用 C 语言写解释器

来自 CSDN

用 C 语言写解释器（一）

声明

为提高教学质量，我所在的学院正在筹划编写 C 语言教材。《用 C 语言写解释器》系列文章经整理后将收入书中“综合实验”一章。因此该系列的文章主要阅读对象定为刚学完 C 语言的学生（不要求有数据结构等其他知识），所以行文比较罗嗦，请勿见怪。本人水平有限，如有描述不恰当或错误之处请不吝赐教！特此声明。

起因

最近，我们学院老师联系我，希望我能提供一段用 C 语言编写的 BASIC 解释器，用于 C 语言课程设计教学。我前段时间也正好着迷于“语言”本身，本就有打算写一个解释器，这下正中我下怀，于是欣然接受。

以前在图书馆看过梁肇新的《编程高手箴言》，第四章“编程语言的运行机理”中就包含了一段 C 语言编写的 BASIC 解释器代码，但代码好像并不完整（我翻了好几遍，都没发现函数 get_token 的实现代码）；再者，这次的代码还有其他用处，不宜牵涉版权问题；最后的原因是我有“想自己编码”的冲动 ^_^。综上所述，我要从零开始用 C 语言来编写一个 BASIC 解释器。

前置知识

1. 要编写解释器，首先就要明白什么是解释器（详细的解释请参看维基百科：http://zh.wikipedia.org/zh-cn/解释器）。盗用《编程高手箴言》里的话：解释程序就是一个字符串的解释器（P165 解释语言的原理）。所以，如果仅仅是为我个人编写的话，我宁可会借助 lex & yacc 甚至 perl，而不会纯粹用 C 语言来写。

2. 在起因中已经提过，这个程序会在学弟学妹们学完 C 语言后作为综合实验。因此需要你熟悉 C 语言的语法、单链表添加/删除节点等操作以及栈的概念（这些内容大部分都能在 C 语言的教材中找到），一些相对冷僻的技术（例如 setjmp/longjmp）则不会出现在程序中。

关于语言

我在《编程和语言之我见》一文中提过，编程是一个很宽泛的概念。从某种意义上来说所有的软件都是一种特定的语言，但根据程序本身的灵活性可以分为“硬编码”、“可配置”、“可控制”和“可编程”四类（详见《四类程序》）。如果一个程序的灵活性达到了“可编程”，它的配置文件就可以被看作一种“编程语言”，而该程序本身也就是一个“解释器”。

要做到“可编程”，程序至少应该具备“输入/输出”、“表达式运算”、“内存管理”和“按条件跳转”四个功能（详见《用 DOS 批处理来做数字图像处理》）。这正好对应了冯·诺依曼计算机的结构：以运算器和控制器为中心，输入/输出设备与存储器之间的数据传输都要经过运算器。下面详细介绍各个部分。

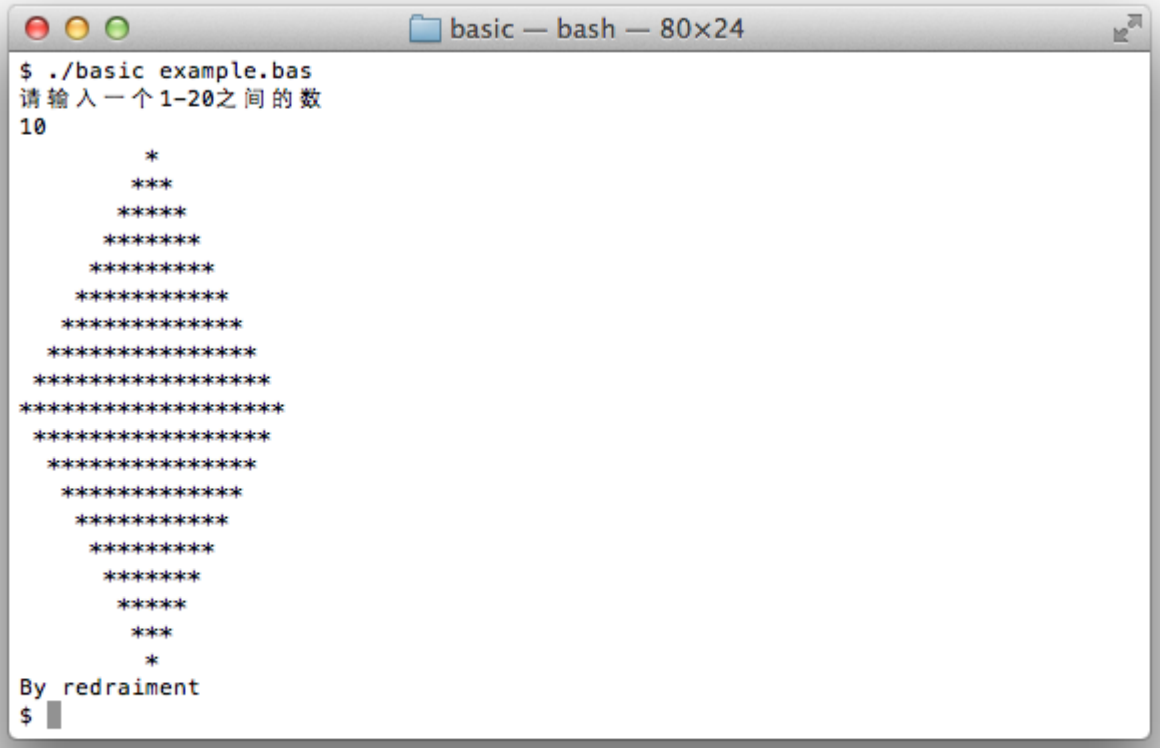
我们的目标

我们要编写解释器，自然也逃不出上面的条条例例。语法就参考 BASIC，但因为是设计我们自己的语言，当然可以根据个人兴趣进行“添油加醋”（比如表达式里提供神往已久的阶乘运算 ^_^）。下面是一段 BASIC 的示例代码（example.bas）：

```
0009 N = 0
0010 WHILE N < 1 OR N > 20
0011   PRINT "请输入一个 1-20 之间的数"
0012   INPUT N
0013 WEND
0020 FOR I = 1 TO N
0030   L = "*"
0040   FOR J = 1 TO N - I
0050     L = " " + L
0060   NEXT
0070   FOR J = 2 TO 2 * I - 1 STEP 2
0080     L = L + "**"
```

```
0090   NEXT
0100   PRINT L
0110 NEXT
0120 I = N - 1
0130 L = ""
0140 FOR J = 1 TO N - I
0150   L = L + " "
0160 NEXT
0170 FOR J = 1 TO ((2*I) - 1)
0180   L = L + "*"
0190 NEXT
0200 PRINT L
0210 I = I - 1
0220 IF I > 0 THEN
0230   GOTO 130
0240 ELSE
0250   PRINT "By redraiment"
0260 END IF
```

BASIC 语法要求行首提供一个 1->9999 之间的数字作为该行的行号（当前行的行号不小于上一行的行号），供 GOTO 语句跳转时调用。BASIC 的语法比 C 严格，这不仅可以降低代码的复杂性还使语言本身更易学。上面的代码差不多涵盖了我们需要实现的所有功能，如果能被正确解析，你将看到下面的运行效果：



下面来依次讨论要实现的功能。

输入/输出（IO）

通过输入/输出来和外部程序或人交互，这是脱离“硬编码”的最基本要求。输入/输出也是很抽象的概念，它并不局限于标准输入输出端（键盘、显示器等），也可以通过文件、互联网等方式获得数据（因此 C 语言中除了 scanf、printf 等，其实 #include 指令也算是一种 IO 操作）。我们这个程序并不强调 IO，因此只要求实现 INPUT 和 PRINT 两条指令，分别用于从键盘输入数据和打印到屏幕。指令的格式如下：

