关于以源代码为输入的一类问题的初步探索

宁波镇海蛟川书院 卢啸尘

摘 要

在ACM-ICPC竞赛中，时常会有以源代码为输入的问题作为考察选手综合代码能力的问题而出现。本文介绍了解决此类问题时的一些实现技巧，并将此类问题与OI竞赛中常用的数据结构相结合，提出了数据结构题的一种新表现形式。

1 引言

在信息学竞赛中，大部分试题都可以被归为想法题、算法题和代码题这三类问题之一。想法题考察选手的创新能力，算法题考察选手的知识面，而代码题考察选手的基本功，包括代码能力和后续的调试能力。OI中比较经典的代码题除了高级数据结构题以外，还有山东省选的《猪国杀》，浙江省选的《杀蚂蚁》等。作者认为，这类以游戏作为背景的问题，存在着部分没有接触过有关游戏的选手无法准确理解题面的弊端。以源代码为输入的问题，其模型贴近选手的日常训练，就不存在这样的弊端，是代码题命题的一种良好选择。

作者整理了一些以源代码为输入的问题，将其分为三类，并对其进行了一些总结。本文的框架如下：

第二节中，介绍了一些预备知识：编译器的结构、程序设计语言的实现方式，以及符号约定。

第三节中，给出了一个基础问题。该问题的特点是不需要对于源代码的具体语义进行分析，而只需要分析代码结构。此节中介绍了词法分析器的实现。

第四节中，给出了几个实现难度较大的问题。这类问题要求对给出的程序代码进行模拟。此节中介绍了程序设计语言的混合实现系统的一种比较方便的代码实现。

第五节是本文的重点，在此节中，作者给出了几个变种问题。这类问题以消除不可达代码为模型，将数据结构和代码分析结合起来。

第五节中的所有题目具有同一个性质：“有修改，有回退修改”，这是一个弱于”可持久化”和”有修改、有取消修改”而强于”有修改”的性质。以这类问题为启发，我们可以设计一些试题（甚至是并非编译类问题的问题），它们具有明显强于单纯”有修改”的要求，又没有现成的优秀解法。而在降低要求到这一要求后就可以设计出高效的算法。

2 预备知识

2.1 编译器的结构

一个编译器的工作是，将一个高级语言的源代码映射到另一种高级语言或机器语言的代码。因此编译器被分为前端和后端两个部分。

前端被称为分析(analysis)部分。这个部分的工作是，将源代码解析成基本单元，并根据语法创建一个中间表示，中间表示通常不是另一种高级语言，因为高级语言难以被直接处理，也不是一种机器码，因为机器码和代码差的太远。常见的中间表示有“三地址码”等。

后端被称为综合(synthesis)部分。这个部分的工作是，根据前端传回的各要素和中间表示，创建目标语言的代码。根据编译器目标语言的不同，这里可能对中间表示进行进一步简化而形成机器语言，也可能生成一个高级语言的源代码。

每个部分又可以被分为若干步骤。前端包括词法分析、语法分析、语义分析、中间代码生成四个步骤。后端包括代码生成这一个步骤。有的编译器会在前端和后端之间设置一个机器无关代码优化步骤，或在后端代码生成步骤后增加一个机器有关代码优化步骤。

至于预处理器、汇编器和链接器，一般不认为其是编译器的一部分。

下面将说明这五个步骤的大致工作。以下图语句为例，其中A是double类型，B是int类型。

```
A = A + B * 123
```

2.1.1 词法分析

词法分析器接受一个字符串（即源代码）作为其输入，输出一个词法单元（词素）列表。

在上面的例子中，源代码被分为以下词素：$\{A, =, A, +, B, *, 123\}$。词素$A$与$B$被识别为变量标识符，词素=，+和∗被识别为算符，123被识别为整数字面值。词法分析器的输出如下:

$$
\begin{aligned}
&Identifier \quad id = 1\\
&\ Assign\\
&Identifier \quad id = 1\\
&Operator \quad \text{”} + \text{”}\\
&Identifier \quad id = 2\\
&Operator \quad \text{”} * \text{”}\\
&\ Literal \quad 123
\end{aligned}
$$

在编译器发展的早期阶段，词法分析器是作为一个独立的部分工作，它一次性地处理整个源文件。但是现在词法分析器通常是作为语法分析器的子程序而出现：语法分析器在工作时不时调用词法分析器，词法分析器从输入流中读取一系列字符直到构成了一个词素，词法分析器每次向它的调用者——语法分析器返回接下来的一个词素。也就是说，以上表为例，第一次调用词法分析器返回上表第一行的内容，第二次是第二行，以此类推。

2.1.2 语法分析

语法分析器接受一系列词素，分析其是否在语法上是正确的。然后如果是正确的，语法分析器返回一棵语法生成树。

```
STATEMENT
  LVALUE
    IDENTIFIER: Identifier [1]
  ASSIGN: Assign
  EXPRESSION
    EXPRESSION
      IDENTIFIER: Identifier [1]
    OPERATOR: Operator "+"
    EXPRESSION
      EXPRESSION
        IDENTIFIER: Identifier [2]
      OPERATOR: Operator "*"
      EXPRESSION
        LITERAL: Literal(Integer) 123
```

（上表并没有明确地指出语法分析器是如何处理不同运算符的优先级的，

这是因为此表只是用来说明语法分析器的大致任务）

虽然在实践中这棵语法生成树并不显式地建出，但是一个语法分析器需要强到可以建出一棵语法生成树，这就是说，至少可以追踪这棵语法生成树的结构。

大多数语言现象都可以使用上下文无关文法描述，因此我们有一个普适的三次方的算法，即CYK算法。它的效率不高。但是计算机科学家们牺牲了普适性换取了效率：对于上下文无关文法的某些特定子集，存在线性的算法。我们将在后面的章节中介绍其细节。

2.1.3 语义分析

语义分析器在语法分析树上对程序的语义进行审查。如类型检查。在这一阶段，一些语义方面的属性被附加到文法符号上。

```
STATEMENT
  LVALUE(double)
    IDENTIFIER: Identifier [1](double)
  ASSIGN: Assign
  EXPRESSION(double)
    EXPRESSION(double)
      IDENTIFIER: Identifier [1](double)
    OPERATOR: double operator + (double, double)
    INT_TO_DOUBLE(double)
      EXPRESSION(int)
        EXPRESSION(int)
          IDENTIFIER: Identifier [2](int)
        OPERATOR: int operator * (int, int)
        EXPRESSION(int)
          LITERAL: Literal(Integer) 123(int)
```

