FLEX&BISON 分析

1.1 MAKEFILE 中 EXPR 任务分析：

第一行：expr 的生成依赖于两个文件 config/expr.y 和 config/expr.lex
第二行：使用 Bison 编译 config/expr.y 得到 src/expr.tab.c 。
第三行：使用 Flex 编译 config/expr.lex 得到 src/expr.lex.c 。
第四行：使用 gcc 编译 src/expr.tab.c 和 src/expr.lex.c 生成可执行文件 bin/expr.out 。
它基本说明了由 flex 和 bison 生成语言编译器的过程。

1.2 BISON

bison 输入文法的规范格式

bison 的输入文法主要有四部分组成

- 第一部分是头文件声明，这部分内容将原封不动的复制到所生成的.c 文件中，基本格式举例：

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

- 第二部分主要是符号和性质的定义，基本格式如下，例:

```
%token number
```

由 flex 输出的字符记号都要在这一部分声明，它们会作为 bison 输入中的终结符，此外还可声明一些其他东西，如 union 或运算符的结合性等。

- 这部分是语法规则的定义，也是最核心的部分，主要包括各个产生式和相应的语义动作。bison 会根据这些规则生成相应的.c 代码。

```
exp     : NUMBER              { $$ = $1;       }
        | exp PLUS  exp       { $$ = $1 + $3;  }
        ;
```

':'是开始符，后面跟着产生式.';'是结束符，{}内是语义动作。{}插入的位置表示语义动作的执行时间。

- 最后一部分是函数和函数定义。

不同输入文法对生成的分析器源代码的影响

以 expr.tab.c 和 expr1.tab.c 为例，在 expr1.y 的表达式中，多定义了一个 term 和 factor,以区别各算符的优先级，而 expr.y 用的是二义文法但是由于在前面定义了算符的优先级，最后得到的也是非二义文法。两者的功能是一样的，但实现起来略有差别。
由于他们前面声明的非终结符等都是一样的，两者得到的.tab.h 文件是一样的。
他们分别生成的代码 expr.tab.c 和 expr1.tab.c 中，两者的数组和生成的分析表的生成不一样。
如在 expr.tab.c 的 1350-1421 行，在表达式的形成规则上，两者就有差异。

BISON 中的语法制导翻译

以 expr.y 为例，它的语法翻译部分主要是

```
Exp     : NUMBER                { $$ = $1;       }
        | exp PLUS  exp          { $$ = $1 + $3;   }
        | exp MINUS exp          { $$ = $1 - $3;   }
        | exp MULT  exp          { $$ = $1 * $3;   }
        | exp DIV   exp         { $$ = $1 / $3;   }
        | MINUS  exp %prec MINUS { $$ = -$2;      }
        | exp EXPON exp          { $$ = pow($1,$3);}
        | LB exp RB             { $$ = $2;       }
        ;
```

{}内是语义动作，他可以加在产生式右部的任何地方，表示该语义动作执行的时机，bison 中采用加入“标记非终结符”的办法将语义动作移到产生式后面。对应到.c 文件如下：

```
switch (yyn)
   {
      case 5:

/* Line 1455 of yacc.c  */
#line 34 "config/expr.y"
   { printf("%g\n",(yyvsp[(1) - (2)].val));}
   break;

  case 6:
```

```
/* Line 1455 of yacc.c  */
#line 36 "config/expr.y"
    { (yyval.val) = (yyvsp[(1) - (1)].val);        }
    break;

  case 7:

/* Line 1455 of yacc.c  */
#line 37 "config/expr.y"
    { (yyval.val) = (yyvsp[(1) - (3)].val) + (yyvsp[(3) - (3)].val);   }
    break;

  case 8:

/* Line 1455 of yacc.c  */
#line 38 "config/expr.y"
    { (yyval.val) = (yyvsp[(1) - (3)].val) - (yyvsp[(3) - (3)].val);   }
    break;

  case 9:

/* Line 1455 of yacc.c  */
#line 39 "config/expr.y"
    { (yyval.val) = (yyvsp[(1) - (3)].val) * (yyvsp[(3) - (3)].val);   }
    break;

  case 10:

/* Line 1455 of yacc.c  */
#line 40 "config/expr.y"
    { (yyval.val) = (yyvsp[(1) - (3)].val) / (yyvsp[(3) - (3)].val);   }
    break;

  case 11:

/* Line 1455 of yacc.c  */
```

```
#line 41 "config/expr.y"
    { (yyval.val) = -(yyvsp[(2) - (2)].val);       }
    break;

  case 12:

/* Line 1455 of yacc.c  */
#line 42 "config/expr.y"
    { (yyval.val) = pow((yyvsp[(1) - (3)].val),(yyvsp[(3) - (3)].val));}
    break;

  case 13:

/* Line 1455 of yacc.c  */
#line 43 "config/expr.y"
    { (yyval.val) = (yyvsp[(2) - (3)].val);        }
    break;

/* Line 1455 of yacc.c  */
#line 1420 "src/expr.tab.c"
      default: break;
    }
```

Expr.y 中语义动作与 expr.tab.c 中的 case 个数刚好一致，每一个语义动作对应到一个 case,它指明了每个语义动作的执行时机和具体实现方法。

1.3 FLEX

- *flex 输入规范*

  类似于 bison 的输入结构，主要分为四部分，如下：

  %{

  　　头文件

  %}

  　　定义段(definitions)

  %%

  　　规则段(rules)

  %%

用户代码段(user code)