**编译原理第三次实验报告**

**Calculator Extension**

19计算机科学与技术（1）班 梅雨欣 2019329621004

目录

[一、实验目的 2](#_Toc90421761)

[二、实验要求 2](#_Toc90421762)

[三、实验准备与设计 2](#_Toc90421763)

[3.1 第三方工具Linux、VSC、gcc、Flex、Bison 2](#_Toc90421764)

[3.2 词法、语法、语义分析 3](#_Toc90421765)

[3.3 常见的中间语言形式 4](#_Toc90421766)

[3.4 实验设计 5](#_Toc90421767)

[四、完整代码 7](#_Toc90421768)

[五、实验结果 14](#_Toc90421769)

[六、实验总结 15](#_Toc90421770)

# 一、实验目的

1、对**《科学计算器》**进行语义分析。

# 二、实验要求

**1、科学计算器的功能**

继续完成以下2项功能：

7） 进制转换

8） 变量存储(计算器上的 MC,MR,M+,M-)

**2、实现 p342 6.26.b**

Rewrite the YACC specification of calculator, so that it prints both a postfix translation and the value.

**3、给出实验结论和反思。**

# 三、实验准备与设计

## 3.1 第三方工具Linux、VSC、gcc、Flex、Bison

本次实验是由Visual Studio Code编辑器通过SSH连接Linux操作系统（Ubuntu 64位）环境下，使用Flex + Bison工具完成。因为Linux编译执行文件只需要命令少，很**方便使用且能查看执行过程**。而Vistual Studio过于庞大，**占用内存**和**性能速度**远不如Linux直接使用terminal命令进行处理。

Flex是一个词法分析程序的自动生成工具。它输入描述构词规则的一系列正规式，然后构建有穷自动机和这个有穷自动机的一个驱动程序，进而生成一个词法分析程序。

Bison是一个语法分析程序的自动生成工具。它接受语言的文法，构造一个LALR(1)分析程序。因为它采用语法制导翻译的思想，还可以接受用C语言描述的语义动作，从而构造一个编译程序。

其执行过程如下：

1. 创建语言描述文件 (.y 文件)
2. 编写词法分析器函数 yylex()、错误报告函数 yyerror()
3. 在 main() 中调用分析器函数 yyparse()
4. 执行 bison -d，由 .y 文件 产生 .tab.c 和 .tab.h 文件
5. 执行 gcc，把 .tab.c 文件编译和链接成可执行程序

Linux环境命令行如下所示：

|  |
| --- |
| flex -o CompileExp03.yy.c CompileExp03.l  bison -d CompileExp03.y  gcc CompileExp03.yy.c CompileExp03.tab.c -lm -o CompileExp03  ./CompileExp03 |

## 3.2 词法、语法、语义分析

**词法分析（Lexical analysis或Scanning）和词法分析程序（Lexical analyzer或Scanner）**

　　词法分析阶段是编译过程的第一个阶段。这个阶段的任务是从左到右字符逐个读入源程序，即对构成源程序的字符流进行扫描，然后根据构词规则识别单词(也称单词符号或符号)。词法分析程序实现这个任务。词法分析程序可以使用lex等工具自动生成。

**语法分析（Syntax analysis或Parsing）和语法分析程序（Parser）**

语法分析是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等。语法分析程序判断源程序在结构上是否正确。源程序的结构由上下文无关文法描述。

**语义分析（Syntax analysis）**

语义分析是编译过程的一个逻辑阶段. 语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查, 进行类型审查。例如一个C程序片断：

int arr[2],b;

b = arr \* 10;

源程序的结构是正确的。语义分析将审查类型并报告错误：不能在表达式中使用一个数组变量，赋值语句的右端和左端的类型不匹配。

语义分析也叫类型检查、上下文相关分析，其依赖于上下文分析，负责检查程序（语法树）的上下文相关的属性。

抽象语法树一定是符合语法的，不会存在语法错误，语义分析器会分析语义是否满足。

## 3.3 常见的中间语言形式

* 逆波兰表示法

波兰表示是一种既不须考虑优先关系、又不用括号的一种表示表达式的方法（前缀式）。另一种波兰表示形式，称为后缀式，即运算符在后。例：

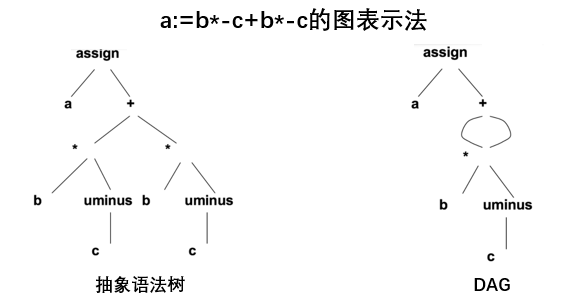
|  |
| --- |
| a+b → ab+  a\*(b+c) → abc+\*  -a+b\*c → a@bc\*+ |

