename.l 的声明部分应包含头文件 filename.tab.h，即，在 filename.l 声明部分应包含如下语句：

```
#include "filename.tab.h"
```

并且，filename.l 文件中凡涉及返回记号名的部分，都返回 filename.y 中定义的终结符名；而用 yylval 返回记号属性值。用命令 **flex filename.l** 编译 filename.l 得到 lex.yy.c。lex.yy.c 中包含了函数 yylex()。

用如下命令编译链接得到语法分析程序（其中方括号内的部分可省略）：

```
gcc [–o outfile] filename.tab.c lex.yy.c [-lfl] [-ly]
```

`-lfl` 是链接 `flex` 库函数的编译选项，`-ly` 是链接 `yacc` 库函数的编译选项，`-o outfile` 是指定输出文件名的编译选项，其中 `outfile` 为用户指定的文件名，若不用`-o`

选项，默认生成可执行文件 a.exe，使用-o 选项后，生成的可执行文件为 outfile.exe。

四．用 YACC 解决动作冲突

如果你在文法说明文件中说明的文法有分析动作冲突（移进-归约冲突或者归约-归约冲突），则 Yacc 有其一般解决方法：除特别说明外，对于归约-归约冲突，Yacc 优先于先出现的产生式（即写在前面的产生式）；对于移进-归约冲突，Yacc 优先于移进。

当然，你可以在文法说明文件中通过%left，%right 等定义终结符的优先级和结合性，用优先级和结合性解决分析动作的冲突。当你定义了终结符的优先级和结合性后，Yacc 解决分析动作冲突的方式为：当移进终结符 a 和归约产生式 A→α相冲突时，若产生式 A→α的优先级高于 a，或者两者优先级相同但产生式左结合时，则按 A→α归约，否则移进 a。产生式的优先级和结合性与产生式右部最右面的终结符（注意，非终结符不用考虑）的优先级和结合性相同。可以用%prec 强制定义产生式的优先级和结合性。例如下面规则中：

```
expr : expr MINUS expr            {$$ = $1 - $3;}
     | MINUS expr %prec UMINUS    {$$ = -$1;}
     ;
```

第一个产生式右部为 expr MINUS expr，该产生式的优先级与结合性跟终结符 MINUS 相同（因为 MINUS 是该产生式右部最右边的终结符）；而第二个产生式右部为 MINUS expr，该产生式的优先级与结合性本应与 MINUS 相同，但由于用%prec UMINUS 强制定义了其优先级与结合性跟 UMINUS 相同，所以，第二个产生式的优先级和结合性应与 UMINUS 相同。

使用 **bison -v filename.y** 命令编译时（注意，要加-v 参数），会生成 filename.output 文件，这个文件是个文本文件，里面说明了该语法分析器使用的识别活前缀的 DFA。如果你在文法说明文件中定义的文法有冲突，则该文件还会报告冲突产生的位置和原因。例子 calculator0 中给出了一个无冲突的 cal0.output 文件，例子 calculator3 给出了一个有移进-归约冲突的 cal3.output 文件，这两个文件中都有对关键部分的注释，请仔细阅读。

五．跟踪你的语法分析器：

当你用 Yacc 写的语法分析器通过 bison 编译，但是却不能按照你所想要的语法分析方式去运行时，你可以通过对它的动作进行跟踪来找出出现问题的原因。在文法说明文件的定义段的%{和%}内加上**#define YYDEBUG 1**（见 calculator0），或者在定义段加上**%debug**（见 calculator1）都可以开放跟踪模式（适用于 bison）。

开放跟踪模式后，只需要在 main 函数中给 yydebug 赋一个非 0 的值就可以在运行语法分析器时得到每步运行的详细信息。

六．关于 makefile 的简单介绍

由于使用 bison 和 flex 联合写一个语法分析器时，需要的编译步骤稍显复杂，用 makefile 可以将这个复杂的编译步骤简化，因此，这里简单介绍一下 makefile 的使用，本指南给出的 Yacc 实例的编译都是使用 makefile 完成的。

Makefile 告诉我们如何对一个包含若干源文件的工程进行编译，比如，先编译什么，后编译什么，怎样链接等等。Makefile 文件是一个纯文本文件，它实际上描述了一些编译的规则。Makefile 文件的文件名可以是 makefile，也可以是 Makefile。当 makefile 写完后，只需要打一个 **make** 命令，就可以完成所有的编译工作。

1. makefile 规则的写法和原理：

下面是一个一般的 makefile 规则的结构：

```
target... : prerequisites ...
    command
    ...
    ...
```