考虑表达式计算：由于存在运算符重载等特性，在语法分析阶段通常无法确定某一运算符具体对应哪一函数（特指那些在语法分析树高处的运算符）。在语义分析阶段，由于已经确定了每个子树（子表达式）的类型，也就确定了每一运算符对应的函数。因此，在语义分析之后，这个程序变得可执行。

2.1.4 中间代码生成

我们希望编译器的前端和后端相互独立。这就是说，如果中间代码的格式被约定了，一旦有N个前端（同一源语言或不同源语言的）和M个后端（同一目标语言或不同目标语言的），就能得到NM个不同的编译器，每一个把某种语言的程序翻译成另一种语言。

然而，虽然语义分析后的程序是可解释的，但是语法分析树仍然带有源语言的色彩——如语法结构等。这明显不符合我们对中间代码的要求。因此，我们需要一个中间代码生成器。

常见的中间表示形式是三地址码。在三地址码之前有时还有一种表示形式——语法DAG，它在优化上有重要作用，但是本文不涉及优化所以略去语法DAG的内容。下图是示例的三地址码：

```
1 MULTIPLY(t3, id2, 123) [t3 = id2 * 123]
2 INT_TO_DOUBLE(t2, t3) [t2 = int_to_double(t3)]
3 ADD(t1, id1, t2) [t1 = id1 + t2]
4 ASSIGN(id1, t1) [id1 = t1]
```

2.1.5 代码生成

代码生成器将中间表示转化为目标代码。

我们注意到，三地址码涉及的都是一些基本操作。所以如果不考虑效率的话，很容易从中间表示生成一个目标代码，只要为三地址码的每一指令设计一个目标代码框架就可以了。但这样通常伴随着大量的冗余代码。本文不涉及优化，所以这里不再赘述了，有兴趣的读者可以查阅参考文献[1]。

2.2 程序设计语言的三种实现方式

程序设计语言是来用描述程序的一段代码。这段代码无法被机器直接识别运行。因此，我们需要通过某种方法，将语言所描述的程序转变为机器可识别运行的形式，这被称为程序设计语言的实现方式。常见的三种实现方式是：完全编译、完全解释和混合实现。

2.2.1 完全编译

在这种方式下，编译器一次性地将整个程序编译成可执行的机器码。其特点是编译阶段和运行阶段分离。使用这种模式的语言一般运行效率较好。C++就是使用完全编译方式的语言。

2.2.2 完全解释

这种方式类似于部署了一个该语言的虚拟机：解释器每次找到源代码中的一个语句并翻译它。注意到这样一来，同一个语句可能被翻译数次，这也就说明了为什么使用完全解释模式的语言一般运行效率较差。然而完全解释也有优点：首先，可移植性好；其次，在调试时，很容易定位错误发生的位置。SHELL是一种完全解释方式的语言，有很多脚本语言都是完全解释的。

2.2.3 混合实现

这一方式是编译和解释的折中。在这种方式下，编译器一次性地处理整个程序，但只使用前端——这样就得到了中间表示，然后对中间表示进行解释。由于中间表示已经经过了一定的格式化，它比源代码更加易解释，因此多次解释同一语句的代价也就小多了。这类语言有着略低于完全编译语言的效率，同时又有完全解释语言的高可移植性。Java就是一种混合性语言，Java代码经过编译生成“字节码”，然后在Java虚拟机上解释运行字节码。

2.3 巴克斯-诺尔范式

巴克斯-诺尔范式被用来描述上下文无关语言。在下面的章节中描述程序文法时将大量使用巴克斯-诺尔范式。

其格式为$A ::= < String >$。其中A是非终结符号，$< String >$是一个符号串，可以为空串。

比如以下文法就描述了整数四则运算的语法：

```
EXPR ::= TERM
EXPR ::= TERM + TERM
EXPR ::= TERM - TERM
TERM ::= FACTOR
```

```
TERM ::= FACTOR * FACTOR
TERM ::= FACTOR / FACTOR
FACTOR ::= (EXPR)
FACTOR ::= integer_value
```

其中*integer_value*是整数字面值，是终结符号。在编译器中，整数字面值由词法分析器，而非语法分析器定义。

3 基本问题: 不涉及语义的问题

3.1 代码长度统计问题[1]

统计一段Pascal代码的代码长度。代码长度用“单元”的数量定义，每个保留字、每个标识符、每个字面值、每对花括号、每对括号、每个运算符被作为一个”单元”。注释不参与统计。

```
{THIS IS COMMENT}
‘THIS IS A STRING’
(A * A) shr 1
```

以上文为例：第一行只有一个单元，即左花括号；第二行也只有一个单元，即字符串字面值；第三行有6个单元，包括一个保留字，两个标识符，一对括号，一个运算符，一个整数字面值。

很明显，这个问题只需要选手实现一个词法分析器。

3.1.1 词法分析器

词法分析器的实现方法类似于读入优化的写法。首先读入若干个空白字符直到遇到一个非空白字符。根据这个非空白字符的类型判断这个词素的类型。在这个问题中，我们可以认为只有三种非空白字符类型：数字类型（0 – 9），标识符头类型（_、A-Z、a-z），其他。

```
const int TEMPTY = -1;
const int TDIG = 0;
const int TSHEAD = 1;
const int TOTHER = 2;

int charType(char x)
{
    if (x == ' ') return TEMPTY;
    if (x == '\n') return TEMPTY;
    if (x == '\t') return TEMPTY;
    if (x == '\r') return TEMPTY;
```

[1]UVA 189. Pascal Program Lengths, ACM-ICPC 南加利福尼亚区域赛(1989/1990赛年)

```
    if (x == '_') return TSHEAD;
    if ((x >= '0') && (x <= '9')) return TDIG;
    if ((x >= 'A') && (x <= 'Z')) return TSHEAD;
    if ((x >= 'a') && (x <= 'z')) return TSHEAD;
    return TOTHER;
}
```