`INPUT var[, var ...]`

其中 `var` 代表变量名(下同)，变量之间用逗号隔开。

作用：从键盘获得一个或多个值，并赋值到相应的变量。同时输入多个变量时，输入的每个数之间用空格、回车或制表符隔开。

例如：`INPUT A, B, C`

`PRINT expression[, expression ...]`

其中 `expression` 为表达式（下同），表达式之间用逗号隔开。

作用：对表达式求值，将结果输出到屏幕并换行。如果有多个表达式，表达式之间用制表符（`/t`）隔开。

例如：PRINT I * 3 + 1, (A + B)*(C + D)

表达式运算

在《DOS》中我称呼它为“算术运算”。但对于计算机来说，“算术运算”不仅包含诸如“四则运算”等算术运算，还包括“关系运算”和“逻辑运算”。为了避免歧义，在此就改称它为“表达式运算”。“表达式运算”是整个程序的核心，地位相当于计算机的运算器。在我们的程序中，需要实现以下几种运算符：

符号	名称	优先级	结合性
(	左括号	17	left2right
)	右边	17	left2right
+	加	12	left2right
-	减	12	left2right
*	乘	13	left2right
/	除	13	left2right
%	取模	13	left2right
^	求幂	14	left2right
+	正号	16	right2left
-	负号	16	right2left
!	阶乘	16	left2right
>	大于	10	left2right
<	小于	10	left2right
=	等于	9	left2right
<>	不等于	9	left2right
<=	不大于	10	left2right
>=	不小于	10	left2right
AND	逻辑与	5	left2right

OR 逻辑或4 left2right

NOT逻辑非15 right2left

内存管理

在我们这个迷你型的解释器中，可以不用考虑内存空间动态分配的问题，只要实现简单的变量管理。我们默认提供 A-Z 26 个可用的弱类型变量（可以随意赋值为整数、浮点数或字符串）。变量要求先赋值才能使用，否则就会提示变量不可用（因此示例代码中第一行就是给 N 赋值为 0）。赋值语句的格式为

```
[LET] var = expression
```

其中 LET 是可选的关键字。BASIC 中不允许出现 var1 = var2 = expression 这样的赋值语句，

因为在表达式中“=”被翻译为“等于”，所以赋值符合没有出现在上面的表格中。

作用：计算表达式的值，并将结果赋值给变量 var。

例如：I = (123 + 456) * 0.09

按条件跳转

如果设计一门最简洁的语言，那它的控制语句就只需提供像汇编中的 JMP、JNZ 等根据条件跳转的语句即可，通过它们的组合即可模拟出 IF、WHILE、FOR、GOTO 等控制语句。但 BASIC 作为一门高级语言，需要提供更高层、更抽象的语句。我们将会实现以下四条语句：

1)

```
GOTO expression
```

其中 expression 是一个数值表达式，计算结果必须为可用的行号。因为它是一个表达式，通过动态计算就能模拟子程序调用。

作用：无条件跳转到指定行。

例如：GOTO 120+10

2)

```
IF expression THEN
  sentence1
[ELSE
  sentence2]
END IF
```

其中 sentence 是语句块（下同），包含一条或多条可执行语句。ELSE 为可选部分。

作用：分支结构。但表达式值为真（数字不等于 0 或者字符串不为空）时执行语句块 1；否则，有 ELSE 语句块时执行 ELSE 语句块。

例如：

```
        IF 1=1 THEN
           PRINT "TRUE"
        ELSE
           PRINT "FALSE"
        END IF
```

3)

```
FOR var = expression TO expression [STEP expression]
  sentence
NEXT
```

所有表达式均为数值表达式。STEP 为可选部分，为迭代器的步长。步长表达式的值不允许为0。

作用：循环迭代结构

例如：

```
        FOR I = 1 TO 10 STEP 3
          PRINT I
        NEXT
```

4)

```
WHILE expression
  sentence
WEND
```

作用：迭代执行语句块，直到表达式的值为假。

例如：

```
        WHILE N < 10
          N = N + 1
        WEND
```

更多细节

1. BASIC 的源代码不区分大小写；
2. 本程序在实现中没有处理字符转义，因此无法无法输出双引号。在介绍完所有源码后，如果你有兴趣可以尝试自行完善；
3. 本程序同样没有考虑注释（REM 关键字）。其实这很简单，但这个问题同样留给你来处理^_^；
4. 也许你也会有兴趣添加 GOSUB 和 RETURN 关键字，让子程序功能从 GOTO 中解放出来。

总结

这一篇主要介绍了我们编写的解释器要实现的功能，接下来会有一系列文章来逐步详细介绍解释器的实现。在下一篇中会首先介绍解释器的核心部分——表达式求值。请关注《用 C 语言写解释器（二）》。

用 C 语言写解释器（二）

声明

为提高教学质量，我所在的学院正在筹划编写 C 语言教材。《用 C 语言写解释器》系列文章经整理后将收入书中"综合实验"一章。因此该系列的文章主要阅读对象定为刚学完 C 语言的学生（不要求有数据结构等其他知识），所以行文比较罗嗦，请勿见怪。本人水平有限，如有描述不恰当或错误之处请不吝赐教！特此声明。

内存管理

既然是表达式求值，自然需要在内存中保存计算结果以及中间值。在《用 C 语言写解释器（一）》中提过：变量要求是若类型，而 C 语言中的变量是强类型，为实现这个目标就需要定义自己的变量类型，参考代码如下（注释部分指出代码所在的文件名，下同）：

[cpp] view plaincopyprint?

```
1. //  in  basic_io.h
2. #define  MEMORY_SIZE  (26)
3.
4. typedef  enum  {
5.         var_null  =  0,
6.         var_double,
7.         var_string
8. }  variant_type;
9. typedef  char  STRING[128];
10.typedef  struct  {
11.        variant_type  type;
12.        union  {
13.                double  i;
14.                STRING  s;
15.        };
16.}  VARIANT;
17.
18.extern  VARIANT  memory[MEMORY_SIZE];
19.
20.//  in  expression.h
```

```
21. typedef  VARIANT  OPERAND;
```

程序自带 A-Z 26 个可用变量，初始时都处于未赋值（ver_null)状态。所有变量必须先赋值再使用，否则就会报错！至于赋值语句的实现请参见后面语法分析的章节。

操作符

表达式中光有数值不行，还需要有操作符。在《一》中"表达式运算"一节已经给出了解释器需要实现的所有操作符，包括"算术运算"、"关系运算"和"逻辑运算"。下面给出程序中操作符的定义和声明：

