

构建一个编译器

Table of Contents

1. 简介	1
1.1. 一些话	1
1.2. 摆篮代码	3
2. 对表达式进行语法分析	6
2.1. 让我们开始吧!	6
2.2. 单字符的数字	6
2.3. 二元表达式	7
2.4. 一般表达式	9
2.5. 使用栈	10
2.6. 乘法和除法	11
2.7. 括号	13
2.8. 一元运算符负号的处理	14
2.9. 有关优化的一点东西	16
3. 更多表达式	17
3.1. 简介	17
3.2. 变量	17
3.3. 函数	18
3.4. 更多有关错误处理	20
3.5. 赋值语句	21
3.6. 多字符token	22
3.7. 空白符	23
4. 解释器	32
4.1. 简介	32
4.2. 解释器	34
4.3. 一点简单的哲学	37

1. 简介

1.1. 一些话

这个系列的文章是一个开发语言的语法分析器和编译器的教程。在我们完成编译器之前，我们将会覆盖构建编译器的每一方面的知识：设计一门新的编程语言，然后构建一个可以工作的编译器。

尽管我不是一个科班出身的计算机科学家（我的博士学位是物理学方面的），我对编译器却已经感兴趣很多年了。我购买了编译器方面的所有书籍并尝试着消化所有的内容。我不介意告诉你这个消化的过程非常的缓慢。编译器方面的书都是写给计算机专业的学生的，而对于我们这些业余爱好者则读起来非常的困难。但经过了很多年的消化，我终于开始学到一点东西了。改变发生于我开始自己鼓捣自己的编译器。现在我决定和你们分享一下我所学到的东西。当这个教程的系列结束时，你当然不可能成为一个计算机科学家，你也不可

能学到所有有关编译器的知识。因为我想做的就是忽略掉所有编译器的理论。而你所能学到的就是构建一个可以工作的编译器所必要的方方面面的有关实践的知识。

所以这是一个边做边学的教程。在这一系列教程中，我将会不断的写程序做实验。你最好跟上我的脚步，把我写过的代码自己也写一遍，最好也能添加一些你自己写的代码。我将使用Turbo Pascal 4.0编译器。我将在TP中不断的编写代码。这些代码都是可以执行的代码，可能你会想把这些代码粘贴到你的电脑上然后运行。如果你没有一份Turbo Pascal的拷贝，那么可能会影响你写代码的进程。你最好搞一份拷贝。毕竟，这是一个伟大的产品，能用在很多的地方。

一些有关编译器的文章会包含一些示例代码，或者包含一个完整的编译器代码（例如Small-C编译器），然后你可以拷贝代码然后运行，但无法理解整个代码是如何工作的。我希望做的比那些文章多一些。我希望能教给你如何搞定这些东西，然后你就可以自己搞定这一切，甚至可以鼓捣自己的编译器，或者改进我写的编译器，而不只是重复一遍我的工作。

毫无疑问写一个编译器是一个很有野心的工作，所以不可能一页纸就搞定。我希望在一系列的文章中把这些工作搞定。每一篇文章将会覆盖编译器的某个单独的主题，这些主题独立性很强。如果你某个时间段只对某个主题感兴趣，那你只需要看那个主题的文章就可以了。每篇文章都比较完整，所以在最终完成之前，你只需要等待最新的文章就可以了。所以请耐心。

我的系列教程将不会讲解编译器理论涉及到的很多的方面。正统的编译器课本会遵循以下教学过程：

- 一个介绍性的章节，用来介绍什么是编译器。
- 一个或者两个章节，用来介绍编程语法方面的东西，使用巴科斯-诺尔范式（Backus-Naur Form，BNF）。
- 一个或者两个章节，用来介绍词法分析。重点讲解确定性和非确定性有限自动机。
- 多个章节。用来介绍语法分析理论，从自顶向下递归下降语法分析开始，到LALR语法分析器结束。
- 一个章节，用来介绍中间语言。重点介绍P-code，以及相似的逆波兰表示形式。
- 多个章节。用来介绍处理子程序、传递参数和类型声明的多种方式。
- 一个章节。用来介绍代码生成。通常针对一个拥有简单指令集的虚拟CPU生成代码。大部分读者（事实上，大部分大学的课程）都不会走到这么远。
- 最后的一个或者两个章节。用来介绍优化。读者通常也不会走到这一章。