* 图表示法

抽象语法树，无循环有向图（DAG）。

DAG与抽象语法树基本上一样，对表达式中的每个子表达式，DAG中都有一个结点。一个内部结点表示一个操作符，它的孩子表示操作数。

两者所不同的是，在一个DAG中代表公共子表达式的结点具有多个父结点，而在一棵抽象语法树中公共子表达式被表示为重复的子树。

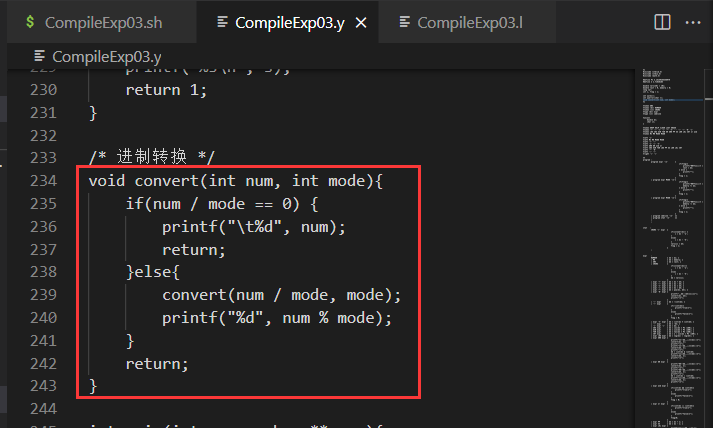


图表 1抽象语法树和DAG的区别

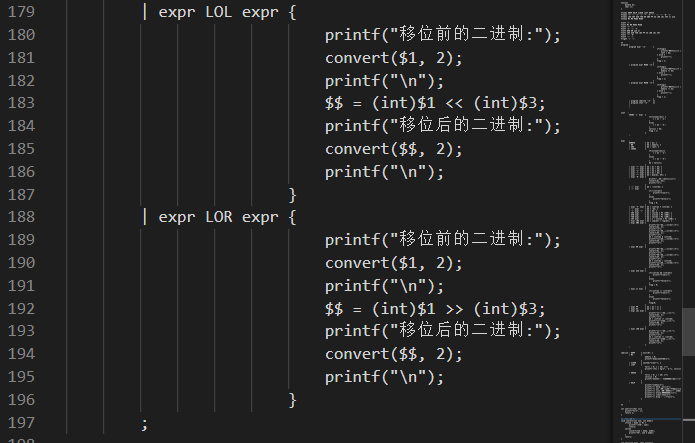
## 3.4 实验设计

* **进制转换**

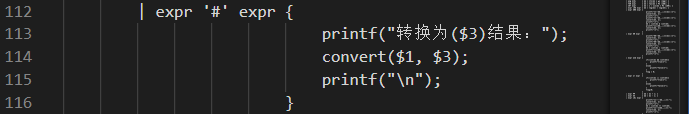
进制转换使用C语言编写函数实现，void convert参数列表：转换前数字（num），进制（mode）。在数字进行左移、右移、位运算（和、或）、进制转换时均会用到该函数。