为了方便在“向前看”的同时不改变输入流中的内容，我们另外写一个输入流管理器来代替标准输入流。输入流管理器有一个缓冲区，这个管理器要支持：返回输入流头的字符，从输入中取得一个字符，因此缓冲区的大小为1。

```
namespace Stream
{
    char buffer;
    void Init()
    {
        buffer = 0;
    }
    char nxtChar()
    {
        if (buffer == 0)
            buffer = getchar();
        return buffer;
    }
    char getChar()
    {
        char ret = nxtChar();
        buffer = 0;
        return ret;
    }
}
```

接下来词法分析器将只调用新的输入流管理器，而不调用标准输入。

```
namespace Lex
{
  string buffer;
    // 缓冲区
  void Init()
  {
```

```
    buffer = "";
  }
  string nxtLexeme()
  {
    if (buffer.size()) return buffer;
    while (charType(Stream::nxtChar()) == TEMPTY) Stream::getChar();
    // 跳过空白字符
        int TYPE = charType(Stream::nxtChar());
    if (TYPE == TDIG)
      {
        while (charType(Stream::nxtChar()) == TDIG)
          buffer += Stream::getChar();
        return buffer;
      }
    if (TYPE == TSHEAD)
      {
        int t;
        while (((t = charType(Stream::nxtChar())) == TDIG) || (t ==
     TSHEAD))
          buffer += Stream::getChar();
        return buffer;
      }
    // TOTHER
    switch (Stream::nxtChar())
    {
      case '+': case '-': case ',':
      case '*': case '/': case '=':
      case '^': case '@': case ';':
      case '(': case ')': case '.':
      case '~': case '[': case ']':
        return buffer += Stream::getChar();
      break;
      case '>': case ':'
        buffer += Stream::getChar();
        if (Stream::nxtChar() == '=')
          buffer += Stream::getChar();
        return buffer;
      break;
      case '<':
        buffer += Stream::getChar();
        if ((Stream::nxtChar() == '=') || (Stream::nxtChar() == '>'))
          buffer += Stream::getChar();
        return buffer;
      break;
    };
  }
    string getLexeme()
    {
```

```
54          string ret = nxtLexeme();
55          buffer = "";
56          return ret;
57      }
58  }
```

这里我们注意到，这段代码中使用string来储存词素。我们知道，除了注释内容和字符串字面值以外，每个词素的类型都可以通过首字符类型判断出：TSHEAD类型的首字符说明一个标识符或关键字（注意在这道题中这两者等价），TDIG类型的首字符说明一个整数字面值开始。

3.1.2 引入科学计数法

上面的那个做法对于原来的这道题是不够的：在原题中数可能以科学计数法读入，并且因此，可以是实数，自然也可能以小数形式出现。

没有科学计数法的词法分析器已经适用于之后的大部分题目。没有兴趣的读者可以认为本题没有这一要素，略过本节。

在这之前先考察本题中的科学计数法是怎么定义的：”The most general form of a numeric constant is illustrated by the constant 10.56E-15. The 10 is the integral part (1 or more digits) and is always present. The .56 is the decimal part and is optional. The E-15 is the exponent and it is also optional. It begins with an upper or lower case E, which is followed by a sign (+ or -). The sign is optional.”

小数点前至少有一个数字。小数点到E前的部分是可选的，E之后的部分也是可选的。

很容易写出这样的代码：

```
1  if (TYPE == TDIG)
2    {
3      int Statu = 0;
4      do
5        {
6          while (charType(Stream::nxtChar()) == TDIG)
7            buffer += Stream::getChar();
8          if ((Statu <= 1) && (Stream::nxtChar() == '.'))
9            {
10             Statu = 1;
11             buffer += Stream::getChar();
12           }
13         else
```

```
14          if ((Statu <= 2) && (Stream::nxtChar() == 'E'))
15            {
16              Statu = 2;
17              buffer += Stream::getChar();
18              buffer += Stream::getChar();
19            }
20          else
21            Statu = 3;
22      }
23    while (Statu <= 2);
24    return buffer;
25  }
```

3.1.3 引入注释

Pascal中注释的策略是：每次找到第一个未被标记为注释或字符串一部分的左花括号，找到这之后的第一个右花括号，它们之间的部分构成注释。读者可以通过在IDE中输入代码{{}}来确认这一点，应当会发现第四个字符不呈现注释的颜色。

那么代码就很简单了。

```
case '{': {
  while (Stream::nxtChar() != '}')
        buffer += Stream::getChar();
  return buffer += Stream::getChar();
}
```

3.1.4 引入字符串

Pascal中识别字符串的策略是：每次找到第一个未被标记为注释或字符串一部分的引号，找到这以后的第一个后面紧跟的不是引号的引号。读者可以通过试着编译以下代码来验证这一点，应当会发现编译错误。

```
begin
  writeln('' '');
end.
```

这部分的代码也很容易写出。

```
case '\'': {
  buffer += Stream::getChar();
  while (1)
    if (Stream::nxtChar() == '\'')
      {
        buffer += Stream::getChar();
        if (Stream::nxtChar() == '\'')
          buffer += Stream::getChar();
        else
          return buffer;
      }
    else
      buffer += Stream::getChar();
}
```

3.1.5 其他实现细节

如果希望AC此题，选手应当注意到以上处理科学计数法的程序段无法处理定义数组时的区间，如[1..2]。这将要求输入流管理器的缓冲区大小扩张到2，并且对处理数的部分进行相应的修改。

4 进阶问题: 模拟源代码的问题

4.1 一个简易python解释器[2]

编写一个解释器。

支持两种语句：

1. 赋值语句：变量=表达式。变量只有整数型，且无需事先声明。表达式中只有以下要素：四则运算符（加、减；乘、整除），变量名，常量。没有括号。

2. 输出语句：print(变量名或常量[, 变量名或常量]*)

这里的常量皆为可以用16位带符号整数型存下的自然数。所有变量在被调用前至少被赋值过一次。输入数据符合Python3语法。

4.1.1 区分两种语句

输出语句是容易模拟的。用一个STL::map将变量名映射到值即可。

关键在于赋值语句，之后会介绍多种方法来处理赋值语句。

首先主程序的结构可以写成一个while循环，每个循环体处理一个语句。

从词法分析器读入一个语句的首词素。这个词素要么是print，要么是变量名。注意：通过这个词素并不能判断出语句的类型——在Python3中print并非是一个关键字，而是一个函数对象。对这个对象进行赋值是合法的——只不过在此之后就不能使用print函数进行输出了。因此正确的区分两个语句的做法是，在读入首词素后检查（这样就只需要大小为1的缓冲区了）下一词素。如果是等于号，则为赋值语句，是左小括号，则为输出语句。