[cpp] view plaincopyprint?

```
1. //  in  expression.h
2. typedef  enum  {
3.         /*  算数运算  */
4.         oper_lparen  =  0,        //  左括号
5.         oper_rparen,               //  右括号
6.         oper_plus,                 //  加
7.         oper_minus,                //  减
8.         oper_multiply,             //  乘
9.         oper_divide,               //  除
10.        oper_mod,                  //  模
11.        oper_power,                //  幂
12.        oper_positive,             //  正号
13.        oper_negative,             //  负号
14.        oper_factorial,            //  阶乘
15.        /*  关系运算  */
16.        oper_lt,                   //  小于
17.        oper_gt,                   //  大于
18.        oper_eq,                   //  等于
19.        oper_ne,                   //  不等于
20.        oper_le,                   //  不大于
21.        oper_ge,                   //  不小于
22.        /*  逻辑运算  */
23.        oper_and,                  //  且
24.        oper_or,                   //  或
25.        oper_not,                  //  非
26.        /*  赋值  */
27.        oper_assignment,        //  赋值
28.        oper_min                   //  栈底
29.}  operator_type;
30.typedef  enum  {
31.        left2right,
32.        right2left
33.}  associativity;
```

```
typedef  struct  {
        int  numbers;                  //  操作数
        int  icp;                      //  优先级
        int  isp;                      //  优先级
        associativity  ass;    //  结合性
        operator_type  oper;  //  操作符
}  OPERATOR;

//  in  expression.c
static  const  OPERATOR  operators[]  =  {
        /*  算数运算  */
        {2,  17,  1,  left2right,  oper_lparen},          //  左括号
        {2,  17,  17,  left2right,  oper_rparen},        //  右括号
        {2,  12,  12,  left2right,  oper_plus},           //  加
        {2,  12,  12,  left2right,  oper_minus},          //  减
        {2,  13,  13,  left2right,  oper_multiply},    //  乘
        {2,  13,  13,  left2right,  oper_divide},        //  除
        {2,  13,  13,  left2right,  oper_mod},            //  模
        {2,  14,  14,  left2right,  oper_power},          //  幂
        {1,  16,  15,  right2left,  oper_positive},    //  正号
        {1,  16,  15,  right2left,  oper_negative},    //  负号
        {1,  16,  15,  left2right,  oper_factorial},  //  阶乘
        /*  关系运算  */
        {2,  10,  10,  left2right,  oper_lt},              //  小于
        {2,  10,  10,  left2right,  oper_gt},              //  大于
        {2,  9,  9,  left2right,  oper_eq},                //  等于
        {2,  9,  9,  left2right,  oper_ne},                //  不等于
        {2,  10,  10,  left2right,  oper_le},              //  不大于
        {2,  10,  10,  left2right,  oper_ge},              //  不小于
        /*  逻辑运算  */
        {2,  5,  5,  left2right,  oper_and},               //  且
        {2,  4,  4,  left2right,  oper_or},                //  或
        {1,  15,  15,  right2left,  oper_not},            //  非
        /*  赋值  */
        //  BASIC  中赋值语句不属于表达式!
        {2,  2,  2,  right2left,  oper_assignment},    //  赋值
        /*  最小优先级  */
        {2,  0,  0,  right2left,  oper_min}                //  栈底
};
```

你也许会问为什么需要 icp（incoming precedence）、isp（in-stack precedence） 两个优先级，现在不用着急，以后会详细解释！

后缀表达式

现在操作数（operand）和操作符（operator）都有了，一个表达式就是由它们组合构成的，我们就统称它们为标记（token）。在程序中定义如下：

[cpp] view plaincopyprint?

```
1. //  in  expression.h
2. typedef  enum  {
3.         token_operand  =  1,
4.         token_operator
5. }  token_type;
6. typedef  struct  {
7.         token_type  type;
8.         union  {
9.                 OPERAND  var;
10.                OPERATOR  ator;
11.        };
12.}  TOKEN;
13.typedef  struct  tlist  {
14.        TOKEN  token;
15.        struct  tlist  *next;
16.}  TOKEN_LIST,  *PTLIST;
```

我们平时习惯将表达式符写作：operand operator operand（比如 1+1），这是一个递归的定义，表达式本身也可作为操作数。像这种将操作符放在两个操作数之间的表达式称为中缀表达式，中缀表达式的好处是可读性强，操作数之间泾渭分明（尤其是手写体中）。但它有自身的缺陷：操作符的位置说明不了它在运算的先后问题。例如 1+2×3 中，虽然 + 的位置在 × 之前，但这并不表示先做加运算再做乘运算。为解决这个问题，数学中给操作符分了等级，级别高的操作符先计算（乘号的级别比加号高），并用**括号**提高操作符优先级。因此上例表达式的值是 7 而不是 (1+2)*3=9。

但对于计算机来说，优先级是一个多余的概念。就像上面提到的，中缀表达式中操作符的顺序没有提供运算先后关系的信息，这就好比用 4 个字节的空间仅保存 1 个字节数据——太浪费了！索性将操作符按照运算的先后排序：先计算的排最前面。此时操作符就不适合再放中间了，可以将它移到被操作数的后面：operand operand operator（比如 1 1 +）。上例中 1+2×3 就变化为 1 2 3 × +；(1+2)×3 变化成 1 2 + 3 ×，这种将操作符符放到操作数后面的表达式称为后缀表达式。同理还有将操作符符按照逆序放到操作数的前面的前缀表达式。

无论是前缀表达式还是后缀表达式，它们的优点都是用操作符的顺序来代替优先级，这样就可以舍弃括号等概念，化繁为简。

后缀表达式求值

请看下面的梯等式计算，比较中缀表达式和后缀表达式的求值过程。

```
  8 × ( 2 + 3 )        8 2 3 + ×
= 8 * 5              = 8 5 ×
= 40                 = 40
```

后缀表达式的求值方式：从头开始一个标记（token）一个标记地往后扫描，碰到操作数时先放到一个临时的空间里；碰到操作符就从空间里取出最后两个操作数，做相应的运算，然后将结果再次放回空间中。到了最后，空间中就只剩下操作数即运算结果！这个中缀表达式求值类似，只不过中缀表达式操作数取的是前后各一个。下面的代码是程序中后缀表达式求值的节选，其中只包含加法运算，其他运算都是类似的。

[cpp] view plaincopyprint?

```
1. //  in  expression.c
2. VARIANT  eval  (  const  char  expr[]  )
3. {
4.         //  ...
5.         //  一些变量的定义和声明
6.
7.         //  将中缀表达式转换成后缀表达式
8.         //  转换方法将在后续文章中介绍
9.         list  =  infix2postfix  ();
10.        while  (  list  )  {
11.                //  取出一个  token
12.                p  =  list;
13.                list  =  list->next;
14.
15.                //  如果是操作数就保存到  stack  中
16.                if  (  p->token.type  ==  token_operand  )  {
17.                        p->next  =  stack;
18.                        stack  =  p;
19.                        continue;
20.                }
21.
22.                //  如果是操作符...
23.                switch  (  p->token.ator.oper  )  {
24.                //  加法运算
25.                case  oper_plus:
26.                        //  取出  stack  中最末两个操作数
27.                        op2  =  stack;
28.                        op1  =  stack  =  stack->next;
29.
30.                        if  (  op1->token.var.type  ==  var_double  &&
31.                                op2->token.var.type  ==  var_double  )  {
32.                                op1->token.var.i  +=  op2->token.var.i;
33.                        }  else  {
34.                                //  字符串的加法即合并两个字符串
35.                                //  如果其中一个是数字则需要先转换为字符串
```

```
                                if ( op1->token.var.type == var_double ) {
                                        sprintf ( s1, "%g", op1->token.var.i
    );
                                } else {
                                        strcpy ( s1, op1->token.var.s );
                                }
                                if ( op2->token.var.type == var_double ) {
                                        sprintf ( s2, "%g", op2->token.var.i
    );
                                } else {
                                        strcpy ( s2, op2->token.var.s );
                                }
                                op1->token.type = var_string;
                                strcat ( s1, s2 );
                                strcpy ( op1->token.var.s, s1 );
                        }
                        free ( op2 );
                        break;
                //  ...
                //  其他操作符方法类似
                default:
                        //  无效操作符处理
                        break;
                }
                free ( p );
        }

        value = stack->token.var;
        free ( stack );

        //  最后一个元素即表达式的值
        return value;
}
```

总结

这一篇文章主要介绍了表达式中的操作符、操作数在程序内部的表示方法、后缀表达式的相关知识以及后缀表达式求值的方法。在下一篇文章中将着重介绍如何将中缀表达式转换成后缀表达式，请关注《用 C 语言写解释器（三）》。