注意：command 前面必须是一个 tab 键，而不能用空格代替。

target 是目标文件，它可以是 object file（.o 文件），也可以是可执行文件（例如.exe 文件），还可以仅仅是一个标签；

prerequisites 是要生成 target 所需要的文件的列表；

command 是 make 真正执行的命令。

这实际上是一个文件的依赖关系，即，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。也就是说，如果 prerequisites 中有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 makefile 的规则。也就是 makefile 中最核心的内容。

makefile 中的注释以#开头，#开头直到行末的部分为注释内容。

2. 一个 makefile 的例子：

下面这个例子是 calculator2 中的 makefile，它包含了 7 条 makefile 规则，每一条的结构都与上述 makefile 规则的结构相同，我将在每条规则后面进行解释：

```
cal2.exe: cal.tab.o lex.yy.o
    gcc -o cal2 cal.tab.o lex.yy.o -ly
#   如果 cal2.exe 还不存在，则要生成 cal2.exe 文件，首先看 cal.tab.o 和
#lex.yy.o 是否存在，如果不存在，则根据其他的 makefile 规则先生成这两文件，如果
#已存在，则检验他们是否是最新的文件，如果不是的话要更新他们。然后运行命令 gcc –o
#cal2 cal.tab.o lex.yy.o –ly（这个命令将 cal.tab.o 和 lex.yy.o 链接起来）
#生成 cal2.exe。
#    如果 cal2.exe 已经存在，则需要将 cal2.exe 更新为最新文件。先将 cal.tab.o
#和 lex.yy.o 更新到最新以后，看他们是否比 cal2.exe 新，如果有一个以上比
#cal2.exe 新，则运行 gcc –o cal2 cal.tab.o lex.yy.o –ly 命令重新生成
#cal2.exe。

lex.yy.o: lex.yy.c cal.tab.h
    gcc -c lex.yy.c
#   lex.yy.o 依赖于 lex.yy.c 和 cal.tab.h，如果目标文件比其依赖文件旧，则要
#重新运行命令生成 lex.yy.o。 -c 选项表示只生成 object 文件（.o 文件）

cal.tab.o: cal.tab.c
    gcc -c cal.tab.c

lex.yy.c: cal.l
    flex cal.l

cal.tab.c: cal.y
```

```
    bison -dv cal.y

cal.tab.h: cal.y
    echo "cal.tab.h was created at the same time as cal.tab.c."
#  echo 命令表示把后面的文字在屏幕上回显一遍

clean:
    rm -f cal2.exe lex.yy.o cal.tab.o lex.yy.c cal.tab.c cal.tab.h
cal2.exe.stackdump cal.output
#    这里 clean 只是个标签，其后也没有依赖文件，运行下面的 rm 命令不会生成 clean
#文件。注意此处 rm 命令（该命令是 linux 底下删除文件的命令）的参数中间没有换行。
```

在 linux 下直接运行 **make** 命令，则会默认执行第一条 makefile 规则，即生成 cal2.exe。如果想执行其他的 makefile 规则，则需要输入命令 **make target**。其中 target 是目标文件。例如，输入 **make cal.tab.c** 则会执行第四条 makefile 规则生成 cal.tab.c，而输入 **make clean** 则会执行最后一条 makefile 规则（强制删除一些文件）。

七．使用 YACC 的实例：

注意：阅读每个 YACC 实例之前，请先阅读对应的 readme 文件。

本指南给出使用 YACC 的 4 个例子，calculator0-3，每个例子中都包含必要的文法说明文件、词法分析规则文件（如果需要的话）、makefile 文件和 readme 文件（对每个例子进行简单说明），calculator0 和 calculator3 还带有 output 文件，请仔细阅读。

另外，还有两个用 YACC 写的简单语言语法分析器的例子 parser0 和 parser1。parser0 是单纯进行语法分析，无任何语义动作，而 parser1 增加了构造分析树的语义动作。这两个例子中也都有 makefile 文件和 readme 文件，阅读例子之前请首先阅读 readme。

由于时间仓促，这些例子并未进行足够的测试，若同学们发现错误，请联系 gelin@ouc.edu.cn 更正。

八．参考资料：

1.《编译原理》，蒋立源等著，西北工业大学出版社，2005.1 第三版，P237-P248

2.《Lex 与 Yacc（第二版）》，John R.Levine 等著，杨作梅等译，机械工业出版社，2003.1 第一版。

3. Bison—the Yacc-compatible Parser Generator，Charles Donnolley & Richard Stallman，在线文档，2009.4

4.《跟我一起写 makefile》，陈皓，在线文档