现在我们已经能区分哪个是赋值语句了。现在有三种方法来处理赋值语句右端的表达式。这些方法分别列于下面。

现在本题还剩余的部分就是支持高精度了。不过那就与本文内容无关，也就不再赘述了。

4.1.2 第一种: 扫描，维护栈的经典表达式计算

具有四则运算的表达式计算是一个经典问题。

[2]水题(py), 某不知名NOI模拟赛

维护一个符号栈和一个值栈，从左到右扫描，遇到加减法压入两个栈，遇到乘除法直接应用到值栈的栈顶元素即可。

在这种做法下，如果要支持括号的话，就要为每个元素设置时间戳。遇到乘除法时并非直接应用，而是同样压入。只不过每轮循环体结束前检查：若符号栈头是乘除法，并且其时间比值栈头来得早的话，把符号栈头元素应用在值栈的最上方两个元素上。

在这种做法下，如果要支持更多优先级的运算，应当这么修改：每次遇到运算符的时候，检查栈顶，应用所有优先级更高的运算。将符号栈做成一个单调栈。

4.1.3 第二种: 一次性读入，按运算符优先级拆解表达式再计算

一次性读入所有词素直到遇到回车（这需要对词法分析器进行一些另外的修改），然后找出其中所有的最低优先级符号——加法和减法。用这些符号将表达式拆分成数个内含运算符优先级皆严格高于加法和减法的“项”（这里借用了“多项式”中的“项”这个术语，并没有什么特别的意思）。然后将每个“项”用次低优先级符号——乘法和除法拆分成多个“系数”即可。求值的时候先计算出所有“项”的值，再计算整个表达式的值即可。

在这种做法下，如果要支持括号的话，只需要在寻找目前最低优先级时忽略掉那些在括号中的符号，并把表达式计算写成一个函数，进行递归调用即可。

在这种做法下，如果要支持更多优先级的运算，简单地增加步骤数即可。

4.1.4 第三种: 扫描，伴随着递归下降的语法分析的表达式计算

这种方法和以上两种相比，在实现时，来得更复杂。虽然如此，之后的题目中并非只有表达式计算，将增加更多的要素。因此，掌握递归下降语法分析将是很重要的。

递归下降分析方法由若干个函数构成，每一个函数对应了描述此语言的上下文无关文法中的一个非终止符号。

当描述这一语言的上下文无关文法具备LL(1)性时，实现将会变得非常简单。

以本题为例。以下是本题中表达式满足的上下文无关文法。其中EXPR为初始符号。

```
EXPR ::= TERM
EXPR ::= EXPR + TERM
EXPR ::= EXPR - TERM
TERM ::= FACTOR
TERM ::= TERM * FACTOR
TERM ::= TERM / FACTOR
FACTOR ::= variable
FACTOR ::= integer
```

与其等价的一个LL(1)文法如下：

```
EXPR ::= TERM REXPR
REXPR ::= ε
REXPR ::= + TERM REXPR
REXPR ::= - TERM REXPR
TERM ::= FACTOR RTERM
RTERM ::= ε
RTERM ::= * FACTOR RTERM
RTERM ::= / FACTOR RTERM
FACTOR ::= variable
FACTOR ::= integer
```

LL(1)性说明，在这一文法的递归下降分析器中，对于每个非终结符号，都只需要从词法分析器中读取一个词素就可以判断它适用的变换规则。借助这个性质就可以写出这样的分析程序：

```
1 void EXPR();
2 void REXPR();
3 void TERM();
4 void RTERM();
5 void FACTOR();
6
```

```
7  void EXPR()
8  {
9    TERM(); REXPR();
10 }
11
12 void REXPR()
13 {
14   if (Lex::nxtLexeme() == "+")
15     {
16       AssertNext("+"); TERM(); REXPR();
17       return;
18     }
19   if (Lex::nxtLexeme() == "-")
20     {
21       AssertNext("-"); TERM(); REXPR();
22       return;
23     }
24   return;
25 }
26
27 void TERM()
28 {
29   FACTOR(); RTERM();
30 }
31
32 void RTERM()
33 {
34   if (Lex::nxtLexeme() == "*")
35     {
36       AssertNext("*"); FACTOR(); RTERM();
37       return;
38     }
39   if (Lex::nxtLexeme() == "/")
40     {
41       AssertNext("/"); FACTOR(); RTERM();
42       return;
43     }
44   return;
```

```
45 }
46
47 void FACTOR()
48 {
49   if (isDigit(Lex::nxtLexeme()[0]))
50     {
51       AssertInteger();
52       return;
53     }
54   if (isSHead(Lex::nxtLexeme()[0]))
55     {
56       AssertVariableName();
57       return;
58     }
59   throw "Syntax Error";
60 }
```

这些过程之间的调用结构就构成了该语句的具体语法树。

```
// 样例词素流
integer + integer * variable - integer

// 调用结构
EXPR(); // nxtLexeme is [integer]
  TERM();
    FACTOR();
      AssertInteger(); // nxtLexeme -> ["+"]
    RTERM();
  REXPR();
    AssertNext("+"); // nxtLexeme -> [integer]
    TERM();
      FACTOR();
        AssertInteger(); // nxtLexeme -> ["*"]
      RTERM();
        AssertNext("*"); // nxtLexeme -> [variable]
        FACTOR();
          AssertVariableName(); // nxtLexeme -> ["-"]
        RTERM();
    REXPR();
```

```
      AssertNext("-"); // nxtLexeme -> [integer]
      TERM();
        FACTOR();
          AssertInteger(); // nxtLexeme -> [EOF]
        RTERM();
      REXPR();

// 具体语法树
EXPR ::= TERM REXPR
  TERM ::= FACTOR RTERM
    FACTOR ::= integer
      [integer]
    RTERM ::= ε
  REXPR ::= + TERM REXPR
    ["+"]
    TERM ::= FACTOR RTERM
      FACTOR ::= integer
        [integer]
      RTERM ::= * FACTOR RTERM
        ["*"]
        FACTOR ::= variable
          [variable]
        RTERM ::= ε
    REXPR ::= - TERM REXPR
      ["-"]
      TERM ::= FACTOR RTERM
        FACTOR ::= integer
          [integer]
        RTERM ::= ε
      REXPR ::= ε
```