用 C 语言写解释器（三）

声明

为提高教学质量，我所在的学院正在筹划编写 C 语言教材。《用 C 语言写解释器》系列文章经整理后将收入书中“综合实验”一章。因此该系列的文章主要阅读对象定为刚学完 C 语言的学生（不要求有数据结构等其他知识），所以行文比较罗嗦，请勿见怪。本人水平有限，如有描述不恰当或错误之处请不吝赐教！特此声明。

操作符排序

如果你忘记了后缀表达式的概念，赶紧翻回上一篇《用 C 语言写解释器（二）》回顾一下。简单地说，将中缀表达式转换成后缀表达式，就是将操作符的执行顺序由“优先级顺序”转换成“在表达式中的先后顺序”。因此，所谓的中缀转后缀，其实就是给原表达式中的操作符排序。

比如将中缀表达式 `5 * ((10 - 1) / 3)` 转换成后缀表达式为 `5 10 1 - 3 / *`。其中数字 `5 10 1 3` 仍然按照原先的顺序排列，而操作符的顺序变为 `- / ×`，这意味着减号最先计算、其次是除号、最后才是乘号。也许你还在担心如何将操作符从两个操作数的中间移到它们的后边。其实不用担心，在完成了排序工作后你就发现它已经跑到操作数的后面了 ^_^。

从中缀表达式 `1+2×3+4` 中逐个获取操作符，依次是 `+ × +`。如果当前操作符的优先级不大于前面的操作符时，前面操作符就要先输出。比如例子中的第二个加号，它前面是乘号，因此乘号从这个队伍中跑到输出的队伍中当了“老大”；此时第二个加号再前面的加号比较，仍然没有比它大，因此第一个加号也排到新队伍中去了；最后队伍中只剩下加号自己了，所以它也走了。得到新队伍里的顺序 `× + +` 就是所求解。下面的表格中详细展示每一个步骤。

序号输入临时空间 输出

1 +

2 × +

3 + + ×

4 + × +

5 + + ×

6 + × +

7 × + +

相信你心里还是牵挂着那些操作数。很简单，如果碰到的是操作符就按上面的规则处理，如果是操作数就直接输出！下面的表格加上了操作数，将输出完整的后缀表达式。

序号输入临时空间 输出

1 1

2 + 1

3 2 + 1

4 × + 1 2

5 3 + × 1 2

6 + + × 1 2 3

7 + × + 1 2 3

8 + + 1 2 3 ×

9 4 + 1 2 3 × +

10 + 1 2 3 × + 4

11 1 2 3 × + 4 +

得到最终结果 1 2 3 × + 4 + 就是所求的后缀表达式。下面是程序中的参考代码（有删减）。

[cpp] view plaincopyprint?

```
1. //  in  expression.c
2. PTLIST  infix2postfix  ()
3. {
4.         PTLIST  list  =  NULL,  tail,  p;
5.         PTLIST  stack  =  NULL;
6.         //  初始时在临时空间放一个优先级最低的操作符
7.         //  这样就不用判断是否为空了，方便编码
8.         stack  =  (PTLIST)calloc(1,  sizeof(TOKEN_LIST));
```

```
9.          stack->next  =  NULL;
10.         stack->token.type  =  token_operator;
11.         stack->token.ator  =  operators[oper_min];
12.         //  before  为全局变量，用于保存之前的操作符
13.         //  具体作用参看下面的章节
14.         memset  (  &before,  0,  sizeof(before)  );
15.         for  (;;)  {
16.                 p  =  (PTLIST)calloc(1,  sizeof(TOKEN_LIST));
17.                 //  calloc  自动初始化
18.                 p->next  =  NULL;
19.                 p->token  =  next_token  ();
20.                 if  (  p->token.type  ==  token_operand  )  {
21.                         //  如果是操作数，就不用客气，直接输出
22.                         if  (  !list  )  {
23.                                 list  =  tail  =  p;
24.                         }  else  {
25.                                 tail->next  =  p;
26.                                 tail  =  p;
27.                         }
28.                 }  else  if  (  p->token.type  ==  token_operator)  {
29.                         if  (  p->token.ator.oper  ==  oper_rparen  )  {
30.                                 //  右括号
31.                                 free  (  p  );
32.                                 while  (  stack->token.ator.oper  !=  oper_lparen
    )  {
33.                                         p  =  stack;
34.                                         stack  =  stack->next;
35.                                         tail->next  =  p;
36.                                         tail  =  p;
37.                                         tail->next  =  NULL;
38.                                 }
39.                                 p  =  stack;
40.                                 stack  =  stack->next;
41.                                 free  (  p  );
42.                         }  else  {
43.                                 while  (  stack->token.ator.isp  >=  p->token.ato
  r.icp  )  {
44.                                         tail->next  =  stack;
45.                                         stack  =  stack->next;
46.                                         tail  =  tail->next;
47.                                         tail->next  =  NULL;
48.                                 }
49.                                 p->next  =  stack;
50.                                 stack  =  p;
51.                         }
52.                 }  else  {
53.                         free  (  p  );
54.                         break;
55.                 }
```

```
56.         }
57.         while  (  stack  )  {
58.                 p  =  stack;
59.                 stack  =  stack->next;
60.                 if  (  p->token.ator.oper  !=  oper_min  )  {
61.                         p->next  =  NULL;
62.                         tail->next  =  p;
63.                         tail  =  p;
64.                 }  else  {
65.                         free  (  p  );
66.                 }
67.         }
68.         return  list;
69.}
```

操作符优先级

上一节介绍了中缀转后缀的方法。其中关键的部分就是比较两个操作符的优先级大小。通常情况下这都很简单：比如乘除的优先级比加减大，但括号需要特殊考虑。

中缀表达式中用括号来提升运算符的优先级，因此左括号正在放入（incoming）临时空间时优先级比任何操作符都大；一旦左括号已经放入（in-stack）空间中，此时它优先级如果还是最大，那无论什么操作符过来它就马上被踢出去，而我们想要的是任何操作符过来都能顺利放入临时空间，因此它放入空间后优先级需要变为最小。这意味着左括号在放入空间前后的优先级是不同的，所以我们需要用两个优先级变量 icp 和 isp 来分别记录操作符在两个状态下的优先级（还记得上一篇的问题吗）。

另一个是右括号，它本身和优先级无关，它会将临时空间里的操作符一个个输出，直到碰到左括号为止。下面是本程序中中缀转后缀的代码（有删减）。

获取标识符

在上面的代码中你会看到一个陌生的函数 next_taken() 。它会从中缀表达式中获得一个标记，方法类似从字符串中提取单词（参看课后习题）。在本程序中能识别的标记除了操作符，还有纯数字、字符串、变量名等操作数。唯一要注意的就是操作符和操作数之间可以存在零到多个空格。下面是参考代码（有删减）。