图表 2实现进制转换的函数



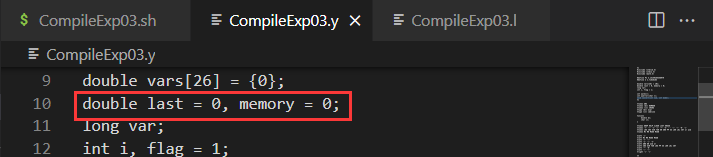
图表 3左移、右移使用进制转换情况



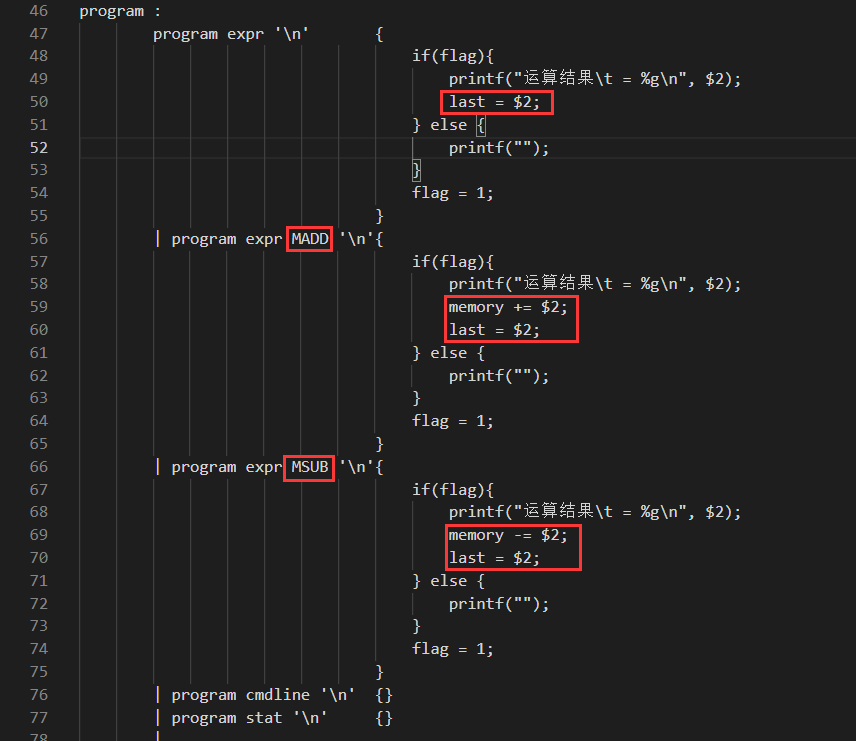
图表 4使用‘#’表示进制转换

* **变量存储（MC/MR/M+/M-）**

double类型的memory、last全局变量用以存储memory和上一次运算结果。



图表 5全局变量memory和last



图表 6M+/M-的使用

# 四、完整代码

**CompileExp03.l**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | %{  #include "CompileExp03.tab.h"  extern int yyerror(const char \*);  %}  %%  [" "; \t] { }  (0(\.[0-9]+)?)|([1-9][0-9]\*(\.[0-9]+)?) { yylval.dv = strtod(yytext, 0); return NUMBER;}  [a-zA-Z] { yylval.cv = \*yytext; return CHARA;}  [-+\*/()^%~!#=\n] {return \*yytext;}  "log" {return LOG;}  "cos" {return COS;}  "sin" {return SIN;}  "tan" {return TAN;}  "cot" {return COT;}  "&" {return AND;}  "|" {return OR;}  "&&" {return and;}  "||" {return or;}  "++" {return PP;}  "--" {return SS;}  "<<" {return LOL;}  ">>" {return LOR;}  "MC" {return MC;}  "MR" {return MR;}  "M+" {return MADD;}  "M-" {return MSUB;}  "ans" {return ANS;}  "drop" {return DROP;}  "list" {return LIST;}  "erase" {return ERASE;}  "clear" {return CLEAR;}  "help" {return HELP;}  %%  int yywrap(){  return 1;  } |