而有了语法树以后就可以利用它来进行计算了。以之前的分析程序衍生出的求值程序是这个样子的：

```
1 valtype EXPR();
2 valtype REXPR(valtype lhs);
3 valtype TERM();
4 valtype RTERM(valtype lhs);
```

```
5  valtype FACTOR();
6
7  valtype EXPR()
8  {
9    return REXPR(TERM());
10 }
11
12 valtype REXPR(valtype lhs)
13 {
14   if (Lex::nxtLexeme() == "+")
15     {
16       AssertNext("+");
17       return REXPR(lhs + TERM());
18     }
19   if (Lex::nxtLexeme() == "-")
20     {
21       AssertNext("-");
22       return REXPR(lhs - TERM());
23     }
24   return lhs;
25 }
26
27 valtype TERM()
28 {
29   return RTERM(FACTOR());
30 }
31
32 valtype RTERM(valtype lhs)
33 {
34   if (Lex::nxtLexeme() == "*")
35     {
36       AssertNext("*");
37       return RTERM(lhs * FACTOR());
38     }
39   if (Lex::nxtLexeme() == "/")
40     {
41       AssertNext("/");
42       return RTERM(lhs / FACTOR());
```

```
    }
  return lhs;
}

valtype FACTOR()
{
  if (isDigit(Lex::nxtLexeme()[0]))
    return GetIntegerVal();
  if (isSHead(Lex::nxtLexeme()[0]))
    return GetVariableVal();
  throw "Syntax Error";
}
```

问题到此解决。不过，如果对于向分析程序内部传入参数感到不适的话，可以活用while循环简化这个程序。

```
valtype EXPR();
valtype TERM();
valtype FACTOR();

valtype EXPR()
{
  valtype ret = TERM();
  do
    {
      if (Lex::nxtLexeme() == "+")
        {
          AssertNext("+");
          ret = ret + TERM();
          continue;
        }
      if (Lex::nxtLexeme() == "-")
        {
          AssertNext("-");
          ret = ret - TERM();
          continue;
        }
      break;
```

```
    }
  while (1);
  return ret;
}

valtype TERM()
{
  valtype ret = FACTOR();
  do
    {
      if (Lex::nxtLexeme() == "*")
        {
          AssertNext("*");
          ret = ret * FACTOR();
          continue;
        }
      if (Lex::nxtLexeme() == "/")
        {
          AssertNext("/");
          ret = ret / FACTOR();
          continue;
        }
      break;
    }
  while (1);
  return ret;
}

valtype FACTOR()
{
  if (isDigit(Lex::nxtLexeme()[0]))
    return GetIntegerVal();
  if (isSHead(Lex::nxtLexeme()[0]))
    return GetVariableVal();
  throw "Syntax Error";
}
```

这个程序实际上是基于以下文法的，它的表达方式是BNF的一个引入

了Kleene星号的扩展版本。

```
EXPR ::= TERM ((+ TERM) | (- TERM))*
TERM ::= FACTOR ((+ FACTOR) | (- FACTOR))*
FACTOR ::= integer | variable
```

现在我们得到了两个扫描并伴随着递归下降语法分析的解决本题表达式计算问题的程序。下面我们考虑它的扩展。

增加更多优先级运算是容易的，只需要如法炮制，增加几个分析过程就可以了。

增加对括号的支持也是容易的。这相当于在FACTOR的变换规则中加上一条FACTOR::=(EXPR)。对FACTOR()稍作修改即可。

如果常数中有负数，就需要增加对单目运算“负”的支持。在TERM和FACTOR之间加上一层分析过程就可以了。这层分析过程大概会是这样的：

```
valtype NEWLAYER()
{
  if (Lex::nxtLexeme() == "-")
    {
      AssertNext("-");
      return - FACTOR();
    }
  return FACTOR();
}
```

从而我们可以方便地将这一分析程序加入更多的特性。这将极大地有助于我们解决接下来的问题。

4.2 一个简易的C++混合实现系统[3]

你需要编写一个C++解释器。它需要支持以下特性：

int型变量，以及int数组（含高维数组）变量；int型变量上的若干运算；使用cin和cout进行int的输入和输出；全局变量和局部变量（不含动态分配内存）；使用int putchar(int)函数输出一个字符；以零或一或多个int为参数的，int型自定义函数（不含提前定义），直接递归；if语句，for循环和while前置的while循环。

这道题目的部分分基本上是按照编写解释器的步骤分步给出的。因此摘抄部分分分布如下：

第一个测试点是送分点。

第二个测试点是只含加减乘除和自然数常数的表达式计算。

第三个和第四个测试点是表达式计算。

第五个测试点中只有main函数，且不含流程控制语句。

第六个和第七个测试点中只有main函数。

4.2.1 测试点2

第二个测试点的做法参见前一节“简易python解释器”。

4.2.2 测试点3和4

这两个测试点就是在第二个测试点的分析程序上加入几级新的运算，加入括号，加入单目运算，以及一个有语义动作且返回自身的函数(即putchar函数)。

同样参见前一节。

4.2.3 测试点3和4, 准备转化到测试点5

在之前的程序中，我们抽取了cout和endl之间的内容并对其进行分析。

为了处理一个cout后跟多个表达式，或cin后跟多个地址的场合，我们应当对整个语句进行分析，这就是说，将输入输出运算符也作为运算而加入表达式处理模块。

[3]tsinsen P7003 / uoj 98. 未来程序・改, 集训队互测’15 第四场

这两个运算是具有动作的。在这里我们使用以下策略来实现这两个运算符：当运算符<<以后的那个更低级非终结符号对应的表达式被计算出来以后，就直接输出它。（因为在本题中可以确定此运算符前的表达式的结果是cout。）cin同理。

写成代码大概是这样的。

```
valtype NEWLAYER()
{
  valtype ret = NEXTLAYER();
  do
    {
      if (Lex::nxtLexeme() == "<<")
        {
          AssertNext("<<");
          putvalue(NEXTLAYER());
          continue;
        }
      if (Lex::nxtLexeme() == ">>")
        {
          AssertNext(">>");
          getvalue(NEXTLAYER());
          continue;
        }
      break;
    }
  while (1);
  return ret;
}
```