在我的系列教程中，我将会走一条很不同的路线。我不会给你很多的技术选项，我只会给你一条路。如果你想探索编译器技术各阶段的各种实现技术，好吧，我会鼓励你这么做，但我只会教你我会的那一条路。我也会跳过很多会让人昏昏欲睡的理论知识。别误解我：我并不是鄙视理论，只是当我们去探索一门给定的编程语言的充满各种奇技淫巧的实现时，才会需要理论。但我一直坚信，我们先得把最终要的事情搞定。在教程中，我们将会搞定95%的编译器方面的技术，而这些是不需要那么多理论就能搞定的。

我将会只讨论一种语法分析技术：自顶向下递归下降语法分析器。这个技术是手工打造一个编译器所能使用的唯一的一种技术。如果使用其他方法实现语法分析器，那你就必须使用工具了，例如YACC。而且使用工具的话，我们无法控制最终的编译器会消耗多少内存。

我也借鉴了Ron Cain写的书的部分内容，他是Small C的作者。很多编译器的作者都会使用一种中间语言（例如P-code），也就是将编译器分割成两部分（一个是编译器的前端，用来将代码编译成P-code，一个是编译器的后端，用来将P-code编译成目标机器的指令集代码）。Ron给我们展示了一种方法，那就是我们可以直接将代码编译成目标机器的可执行代码（也就是目标机器的汇编语言程序）。当然编译成的汇编语言代码肯定不是最紧凑的代码，因为编译出紧凑的代码，也就是优化是一个很难的工作。但编译出来的汇编代码是可以工作的，这就够了。当然我也不想让你们觉得我们写的编译器是毫无价值的，所以我会告诉你一些优

化的方法。

最后，我会在教程中使用一些技巧，来帮助我们理解编译器的构造过程。这里面我使用的最主要的技巧就是单字符token（token里没有空格）的使用。因为我发现一旦我能够写出一个识别I-T-L这些token的语法分析器，那么写一个识别IF-THEN-ELSE的语法分析器就很简单了。在第二课中，我会向你展示如何编写一个语法分析器来识别任意长度的token。还有一个技巧，就是我会忽略掉文件I/O，因为我发现如果我能够处理来自键盘的输入以及将结果输出到屏幕上，我就可以处理来自硬盘的文件读写。经验表明，如果一个翻译器可以正常工作，那么将翻译器和文件I/O连接起来将会很简单。最后一个技巧就是，我不会去做错误处理和错误恢复方面的工作。我们编写的程序将会识别错误，但不会崩溃，而是直接停止运行，停止的位置是编译器发现的第一个错误，就像Turbo Pascal做的那样。你也可以发明一些其他的技巧。这些技巧不会出现在编译器的课本里，但他们很好使。

有关编程风格和效率的一点补充。就像你看到的那样，我写代码倾向于写非常小而且容易理解的代码片段。我所写的代码不会超过15到20行。我是KISS（keep it simple, sidney）的忠实信徒。所以当简单的代码能搞定时，我不会写技巧性很强或者很复杂的代码。这样写代码或许效率不够高？或许吧，但你会喜欢这样的风格的。就像Brian Kernighan所说的那样，先把代码跑起来，再去让代码跑的更快。如果后面你再想去优化之前所写的代码，你可以这样去做，因为之前写的代码很容易理解。如果你要这样做，那么最好在代码都能跑起来时，再去这样做。

我写代码还有一个倾向就是直到我需要时，我才会去构建一个模块。想要预测未来碰到的所有的情况，会把人逼疯，而且很容易出错。在现在这个充满了好用的编辑器和快速的编译器的时代，如果需要，我会毫不犹豫的重构一个模块。到那时候，我可能才会写出我真正想要的代码。

最后一点：我们这里所遵循的原则是不和P-code以及虚拟CPU指令集搞在一起。当我们写完第一天的代码，我们的代码就可以产生可执行的汇编代码。不过你可能并不喜欢我这里选择的汇编语言：68000汇编语言。我电脑运行的就是这种汇编语言。但是你会发现，将我们的代码修改成能够编译出80x86汇编语言代码的程序，也是很容易的，所以我不觉得这是一个问题。实际上，我很希望某个熟悉8086汇编语言的朋友能够写一份和我的代码所对应的编译器。因为这正是我们所需要的。

1.2. 摆籃代碼

每一个程序都有一些固定的写法…I/O的处理，错误信息的处理等等。我们要写的程序也不例外。我会尽可能将这些样板代码浓缩到最小，这样我们可以集中精力写最重要的部分，而不是迷失在样板代码中。下面的代码就是我们要写出的一些样板代码。包括I/O程序，错误处理程序，一个骨架程序和主程序。我把这些程序叫做我们的揆篮（cradle）。当我们编写其他程序时，会把它们添加到揆篮里面，然后添加一些对这些程序的调用程序。拷贝一份揆篮程序吧，因为我们在多处使用这些代码。