[cpp] view plaincopyprint?

```
1. //  in  expression.c
2. static  TOKEN  next_token  ()
3. {
4.         TOKEN  token  =  {0};
5.         STRING  s;
6.         int  i;
```

```
7.          if  (  e  ==  NULL  )  {
8.                  return  token;
9.          }
10.         //  去掉前导空格
11.         while  (  *e  &&  isspace(*e)  )  {
12.                 e++;
13.         }
14.         if  (  *e  ==  0  )  {
15.                 return  token;
16.         }
17.         if  (  *e  ==  '"'  )  {
18.                 //  字符串
19.                 token.type  =  token_operand;
20.                 token.var.type  =  var_string;
21.                 e++;
22.                 for  (  i  =  0;  *e  &&  *e  !=  '"';  i++  )  {
23.                         token.var.s[i]  =  *e;
24.                         e++;
25.                 }
26.                 e++;
27.         }  else  if  (  isalpha(*e)  )  {
28.                 //  如果首字符为字母则有两种情况
29.                 //  1.  变量
30.                 //  2.  逻辑操作符
31.                 token.type  =  token_operator;
32.                 for  (  i  =  0;  isalnum(*e);  i++  )  {
33.                         s[i]  =  toupper(*e);
34.                         e++;
35.                 }
36.                 s[i]  =  0;
37.                 if  (  !strcmp  (  s,  "AND"  )  )  {
38.                         token.ator  =  operators[oper_and];
39.                 }  else  if  (  !strcmp  (  s,  "OR"  )  )  {
40.                         token.ator  =  operators[oper_or];
41.                 }  else  if  (  !strcmp  (  s,  "NOT"  )  )  {
42.                         token.ator  =  operators[oper_not];
43.                 }  else  if  (  i  ==  1  )  {
44.                         token.type  =  token_operand;
45.                         token.var  =  memory[s[0]-'A'];
46.                         if  (  token.var.type  ==  var_null  )  {
47.                                 memset  (  &token,  0,  sizeof(token)  );
48.                                 fprintf  (  stderr,  "变量%c 未赋值!
   /n",  s[0]  );
49.                                 exit  (  EXIT_FAILURE  );
50.                         }
51.                 }  else  {
52.                         goto  errorhandler;
53.                 }
54.         }  else  if  (  isdigit(*e)  ||  *e  ==  '.'  )  {
```

```
                // 数字
                token.type = token_operand;
                token.var.type = var_double;
                for ( i = 0; isdigit(*e) || *e == '.'; i++ ) {
                        s[i] = *e;
                        e++;
                }
                s[i] = 0;
                if ( sscanf ( s, "%lf", &token.var.i ) != 1 ) {
                        // 读取数字失败!
                        // 错误处理
                }
        } else {
                // 剩下算数运算符和关系运算符
                token.type = token_operator;
                switch (*e) {
                case '(':
                        token.ator = operators[oper_lparen];
                        break;
                // ...
                // 此处省略其他操作符的代码
                default:
                        // 不可识别的操作符
                        break;
                }
                e++;
        }
        before = token;
        return token;
}
```

总结

本章主要介绍中缀表达式转后缀表达式的方法，并给出了相应的参考代码。和前一篇文章结合起来就完成了解释器中"表达式求值"和"内存管理"两部分，在下一篇文章中我们将介绍语句的解析，其中包含了输入/输出、分支以及循环语句，请关注《用 C 语言写解释器（四）》。

用 C 语言写解释器（四）

声明

为提高教学质量，我所在的学院正在筹划编写 C 语言教材。《用 C 语言写解释器》系列文章经整理后将收入书中"综合实验"一章。因此该系列的文章主要阅读对象定为刚学完 C 语言的学生（不要求有数据结构等其他知识），所以行文比较罗嗦，请勿见怪。本人水平有限，如有描述不恰当或错误之处请不吝赐教！特此声明。

语句

在前面的章节中已经成功实现了内存管理和表达式求值模块。之所以称表达式求值是解释器的核心部分，是因为几乎所有语句的操作都伴随着表达式求值。也许你已经迫不及待地给 eval 传值让它执行复杂的运输了，但目前来讲它充其量只是一个计算器。要想成为一门语言，还需要一套自成体系的语法，包括输入输出语句和控制语句。但在进行语法分析之前，首先需要将 BASIC 源码载入到内存中。

BASIC 源码载入

在《用 C 语言写解释器（一）》中附了一段 BASIC 参考代码，每一行的结构是一个行号+一条语句。其中行号为 1-9999 之间的正整数，且当前行号大于前面的行号；语句则由以下即将介绍的 3 条 I/O 语句和 8 条控制语句组成。为方便编码，程序中采用静态数组来保存源代码，读者可以尝试用链表结构实现动态申请的版本。下面是代码结构的定义。

[cpp] view plaincopyprint?

```
1. //  in  basic_io.h
2. #define  PROGRAM_SIZE  (10000)
3.
4. typedef  struct  {
5.         int  ln;          //  line  number
6.         STRING  line;
7. }  CODE;
```

```
8.
9. extern  CODE  code[PROGRAM_SIZE];
10.extern  int  cp;
11.extern  int  code_size;
```

其中 code_size 的作用顾名思义：记录代码的行数。cp （0 ≤ cp < code_size）记录当前行的下标（比如 cp 等于 5 时表明执行到第 5 行）。下面是载入 BASIC 源码的参考代码，在载入源码的同时会去除两端的空白字符。

[cpp] view plaincopyprint?

```
1. //  in  basic_io.c
2. void  load_program  (  STRING  filename  )
3. {
4.         FILE  *fp  =  fopen  (  filename,  "r"  );
5.         int  bg,  ed;
6.
7.         if  (  fp  ==  NULL  )  {
8.                 fprintf  (  stderr,  "文件  %s  无法打开！/n",  filename  );
9.                 exit  (  EXIT_FAILURE  );
10.        }
11.
12.        while  (  fscanf  (  fp,  "%d",  &code[cp].ln  )  !=  EOF  )  {
13.                if  (  code[cp].ln  <=  code[cp-1].ln  )  {
14.                        fprintf  (  stderr,  "Line  %d:  标号错误！
   /n",  cp  );
15.                        exit  (  EXIT_FAILURE  );
16.                }
17.
18.                fgets  (  code[cp].line,  sizeof(code[cp].line),  fp  );
19.                for  (  bg  =  0;  isspace(code[cp].line[bg]);  bg++  );
20.                ed  =  (int)strlen  (  code[cp].line  +  bg  )  -  1;
21.                while  (  ed  >=  0  &&  isspace  (  code[cp].line[ed+bg]  )  )
   {
22.                        ed--;
23.                }
24.                if  (  ed  >=  0  )  {
25.                        memmove  (  code[cp].line,  code[cp].line  +  bg,  ed  +
     1  );
26.                        code[cp].line[ed  +  1]  =  0;
27.                }  else  {
28.                        code[cp].line[0]  =  0;
29.                }
30.
31.                cp++;
32.                if  (  cp  >=  PROGRAM_SIZE  )  {
33.                        fprintf  (  stderr,  "程序%s 太大，代码空间不足！
   /n",  filename  );
```

```
34.                         exit  (  EXIT_FAILURE  );
35.                 }
36.         }
37.
38.         code_size  =  cp;
39.         cp  =  1;
40.}
```

语法分析

源码载入完成后就要开始逐行分析语句了，程序中总共能处理以下 11 种语句：

[cpp] view plaincopyprint?

```
1. //  in  main.c
2. typedef  enum  {
3.         key_input  =  0,    //  INPUT
4.         key_print,            //  PRINT
5.         key_for,               //  FOR  ..  TO  ..  STEP
6.         key_next,             //  NEXT
7.         key_while,            //  WHILE
8.         key_wend,             //  WEND
9.         key_if,                 //  IF
10.        key_else,             //  ELSE
11.        key_endif,            //  END  IF
12.        key_goto,             //  GOTO
13.        key_let                 //  LET
14.}  keywords;
```