其中NEWLAYER处理输入输出运算以及比它优先级更低的运算（其实就是一切运算，NEWLAYER就是语句），NEXTLAYER处理低于输入输出运算，也就是通常意义上所说的表达式。putvalue和getvalue是valtype有关的两个函数，关于getvalue的实现将在之后引入变量概念时具体介绍。

4.2.4 测试点5, 新特性一: 变量

测试点5引入了变量。

在本题中，有以下对象存在：int型变量，int&型变量（赋值号左边的那个东西），cin，cout，endl。

我们先考虑如何处理前面那两个东西。我们开一个大数组来模拟内存，然后用一个$pair < bool, int >$（下简写为，PBI）来表示对象。当PBI的bool是真时，代表这个PBI表示的对象在内存中有确定的地址（这个变量在大数组中的位置保存在PBI的int中），是一个int&；否则它只是一个值，具体的值保存在PBI的int中。

接下来要加入cin，cout，endl这三个对象。只需要将某个PBI的bool设为真，然后在int中写一个不可能出现在地址中的整数，比如说，负数，就可以表示像cin这样的特殊对象了。

下面是putvalue和getvalue的一种实现。

```
const constENDL = make_pair(true, -1);

void putvalue(PBI x)
{
  if (x == constENDL)
    cout << endl;
  else
    if (x.first)
      cout << MEMORY[x.second];
    else
      cout << x.second;
}

void getvalue(PBI x)
{
  if (!x.first) throw "Syntax Error";
  if (x.second < 0) throw "Syntax Error";
  cin >> MEMORY[x.second];
}
```

当两个PBI参与除赋值和输入输出以外的运算时，建立一个新的PBI作为其结果。其bool为false，其int为运算结果。示例如下。

```
int to_val(PBI x)
```

```
{
  if ((x.first) && (x.second < 0))
    throw "Runtime Error";
  if (x.first)
    return MEMORY[x.second];
  else
    return x.second;
}

PBI operator + (PBI lhs, PBI rhs)
{
  return make_pair(false, to_val(lhs) + to_val(rhs));
}
```

当两个PBI参与赋值时则根据左手侧的PBI的int来把右侧的值放到内存数组中适当的位置。

4.2.5 测试点5, 新特性二: 数组

对于数组的处理有多种方式。有一种方法是用恰好等于数组中元素个数的连续内存来保存数组，每次用公式计算需要的元素在那个位置。但由于本题中的变量占用内存数量不多，所以还有一种空间利用率在50%以上，100%以下的实现方法，这种方法与已有的表达式计算模块更加相容。

对一个数组建立一棵树。每个i维数组的孩子就是它下面的那数个$i-1$维数组。这种实现就为这棵树上的所有节点都设置一个内存中的位置。在叶子节点保存这个变量的竖直，在非叶子节点保存它第一个子节点的在内存中的位置，并将每个非叶子节点的孩子放置在连续的内存位置中。数组A[3][2]的一种安排方式是这个样子的：

Address	*Content*	*Value*
x	A	$x+1$
$x+1$	$A[0]$	$x+4$
$x+2$	$A[1]$	$x+6$
$x+3$	$A[2]$	$x+8$
$x+4$	$A[0][0]$	$A[0][0]$
$x+5$	$A[0][1]$	$A[0][1]$
$x+6$	$A[1][0]$	$A[1][0]$
$x+7$	$A[1][1]$	$A[1][1]$
$x+8$	$A[2][0]$	$A[2][0]$
$x+9$	$A[2][1]$	$A[2][1]$

在这种安排下，容易将取数组元素”[]”写成一个PBI与int的运算。

4.2.6 测试点5, 新特性三: 内存管理

虽然测试点5中只有顺序结构，但是，并没有保证只存在int main后的那对花括号。

这就是说，需要考虑变量作用域的问题。

当一个函数中调用了某一变量时，通常使用以下原则确定具体是哪个变量：从这个变量调用语句所属的花括号向上，最低一层在该位置前包含声明语句的花括号中的那个声明语句所声明的变量就是要调用的变量。

在只有一个函数时，我们考虑在遇到声明语句时记录这个语句，在块尾撤销本块中所有的声明语句，那么执行调用语句的时候调用的恰好就是叫这个变量名的最后一个声明语句所声明的变量。

使用一个$map < string, stack < int >>$和一个$stack < stack < string >>$来处理。前者用来调用，后者用来撤销块中所有声明。

4.2.7 测试点5, 新特性四: 多个语句

用一个递归调用的分析过程来分析一个块。

每次通过首词素来判断语句类型。

如果首词素是右花括号说明块结束。

否则：

int指示一个声明语句；

左花括号指示一个块语句；

return指示一个返回值语句，当只有一个函数的时候这可以用分析程序中的一个exit(0)解决；

其他情况下交由之前已经增强过的表达式处理模块处理。

4.2.8 测试点6和7

这两个测试点中引入了循环语句。这可能会导致同一源代码段被反复处理。因此，每次对其进行扫描和语法分析，在效率上变得不佳。

为了解决这一问题，需要在第一遍扫描的时候为其显式地建出抽象语法树，然后之前的扫描过程变成扫描语法树上某一节点的孩子。

抽象语法树有很多实现方法。笔者认为它至少要有以下要素：

一个用来保存有关信息的*vector < string >*。一个变量声明语句的*vector < string >*中可能保存了这个变量的变量名，以及各维度的大小；代表函数的节点的*vector < string >*中可能含有了它的参数的名称；代表只含有加法和减法及其以下运算的子表达式的节点中可能保存了其第二个分量开始的各分量前是加号还是减号。

一个用来保存孩子的*vector < Tree* >*。根节点的孩子可能指向几个变量声明语句节点和几个函数节点。

在建出这棵语法树后，扫描执行就变成了语法树上的某一个节点。整个模拟过程就是执行根节点下的名称为main的function节点。

下面还是使用表达式计算的例子来说明大致的思想。

4.2.9 测试点6和7: 一个示例

这个示例是一个只包含加法、减法和乘法两级的表达式。

上下文无关文法：

```
EXPR = TERM ((+ TERM) | (- TERM))*
TERM = FACTOR (* FACTOR)*
FACTOR = integer
```