**CompileExp03.y**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249 | %{  #include <stdlib.h>  #include <stdio.h>  #include <math.h>  #define PI 3.14159265358979  #define e 2.71828183  double vars[26] = {0};  double last = 0, memory = 0;  long var;  int i, flag = 1;  int yylex();  int yyerror(char \*);  void convert(int num, int mode);  %}  %token ANS  %token <dv> NUMBER  %token <cv> CHARA  %type <dv> expr  %type <cv> cmdline  %union{  double dv;  char cv;  }  %token DROP HELP CLEAR LIST ERASE  %token '+' '-' '\*' '/' '^' '%' '`' '~' '!' '#' '='  %token LOG COS SIN TAN OR AND PP SS LOR LOL COT or and  %token MC MR MADD MSUB  %left '='  %left MC MR MADD MSUB  %left '+' '-'  %left '\*' '/' '%'  %left AND OR and or  %left COS SIN TAN LOG PP SS LOR LOL COT  %left '^' '#'  %left '~' '!'  %right '(' ')'  %%  program :  program expr '\n' {  if(flag){  printf("运算结果\t = %g\n", $2);  last = $2;  } else {  printf("");  }  flag = 1;  }  | program expr MADD '\n'{  if(flag){  printf("运算结果\t = %g\n", $2);  memory += $2;  last = $2;  } else {  printf("");  }  flag = 1;  }  | program expr MSUB '\n'{  if(flag){  printf("运算结果\t = %g\n", $2);  memory -= $2;  last = $2;  } else {  printf("");  }  flag = 1;  }  | program cmdline '\n' {}  | program stat '\n' {}  |  ;  stat :  CHARA '=' expr {  if(islower($1)) {  i = $1 - 'a';  }  else{  i = $1 - 'A';  }  vars[i] = $3;  flag = 1;  }  ;  expr :  NUMBER { $$ = $1; }  | MR { $$ = memory; }  | ANS { $$ = last; }  | CHARA {  if(islower($1)){  i = $1 - 'a';  }  else{  i = $1 - 'A';  }  $$ = vars[i];  }  | expr '+' expr { $$ = $1 + $3; }  | expr '-' expr { $$ = $1 - $3; }  | expr '\*' expr { $$ = $1 \* $3; }  | expr '/' expr { $$ = $1 / $3; }  | expr '^' expr { $$ = pow($1, $3); }  | expr '#' expr {  printf("转换为($3)结果：");  convert($1, $3);  printf("\n");  }  | '~' expr { $$ = ~(int)$2; }  | '!' expr {  if(!(int)$2){  printf("true\n");  }  else{  printf("false\n");  }  flag = 0;  }  | expr '%' expr { $$ = (int)$1 % (int)$3; }  | '-' expr { $$ = -$2; }  | '(' expr ')' { $$ = $2; }  | COS expr { $$ = cos($2 \* PI /180); }  | SIN expr { $$ = sin($2 \* PI /180); }  | TAN expr { $$ = tan($2 \* PI /180); }  | COT expr { $$ = 1 / sin($2 \* PI /180); }  | expr LOG expr { $$ = log($1) / log($3); }  | expr AND expr {  printf("与前的二进制($1):\n");  convert($1, 2);  printf("\n");  printf("与前的二进制($3):\n");  convert($3, 2);  printf("\n");  $$ = (int)$1 & (int)$3;  printf("结果的二进制($$):\n");  convert($$, 2);  printf("\n");  }  | expr OR expr {  printf("或前的二进制($1):\n");  convert($1, 2);  printf("\n");  printf("或前的二进制($3):\n");  convert($3, 2);  printf("\n");  $$ = (int)$1 | (int)$3;  printf("结果的二进制($$):\n");  convert($$, 2);  printf("\n");  }  | expr and expr {  if((int)$1 && (int)$3){  printf("true\n");  }  else{  printf("false\n");  }  flag = 0;  }  | expr or expr {  if((int)$1 || (int)$3){  printf("true\n");  }  else{  printf("false\n");  }  flag=0;  }  | expr PP { $$ = $1 + 1; }  | expr SS { $$ = $1 - 1; }  | expr LOL expr {  printf("移位前的二进制:");  convert($1, 2);  printf("\n");  $$ = (int)$1 << (int)$3;  printf("移位后的二进制:");  convert($$, 2);  printf("\n");  }  | expr LOR expr {  printf("移位前的二进制:");  convert($1, 2);  printf("\n");  $$ = (int)$1 >> (int)$3;  printf("移位后的二进制:");  convert($$, 2);  printf("\n");  }  ;  cmdline : DROP { exit(0); }  | MC {  memory = 0;  printf("memory已清零\n");  }  | CLEAR { system("clear"); }  | LIST {  for(i = 0; i < 26; i++)  printf("\t%c = %g\n", 'a'+i, vars[i]);  }  | ERASE {  for(i = 0; i < 26; i++)  vars[i] = 0;  printf("已经清空所有的寄存器的值!\n");  }  | HELP {  printf("命令：\n");  printf(">> help :帮助\n");  printf(">> ans :列出上次计算的结果\n");  printf(">> list :列出寄存器中所有的值 'a'/'z'\n");  printf(">> erase:重置寄存器\n");  printf(">> clear:清屏\n");  printf(">> drop :退出程序\n");  }  ;  %%  int yyerror(char \*s){  printf("%s\n", s);  return 1;  }  /\* 进制转换 \*/  void convert(int num, int mode){  if(num / mode == 0) {  printf("\t%d", num);  return;  }else{  convert(num / mode, mode);  printf("%d", num % mode);  }  return;  }  int main(int argc, char \*\*argv){  yyparse();  } |

**CompileExp03.sh**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #!/bin/bash  echo "flex开始执行："  flex -o CompileExp03.yy.c CompileExp03.l  echo "bison开始执行："  #bison -o CompileExp03.tab.h CompileExp03.y  bison -d CompileExp03.y  echo "gcc开始编译："  gcc CompileExp03.yy.c CompileExp03.tab.c -lm -o CompileExp03  #echo "删除中间文件..."  #rm CompileExp03.yy.c CompileExp03.tab.\*  echo "执行生成最终目标文件："  ./CompileExp03 |

# 五、实验结果

首先是**进制转换**结果测试：

* **输入：20#5（十进制的20转换为5进制）**



* **输入：2|1（位运算）**



* **输入：2&3（位运算）**



* **输入：4>>1（4右移1位）**



* **输入：5<<2（5左移2位）**



然后是**变量存储**结果测试：



# 六、实验总结

本次实验在实验二的基础上继续实现进制转换和变量存储，难度不大，因为上次实验二基本都写完了，这次只完善了进制转换，用‘#’表示表示。补充了memory，和last全局变量的目的相似。话虽如此，要全部实现还是需要对Flex和Bison语法结构熟悉。

经过这三次实验，感觉flex和Bison有了更深的理解， flex文件是定义pattern（哪个是数字/字符），通过flex处理（词法分析）将输出切分成一段段的token（将输入的数字/字符一个个摘出来），执行不同的action：数字就自增（action），字符就输出（action）。flex 生成的tokens可以给Bison处理，更简便易调试。使用bison可以更方便的处理复杂的逻辑，编写简单，调试方便。