《用 C 语言写解释器（一）》中详细描述了每个语句的语法，本程序中所谓的语法其实就是字符串匹配，参考代码如下：

[cpp] view plaincopyprint?

```
1. //  in  main.c
2. keywords  yacc  (  const  STRING  line  )
3. {
4.         if  (  !strnicmp  (  line,  "INPUT  ",  6  )  )  {
5.                 return  key_input;
6.         }  else  if  (  !strnicmp  (  line,  "PRINT  ",  6  )  )  {
7.                 return  key_print;
8.         }  else  if  (  !strnicmp  (  line,  "FOR  ",  4  )  )  {
9.                 return  key_for;
10.        }  else  if  (  !stricmp  (  line,  "NEXT"  )  )  {
11.                return  key_next;
12.        }  else  if  (  !strnicmp  (  line,  "WHILE  ",  6  )  )  {
13.                return  key_while;
```

```
14.         } else if ( !stricmp ( line, "WEND" ) ) {
15.                 return key_wend;
16.         } else if ( !strnicmp ( line, "IF ", 3 ) ) {
17.                 return key_if;
18.         } else if ( !stricmp ( line, "ELSE" ) ) {
19.                 return key_else;
20.         } else if ( !stricmp ( line, "END IF" ) ) {
21.                 return key_endif;
22.         } else if ( !strnicmp ( line, "GOTO ", 5 ) ) {
23.                 return key_goto;
24.         } else if ( !strnicmp ( line, "LET ", 4 ) ) {
25.                 return key_let;
26.         } else if ( strchr ( line, '=' ) ) {
27.                 return key_let;
28.         }
29.
30.         return -1;
31.}
```

每个语句对应有一个执行函数，在分析出是哪种语句后，就可以调用它了！为了编码方便，我们将这些执行函数保存在一个函数指针数组中，请看下面的参考代码：

[cpp] view plaincopyprint?

```
1. // in main.c
2. void (*key_func[])( const STRING ) = {
3.         exec_input,
4.         exec_print,
5.         exec_for,
6.         exec_next,
7.         exec_while,
8.         exec_wend,
9.         exec_if,
10.        exec_else,
11.        exec_endif,
12.        exec_goto,
13.        exec_assignment
14.};
15.
16.int main ( int argc, char *argv[] )
17.{
18.         if ( argc != 2 ) {
19.                 fprintf ( stderr, "usage: %s basic_script_file/n", argv[0]
    );
20.                 exit ( EXIT_FAILURE );
21.         }
22.
23.         load_program ( argv[1] );
24.
25.         while ( cp < code_size ) {
```

```
26.                 (*key_func[yacc ( code[cp].line )]) ( code[cp].line );
27.                 cp++;
28.         }
29.
30.         return EXIT_SUCCESS;
31.}
```

以上代码展示的就是整个程序的基础框架，现在欠缺的只是每个语句的执行函数，下面将逐个详细解释。

I/O 语句

输入输出是一个宽泛的概念，并不局限于从键盘输入和显示到屏幕上，还包括操作文件、连接网络、进程通信等。《我们的目标》中指出只需实现从键盘输入（INPUT）和显示到屏幕上（PRINT），事实上还应该包括赋值语句，只不过它属于程序内部的 I/O。

INPUT 语句

INPUT 语句后面跟着一堆变量名（用逗号隔开）。因为变量是弱类型，你可以输入数字或字符串。但 C 语言是强类型语言，为实现这个功能就需要判断一下 scanf 的返回值。我们执行 scanf ("%lf", &memory[n].i)，如果你输入的是一个数字，就能成功读取一个浮点数，函数返回 1、否则就返回 0；不能读取时就采用 getchar 来获取字符串！参考代码如下：

[cpp] view plaincopyprint?

```
1. //  in  basic_io.c
2. void  exec_input  (  const  STRING  line  )
3. {
4.         const  char  *s  =  line;
5.         int  n;
6.
7.         assert  (  s  !=  NULL  );
8.         s  +=  5;
9.
10.        while  (  *s  )  {
11.                while  (  *s  &&  isspace(*s)  )  {
12.                        s++;
13.                }
14.                if  (  !isalpha(*s)  ||  isalnum(*(s+1))  )  {
15.                        perror  (  "变量名错误！/n"  );
16.                        exit  (  EXIT_FAILURE  );
17.                }  else  {
```

```
                        n = toupper(*s) - 'A';
                }

                if ( !scanf ( "%lf", &memory[n].i ) ) {
                        int i;
                        // 用户输入的是一个字符串
                        memory[n].type = var_string;
                        if ( (memory[n].s[0] = getchar()) == '"' ) {
                                for ( i = 0; (memory[n].s[i]=getchar())!='"' ; i++ );
                        } else {
                                for ( i = 1; !isspace(memory[n].s[i]=getchar()); i++ );
                        }
                        memory[n].s[i] = 0;
                } else {
                        memory[n].type = var_double;
                }

                do {
                        s++;
                } while ( *s && isspace(*s) );
                if ( *s && *s != ',' ) {
                        perror ( "INPUT 表达式语法错误!/n" );
                        exit ( EXIT_FAILURE );
                } else if ( *s ) {
                        s++;
                }
        }
}
```

PRINT 语句

输出相对简单些，PRINT 后面跟随的是一堆表达式，表达式只需委托给 eval 来求值即可，因此 PRINT 要做的仅仅是按照值的类型来输出结果。唯一需要小心的就是类似 PRINT "hello, world" 这样字符串中带有逗号的情况，以下是参考代码：

[cpp] view plaincopyprint?

```
// in basic_io.c
void exec_print ( const STRING line )
{
        STRING l;
        char *s, *e;
        VARIANT v;
        int c = 0;

```

```
9.          strcpy ( l, line );
10.         s = l;
11.
12.         assert ( s != NULL );
13.         s += 5;
14.
15.         for (;;) {
16.                 for ( e = s; *e && *e != ','; e++ ) {
17.                         // 去除字符串
18.                         if ( *e == '"' ) {
19.                                 do {
20.                                         e++;
21.                                 } while ( *e && *e != '"' );
22.                         }
23.                 }
24.                 if ( *e ) {
25.                         *e = 0;
26.                 } else {
27.                         e = NULL;
28.                 }
29.
30.                 if ( c++ ) putchar ( '/t' );
31.                 v = eval ( s );
32.                 if ( v.type == var_double ) {
33.                         printf ( "%g", v.i );
34.                 } else if ( v.type == var_string ) {
35.                         printf ( v.s );
36.                 }
37.
38.                 if ( e ) {
39.                         s = e + 1;
40.                 } else {
41.                         putchar ( '/n' );
42.                         break;
43.                 }
44.         }
45.}
```

LET 语句

在 BASIC 中，“赋值”和“等号”都使用“=”，因此不能像 C 语言中使用 A = B = C 这样连续赋值，在 BASIC 中它的意思是判断 B 和 C 的值是否相等并将结果赋值给 A 。而且关键字 LET 是可选的，即 LET A = 1 和 A = 1 是等价的。剩下的事情那个就很简单了，只要将表达式的值赋给变量即可。以下是参考代码：

[cpp] view plaincopyprint?

```
1. //  in  basic_io.c
2. void  exec_assignment  (  const  STRING  line  )
3. {
4.         const  char  *s  =  line;
5.         int  n;
6.
7.         if  (  !strnicmp  (  s,  "LET  ",  4  )  )  {
8.                 s  +=  4;
9.         }
10.        while  (  *s  &&  isspace(*s)  )  {
11.                s++;
12.        }
13.        if  (  !isalpha(*s)  ||  isalnum(*(s+1))  )  {
14.                perror  (  "变量名错误! /n"  );
15.                exit  (  EXIT_FAILURE  );
16.        }  else  {
17.                n  =  toupper(*s)  -  'A';
18.        }
19.
20.        do  {
21.                s++;
22.        }  while  (  *s  &&  isspace(*s)  );
23.        if  (  *s  !=  '='  )  {
24.                fprintf  (  stderr,  "赋值表达式  %s  语法错误!
   /n",  line  );
25.                exit  (  EXIT_FAILURE  );
26.        }  else  {
27.                memory[n]  =  eval  (  s  +  1  );
28.        }
29.}
```