EXPR节点的配置是：$vector < string >$含有若干个"+"或"-"，指示了第二个TERM开始的符号，$vector < Tree* >$含有若干个TERM节点。

TERM节点的配置是：$vector < string >$为空，$vector < Tree* >$含有若干个FACTOR节点。

FACTOR节点的配置是：$vector < string >$含有一个元素，为integer的具体数值，$vector < Tree* >$为空。

它的语法树节点定义部分是这样的：

```
struct Tree
{
  vector<string> VS;
  vector<Tree*> VTp;
  Tree()
  {
    VS.clear();
    VTp.clear();
  }
};
```

那么它对应的分析程序是这样的：

```
Tree* analyseEXPR();
Tree* analyseTERM();
Tree* analyseFACTOR();

Tree* analyseEXPR()
{
  Tree* ret = new Tree;
  ret->VTp.push_back(analyseTERM());
  while ((Lex::nxtLexeme() == "+") || (Lex::nxtLexeme()
       == "-"))
    {
      ret->VS.push_back(Lex::getLexeme());
      ret->VTp.push_back(analyseTERM());
    }
  return ret;
}
```

```
16
17 Tree* analyseTERM()
18 {
19   Tree* ret = new Tree;
20   ret->VTp.push_back(analyseFACTOR());
21   while (Lex::nxtLexeme() == "*")
22     {
23       Lex::getLexeme();
24       ret->VTp.push_back(analyseFACTOR());
25     }
26   return ret;
27 }
28
29 Tree* analyseFACTOR()
30 {
31   Tree* ret = new Tree;
32   ret->VS.push_back(Lex::getLexeme());
33   return ret;
34 }
```

它的解释程序也相应的使用数个函数。

```
1  int executeEXPR(Tree* cur);
2  int executeTERM(Tree* cur);
3  int executeFACTOR(Tree* cur);
4
5  int executeEXPR(Tree* cur)
6  {
7    int ret = executeTERM(cur->VTp[0]);
8    int SZ = cur->VS.size();
9    for (int i = 0; i < SZ; i++)
10     if (cur->VS[i] == "+")
11       ret = ret + executeTERM(cur->VTp[i + 1]);
12     else
13       ret = ret - executeTERM(cur->VTp[i + 1]);
14   return ret;
15 }
16
17 int executeTERM(Tree* cur)
```

```
18 {
19   int ret = 1;
20   for (vector<Tree*>::iterator it = cur->VTp.begin();
         it != cur->VTp.end(); it++)
21     ret = ret * executeFACTOR(*it);
22   return ret;
23 }
24
25 int executeFACTOR(Tree* cur)
26 {
27   int ret = 0, SZ = cur->VS[0].size();
28   for (int i = 0; i < SZ; i++)
29     ret = ret * 10 + cur->VS[0][i] - '0';
30   return ret;
31 }
```

读者可以参考这个示例写出适用于本题的更复杂的建立语法树和解释语法树过程。

4.2.10 测试点8, 9, 10

这三个测试点引入了函数。这要求分析程序的数个模块都要进行改进。

4.2.11 改进一: 表达式计算模块

本来，表达式计算模块中对putchar函数进行了特判。在引入了自定义函数以后，应当有可以处理所有函数的机制。

具体的来说是这样的：当表达式运算的最底层（即之前示例中的FACTOR层）遇到一个标识符时，判断它是不是一个函数。函数表可以通过预先检索语法树中根节点的儿子得到。如果不是函数，那么就是一个变量，向内存管理模块请求对应变量的值。如果是函数，检查是否是预置的putchar，如果是的话执行语义动作，否则，这里需要调用函数，从预先检索得到的$map < string, Tree* >$中检查这个函数对应的语法树子树，下面就是要执行这个子树。在这个时刻向内存管理模块把函数的参数当做变量进行声明和赋值。

4.2.12 改进二: 内存管理模块

4.2.6中的内存管理机制在这三个点将出现错误。

以以下程序为例。

```
int x;

int func()
{
  x;
}

int main()
{
  int x;
  func();
}
```

根据4.2.6的机制，在开始执行main之前，全局变量x被声明，入栈。开始执行main以后，局部变量x被声明，入栈。调用func以后，需要申请x的地址时返回的是x这个名字对应的栈顶，即main函数的局部变量x。这是不符合常理的，在函数func中的x应当指的是一层大括号以外的全局变量x。

于是要引入新数据结构$stack < multiset < string >>$。这个数据结构用来保存每层函数中的变量声明情况。在询问变量地址，首先检查该层函数中是否声明过此变量标识符，若是，返回$map < string, stack < int >>$的栈顶，否则返回那个全局变量的地址，当然这个地址就在$map < string, stack < int >>$的栈底。

4.2.13 一个实现细节: return

return关键字将中断一个函数的调用。

而在执行语法树的过程中中断一个子程序并不等于中断了一个函数调用。这带来一个如何处理return语句的问题。

在之前的测试点中，由于只有一个函数main，当遇到return的时候可以直接中断整个程序（如使用exit(0)）。

我们将一个return的执行过程划分成几个阶段。

执行前阶段：分析器遇到了return关键字，正在计算后面的表达式。

执行时阶段：分析器进行一系列回退，直到回到这个函数的调用语句。

执行后阶段：进行退出一层函数后的打扫工作。

我们注意到，虽然可以有很多个函数中的return语句同时处于执行前阶段，但每时每刻处于执行时阶段的return语句只有一个！这就是说，可以使用全局变量来管理return语句。实现略。

5 变种问题: 结合了数据结构的问题

在4.1所在的那场模拟赛中，4.1被称为“编程难度比较大”的“超级细节题”，需要“熟练的选手的1个小时到1.5个小时”或是“薄弱的选手的3个小时”来完成。而如同4.2一样的强模拟题，即使是出题人在阅读过有关材料（见参考文献）之后，也花了6个小时来写出其长度超千行的标程，也就难怪4.2被用来测试12名候选队员时，最高分只有40分，也就是“复杂表达式计算”的程度了。

第五章的题目在代码强度上远低于4.2（代码长度上是四分之一强的程度），略低于4.1（注意如果要AC4.1还需要一个高精度实现），这一强度是可以接受的，是适合被放置在一场五个小时的OI/ACM比赛中的。