控制语句

现在是最后一个模块——控制语句。控制语句并不参与交互，它们的作用只是根据一定的规则来改变代码指针（cp）的值，让程序能到指定的位置去继续执行。限于篇幅，本节只介绍 for、next 以及 goto 三个控制语句的实现方法，读者可以尝试自己完成其他函数，也可以参看附带的完整代码。

FOR 语句

先来看一下 FOR 语句的结构：

FOR var = expression1 TO expression2 [STEP expression3]

它首先要计算三个表达式，获得 v1、v2、v3 三个值，然后让变量（var）从 v1 开始，每次迭代都加 v3，直到超出 v2 的范围位置。因此，每一个 FOR 语句，我们都需要保存这四个信息：变量名、起始值、结束值以及步长。另外，不要忘记 FOR 循环等控制语句可以嵌套使用，因此需要开辟一组空间来保存这些信息，参考代码如下：

[cpp] view plaincopyprint?

```
1. //  in  grammar.h
2. static  struct  {
3.         int  id;                    //  memory  index
4.         int  ln;                    //  line  number
5.         double  target;        //  target  value
6.         double  step;
7. }  stack_for[MEMORY_SIZE];
8. static  int  top_for  =  -1;
```

分析的过程就是通过 strstr 在语句中搜索"="、"TO"、"STEP"等字符串，然后将提取的表达式传递给 eval 计算，并将值保存到 stack_for 这个空间中。参考代码如下：

[cpp] view plaincopyprint?

```
1. //  in  grammar.c
2. void  exec_for  (  const  STRING  line  )
3. {
4.         STRING  l;
5.         char  *s,  *t;
6.         int  top  =  top_for  +  1;
7.
8.         if  (  strnicmp  (  line,  "FOR  ",  4  )  )  {
9.                 goto  errorhandler;
10.        }  else  if  (  top  >=  MEMORY_SIZE  )  {
11.                fprintf  (  stderr,  "FOR  循环嵌套过深！/n"  );
12.                exit  (  EXIT_FAILURE  );
13.        }
14.
15.        strcpy  (  l,  line  );
16.
17.        s  =  l  +  4;
18.        while  (  *s  &&  isspace(*s)  )  s++;
19.        if  (  isalpha(*s)  &&  !isalnum(s[1])  )  {
20.                stack_for[top].id  =  toupper(*s)  -  'A';
21.                stack_for[top].ln  =  cp;
22.        }  else  {
23.                goto  errorhandler;
24.        }
```

```
25.
26.         do  {
27.                 s++;
28.         }  while  (  *s  &&  isspace(*s)  );
29.         if  (  *s  ==  '='  )  {
30.                 s++;
31.         }  else  {
32.                 goto  errorhandler;
33.         }
34.
35.         t  =  strstr  (  s,  "  TO  "  );
36.         if  (  t  !=  NULL  )  {
37.                 *t  =  0;
38.                 memory[stack_for[top].id]  =  eval  (  s  );
39.                 s  =  t  +  4;
40.         }  else  {
41.                 goto  errorhandler;
42.         }
43.
44.         t  =  strstr  (  s,  "  STEP  "  );
45.         if  (  t  !=  NULL  )  {
46.                 *t  =  0;
47.                 stack_for[top].target  =  eval  (  s  ).i;
48.                 s  =  t  +  5;
49.                 stack_for[top].step  =  eval  (  s  ).i;
50.                 if  (  fabs  (  stack_for[top].step  )  <  1E-6  )  {
51.                         goto  errorhandler;
52.                 }
53.         }  else  {
54.                 stack_for[top].target  =  eval  (  s  ).i;
55.                 stack_for[top].step  =  1;
56.         }
57.
58.         if  (  (stack_for[top].step  >  0  &&
59.                   memory[stack_for[top].id].i  >  stack_for[top].target)||
60.                   (stack_for[top].step  <  0  &&
61.                   memory[stack_for[top].id].i  <  stack_for[top].target))  {
62.                 while  (  cp  <  code_size  &&  strcmp(code[cp].line,  "NEXT")  )
    {
63.                         cp++;
64.                 }
65.                 if  (  cp  >=  code_size  )  {
66.                         goto  errorhandler;
67.                 }
68.         }  else  {
69.                 top_for++;
70.         }
71.
72.         return;
```

```
73.
74.errorhandler:
75.        fprintf  (  stderr,  "Line  %d:  语法错误！/n",  code[cp].ln  );
76.        exit  (  EXIT_FAILURE  );
77.}
```

NEXT 语句

NEXT 的工作就简单得多了。它从 stack_for 这个空间中取出最后一组数据，让变量的值累加上步长，并判断循环是否结束。如果结束就跳出循环执行下一条语句；否则就将代码指针移回循环体的顶部，继续执行循环体。下面是参考代码。

[cpp] view plaincopyprint?

```
1. //  in  grammar.c
2. void  exec_next  (  const  STRING  line  )
3. {
4.         if  (  stricmp  (  line,  "NEXT"  )  )  {
5.                 fprintf  (  stderr,  "Line  %d:  语法错误！
   /n",  code[cp].ln  );
6.                 exit  (  EXIT_FAILURE  );
7.         }
8.         if  (  top_for  <  0  )  {
9.                 fprintf  (  stderr,  "Line  %d:  NEXT  没有相匹配的  FOR！
   /n",  code[cp].ln  );
10.                exit  (  EXIT_FAILURE  );
11.        }
12.
13.        memory[stack_for[top_for].id].i  +=  stack_for[top_for].step;
14.        if  (  stack_for[top_for].step  >  0  &&
15.                  memory[stack_for[top_for].id].i  >  stack_for[top_for].target
   )  {
16.                top_for--;
17.        }  else  if  (  stack_for[top_for].step  <  0  &&
18.                  memory[stack_for[top_for].id].i  <  stack_for[top_for].target
   )  {
19.                top_for--;
20.        }  else  {
21.                cp  =  stack_for[top_for].ln;
22.        }
23.}
```

GOTO 语句

也许你认为 GOTO 语句只是简单的将 cp 的值设置为指定的行，但事实上它比想象中的要复杂些。考虑下面的 BASIC 代码：

```
0010 I = 5
0020 GOTO 40
0030 FOR I = 1 TO 10
0040   PRINT I
0050 NEXT
```

像这类代码，直接跳到循环体内部，如果只是简单地将 cp 移动到指定位置，当代码继续执行到 NEXT 时就会报告没有对应的 FOR 循环！跳到其他的控制结构，如 WHILE、IF 等，也会出现相同的问题。以下是参考代码（有删减）。