在5.1中将讲解一个基础向问题。它最初出现在一场ACM比赛中，是该场比赛中最难的问题，在逾400个参赛队伍/选手中只有1个队伍/选手解出此题。5.2中的几个问题则是将5.1与一些低阶的高级数据结构相结合得到的产物。

5.1 Unreachable Statement[4]

找出一段Cb语言源代码中所有不可能被执行到的语句。

Cb语言是C语言的一个子集。

一个Cb程序由若干个function组成。

每个function的格式是function 函数名() {语句*}

语句有三种，“print 变量”，“if (表达式) 语句[else 语句]”和”{语句*}”

print没有实际意义。

表达式只有三种，true，false和“变量名二元算符非负常量”。二元算符包括四种不等号。

所有变量为int类型。没有变量声明。

5.1.1 分析程序框架

分析这个问题的分析程序可以被分成以下模块。

1. 输入流管理器、词法分析器（见3.1）。
2. 语法分析器（见4.2）。
3. 可行性判定器。

[4]ZOJ 3786. Unreachable Statement, 第十一届浙江省大学生程序设计竞赛

可行性判定器是本章新引入的一个构件，5.2中的与数据结构进行的结合就发生在可行性判定器。

下面将分模块讨论针对这道题的实现措施。

5.1.2 输入流管理器的修改

在本题中，需要输出具体在哪些坐标，代码从可执行到变成了不可执行到。对于具体坐标的要求使得有必要在输入流管理器中加入一个当前坐标信息。这个信息在每次调用Stream::getChar()函数时更新。

5.1.3 词法分析器的修改

一个根据本题中可能出现符号而设计的普通的词法分析器就可以达到本题的要求。但是同样是由于上一小节中提到的原因，需要记录一个缓冲区词素坐标位置信息。这个信息在每次调用Lex::nxtLexeme()时，在跳过所有空白字符后更新。

5.1.4 语法分析器的修改

本题中不存在循环语句。从而一个初级的，如同4.1.4的递归下降语法分析器就可以胜任这份工作。

在扫描到一个条件分歧语句，并准备进入其一个分支时，语法分析器就需要收集有关表达式的信息，将这一信息传入可行性判定器，若可行性判定器在这一信息传入前后返回的结果从“可行”变成“不可行”，则语法分析器需要从词法分析器中取得坐标信息并把它输出来。在这一个分支结束之后语法分析器还应当告诉判定器删除这一信息。

由于复杂非终止符号的数目比较少，因此这个语法分析器可以用非递归的形式实现。在5.2.1和5.2.2中要求使用非递归形式的语法分析器，具体实现留给读者。

注意到有if就存在else匹配问题。一个这样的上下文无关文法难以被改造成LL(1)形式。然而根据常理，一个else和它之前最近一个尚未匹配的if匹配。因此在IF分析程序中只需要贪心往前取else即可，并不一定要把整个文法改造成LL(1)文法。

5.1.5 可行性判定器

可行性判定器是一个数据结构。它处理一系列命题，并实时地返回这一系列命题是否矛盾。这些命题对应了源代码中套着当前位置的那些条件分歧语句。当可行性判定器返回矛盾时，就代表语法分析器当前扫描到的位置是不可访问到的。

可行性判定器需要支持以下操作：

1. 增加一个命题。

2. 删除最后一个增加的尚未删除的命题。

3. 返回是否矛盾。

在本题中，这个可行性判定器包含一个$map < string, stack < pair < int, int >>>$和一个$int$。前者保存某个变量当前的取值范围的上界和下界，后者保存发现的冲突数，后者的值等于$false$的数量加上前者中那些上界小于下界的变量的个数。使用$stack$是为了便于撤销最后一个命题。当判定器中的int为0时，说明没有冲突，否则说明发现了冲突。

5.2 变体

从5.1.1中可以看到，可行性判定器是一个独立的模块。通过更换这个模块，可以实现一些其他功能。

5.2.1 运用线段树的变体[5]

在5.1的基础上：

if的表达式的格式变更为“a[下标] >= 值”，这里a是一个长为N的int数组，其每个元素被要求大于等于0，并且有公理：如果a[i..j]皆非0，则a[i..j]的元素和小于等于给定的常数K。

在本题中，可行性判定器的工作就是在判定取值范围冲突以外判断一个数列的最大非零子段和大小。这里只需要加上一棵线段树即可。

[5]HDU 4874. ZCC Loves RPG, 多校'14 第二场

5.2.2 运用并查集的变体[6]

在5.1的基础上：

变量的类型从int替换成bool。if的表达式变更为两个变量是否相等。

在本题中，可行性判定器是一个并查集。由于其撤销操作只针对最后的操作，所以只需要用一个栈记录每次连了哪条边即可。使用按秩合并来保证复杂度。

5.2.3 运用凸包的变体

在5.1的基础上：

变量的类型从int替换成浮点数且只有两个（x和y）。if的表达式变更为“ax+by+c >= 0”。并且有公理$|x|, |y| \leq 100$。

在本题中，每时每刻的(x,y)范围在一个凸多边形中。用两棵伸展树来保存这个凸包。每个命题代表的直线将一个凸包分成两个，讨论一下与两个凸壳的相交的各种情形即可。

5.3 小结

本类问题的解答程序，就是在固定的输入流管理器+词法分析器+语法分析器的基础上，添加不同的可行性判定器模块。

这类问题自带的“撤销最后一个未被撤销的操作”的性质，使得一些难以可持久化的问题在这一性质之下变得容易实现——同时还没有现成的算法。

从而，这类问题不仅在设计试题上，还在深入分析数据结构类问题的方面上有其特别的意义。

还有哪些数据结构问题是难以可持久化，但是容易进行回退操作的？直到现在笔者只想到了5.2.2和5.2.3两个例子。这里权当抛砖引玉，将这个问题留给我们的读者思考了。

[6]ZCC Loves AVG, 校内模拟赛

6 感谢

感谢CCF提供这样一个学习和交流的机会；

感谢教练李建老师的指导；

感谢在本文写作过程中给予我帮助的同学们；

感谢浙江省大学生程序设计竞赛的出题人员，他们的工作让我发现了这个对我而言全新的领域。

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman “Compilers: Principles, Techniques & Tools (2nd ed.)” （《编译原理(第2版)》，机械工业出版社）

[2] Robert W. Sebesta “Concepts of Programming Language (10th ed.)” （《编程语言原理(第10版)》，清华大学出版社）