[cpp] view plaincopyprint?

```
1. //  in  grammar.c
2. void  exec_goto  (  const  STRING  line  )
3. {
4.         int  ln;
5.
6.         if  (  strnicmp  (  line,  "GOTO  ",  5  )  )  {
7.                 fprintf  (  stderr,  "Line  %d:  语法错误!
   /n",  code[cp].ln  );
8.                 exit  (  EXIT_FAILURE  );
9.         }
10.
11.        ln  =  (int)eval  (  line  +  5  ).i;
12.        if  (  ln  >  code[cp].ln  )  {
13.                //  往下跳转
14.                while  (  cp  <  code_size  &&  ln  !=  code[cp].ln  )  {
15.                        if  (  !strnicmp  (  code[cp].line,  "IF  ",  3  )  )  {

16.                                top_if++;
17.                                stack_if[top_if]  =  1;
18.                        }  else  if  (  !stricmp  (  code[cp].line,  "ELSE"  )  )
    {
19.                                stack_if[top_if]  =  1;
20.                        }  else  if  (  !stricmp  (  code[cp].line,  "END  IF"  )
    )  {
21.                                top_if--;
22.                        }  else  if  (  !strnicmp  (  code[cp].line,  "WHILE  ",
    6  )  )  {
23.                                top_while++;
24.                                stack_while[top_while].isrun  =  1;
25.                                stack_while[top_while].ln  =  cp;
```

```
26.                     } else if ( !stricmp ( code[cp].line, "WEND" ) )
    {
27.                         top_while--;
28.                     } else if ( !strnicmp ( code[cp].line, "FOR ", 4
    ) ) {
29.                         int i = 4;
30.                         VARIANT v;
31.                         while ( isspace(code[cp].line[i]) ) i++;
32.                         v = memory[toupper(code[cp].line[i])-'A'];
33.                         exec_for ( code[cp].line );
34.                         memory[toupper(code[cp].line[i])-'A'] = v;
35.                     } else if ( !stricmp ( code[cp].line, "NEXT" ) )
    {
36.                         top_for--;
37.                     }
38.                     cp++;
39.                 }
40.         } else if ( ln < code[cp].ln ) {
41.                 // 往上跳转
42.                 // 代码类似，此处省略
43.         } else {
44.                 // 我不希望出现死循环，你可能有其他处理方式
45.                 fprintf ( stderr, "Line %d: 死循环!
   /n", code[cp].ln );
46.                 exit ( EXIT_FAILURE );
47.         }
48.
49.         if ( ln == code[cp].ln ) {
50.                 cp--;
51.         } else {
52.                 fprintf ( stderr, "标号 %d 不存在! /n", ln );
53.                 exit ( EXIT_FAILURE );
54.         }
55.}
```

总结

本章介绍了源码载入、语法分析以及部分语句的实现，WHILE 和 IF 等控制语句方法和 FOR、NEXT 类似，有兴趣的读者请尝试自己实现（或者参看附带的完整源码）。这样一个解释器的四个关键部分"内存管理"、"表达式求值"、"输入输出"和"控制语句"就全部介绍完了，希望你也能写出自己的解释器。下一篇我将总结一下我个人对编程语言的一些思考，如果你也有兴趣请继续关注《用 C 语言写解释器（五）》！

用 C 语言写解释器（五）

写完解释器之后

这一篇文章我只想和大家侃侃编程语言的事情，不会被放到书中。因此可以天南地北地扯淡，不用像前几篇一样畏首畏尾的了。

经过前面几篇文章的讨论，已经把用纯 C 语言来实现一个解释器的方法介绍完了。但那些是写给我校 C 语言初学者看的，并不只是你，我得也觉得很不过瘾 ^_^。因此准备继续深入学习编译原理等课程，希望有志同道合的朋友和我一起交流！

富饶的语言（工具）

在前几篇文章中一直在鼓吹我拍脑袋想出的语言四大要素：“内存管理”、“表达式求值”、“输入/输出”、“按条件跳转”，在这篇文章中您就姑且信一回当它是真的。按照这四条准则去匹配，汇编语言是完全符合的。那为什么又需要 C 语言、Java、C# 等高级语言？这是因为编程除了需要“语言”之外还需要“抽象”！

“抽象”是个很有效的工具，相信你在为别人介绍自己房间时不会具体到每个木纤维、油漆分子和铁原子。同样的，我们也不乐意总是写一堆 JNZ、JMP 指令，而仅仅是为了实现 if、for、while 等控制结构。C 语言等高级语言提供的抽象的层次更高、表现力更强，允许用更少的语句描述更多的操作。感谢如此富饶的语言为我们带来不同的视角去审视这个世界。
高级语言相较于低级语言属于更高地抽象层次，高级语言之间的差别主要体现在适用范围上。比如一些语言适合写 WEB 程序，另一些适合做数值分析等。术业有专攻，你只需根据自己的问题来选择一门合适的语言。

什么时候需要创造新的语言

当我们碰到一类新的问题时，首先考虑的就是定义新的数据结构，并设计多个函数去操作它，最后将它们独立出来打包成一个类库方便在其他地方调用（比如处理图形图像的 OpenGL 库）。上面已

经提过，每种语言都有它适合的领域，强行将一门语言用在它不擅长的领域中就出现冗长、繁琐的代码。自然语言也是如此：英语中有种语法叫虚拟语气，描述的是一种假设，并非事实。比如“If I have time, I will go to see you. ”。如果按原意一字不差地翻译相信会很繁琐，我知道台湾作家痞子蔡在使用中文式的虚拟语气很有一套：

如果我还能活一天，
我就要做你的爱侣。
我能活一天吗？可惜。
所以我不是你的爱侣。 ——《第一次亲密接触》

上面是一段完整表现虚拟语气精髓的话，相信在生活中我们不会这么罗嗦。同样的，如果你发现用现有语言来描述某个特定领域问题时显得力不从心，就可以考虑为这个领域定制一种特定的语言了（Domain Specific Language）！使用现成的词法分析器和语法分析器（比如 lex 和 yacc）对提高开发效率很有帮助，但你也可以考虑采用像 REBOL 这样的语言设计一个“方言”，这会更简单。如果你对 DSL 或 REBOL 有兴趣，可以加入阿里旺旺 REBOL 群（16626148）和蔡学镛前辈交流，他是这方面的专家。

从语言（工具）中挣脱

从写解释器这件事中可以获得一些建议：不要再争论哪个语言更优秀，只有最适合的；用高级语言写代码首先力求可读性好。第一条建议我在以前讨论“工具理论”时提过很多次，就不再重复，主要交流一下可读性的问题。

经过了上面冗长的解释和亲自实现解释器以后，大家应该能了解到：一门新语言诞生的动机多数情况下不是为了提高执行效率，而是为了提高开发效率。很多人都沉浸在“++i”比“i++”高效、“10>>1”比“10/2”快等奇技淫巧中。像我以前玩 ACM 时，一心只想着迷人的“0k 0ms”，代码写得它认识我我不认识它。就像《求质数之筛法》一文中的程序，我多少次想把这篇文章删了，免得丢人现眼，但最终还是决定留下，时刻提醒自己不要写如此招人诟病的代码！

在你自己实现过解释器后希望也能明白，如果真有哪个解释器执行语句“i++;”的效率比“++i;”低，那只能说明这个解释器写得烂！像现代的 C 语言编译器都会有优化的选项，编译时去识别一些常见的热点进行优化，难保那些自以为是的优化反而将代码破坏得连编译器也无法识别。所以要迁就解释器而将代码改得乱七八糟，我宁可换一个更好的解释器！

真的想深入研究算法，就势必会和硬件相关。你需要精确地知道代码一共执行了多少个时钟周期，而不是简单地根据嵌套了几层 FOR 循环来判断复杂度是 O(n) 还是 $O(n^2)$。除非你深入了解你的解释器，否则无从知晓执行一条 FOR 语句时解释器会不会背着你扫描了整个内存空间。无怪乎经典巨著《计算机程序设计艺术》三卷本中要使用汇编语言来编写代码。

总结

废话了这么多，我只是想表达“我们是主人”，不要被一个蹩脚的工具牵着鼻子走。当你发现打字员平均打字速度慢时，总不会为了迁就她而只说一些她打得快的字吧？以上内容属于个人观点，切莫认真。欢迎大家通过邮件和我交流你们的想法，我的邮箱地址：redraiment@gmail.com。