有很多种方法来组织一个语法分析器的扫描活动。在Unix系统中，人们倾向于使用getc方法和ungetc方法来读取和回退字符。我这里使用的方法是，用一个单独的全局变量来记录向前看到的一个字符。初始化的部分（唯一的一个初始化部分）读取输入流中的第一个字符。我们没有用到Turbo 4.0的任何的特殊的技术。每个接下来的GetChar方法的调用，都将读取输入流中的下一个字符。

```
program Cradle;

{ 声明常量 }

const TAB = ^I;

{ 声明变量 }
```

```
var Look: char;           { 向前看字符 }

{ 从输入流中读取新的字符 }

procedure GetChar;
begin
  Read(Look);
end;

{ 打印错误信息 }

procedure Error(s: string);
begin
  WriteLn;
  WriteLn(^G, 'Error: ', s, '.');
end;

{ 打印错误信息然后将程序挂起 }

procedure Abort(s: string);
begin
  Error(s);
  Halt;
end;

{ 打印预期看到的信息 }

procedure Expected(s: string);
begin
  Abort(s + ' Expected');
end;

{ 匹配一个特定的输入字符 }

procedure Match(x: char);
begin
  if Look = x then GetChar
  else Expected(' ' + x + ' ');
end;

{ 识别一个字母 }

function IsAlpha(c: char): boolean;
begin
  IsAlpha := upcase(c) in ['A'..'Z'];
end;
```

```

{ 识别一个十进制数字 }

function IsDigit(c: char): boolean;
begin
    IsDigit := c in ['0'..'9'];
end;

{ 获取一个标识符 }

function GetName: char;
begin
    if not IsAlpha(Look) then Expected('Name');
    GetName := UpCase(Look);
    GetChar;
end;

{ 获取一个数值 }

function GetNum: char;
begin
    if not IsDigit(Look) then Expected('Integer');
    GetNum := Look;
    GetChar;
end;

{ 输出一个带有制表符TAB的字符串 }

procedure Emit(s: string);
begin
    Write(TAB, s);
end;

{ 输出带有制表符TAB和CRLF字符的字符串 }

procedure EmitLn(s: string);
begin
    Emit(s);
    WriteLn;
end;

{ 初始化 }

procedure Init;
begin
    GetChar;

```

```
end;
```

```
{ 主程序 }
```

```
begin
```

```
Init;
```

```
end.
```

简介结束了。将上面的代码拷贝到TP中，然后编译它们。要保证编译能够通过然后正确的运行起来。接下来我们将要开始第一课，也就是表达式的语法分析。



如果在Ubuntu下想要进行Pascal编程，可以 `sudo apt-get install fpc`。

2. 对表达式进行语法分析

2.1. 让我们开始吧！

如果你已经阅读了简介这一章，你就知道我们要干什么了。你也应该已经将摇篮代码都拷贝到你的Turbo Pascal软件中了，并且还编译过了。现在我们可以开始了。

我们这篇文章将要学习如何来对数学表达式进行语法分析，以及如何将数学表达式翻译成68000汇编代码。我们预期的输出是一系列的汇编语句，而汇编语句的执行结果是正确的计算结果。一个表达式就是等式的右边，如下：

```
x = 2*y + 3/(4*z)
```

在早期阶段，我的步子会迈的非常非常小。这样初学者不会迷失。有一些很好的课程需要我们在很早的时候就学会，这样我们后面会很容易学习其他的知识。对于有经验的读者，需要忍受一下我讲的一些非常基础的知识。我们很快就会进入到核心区域的知识。

2.2. 单字符的数字

为了保持教程一贯的风格（KISS，还记得吗？），让我们先从绝对最简单的情况开始思考。对我来说，就是一个表达式只包含一个单个字符的数字的这种情况。

在开始写代码之前，要保证你将上一章的摇篮代码已经拷贝到你的Turbo Pascal中了。我们在别的代码中将会再次使用它们。接下来将下面的代码添加到程序中：

```
{ 对数学表达式进行语法分析和翻译 }
```

```
procedure Expression;
```

```
begin
```

```
EmitLn('MOVE #' + GetNum + ',D0')
```

```
end;
```

然后将 `Expression`；这一行添加到主程序当中去，现在主程序如下：

```
begin
    Init;
    Expression;
end.
```

现在运行程序。尝试一下将任意单个数字作为输入。你将会得到一行汇编代码的输出。然后再尝试一下输入任意其他的单个字符，你将会发现我们的语法分析器将会打印一个错误信息。

恭喜你！我们现在已经有一个可以工作的翻译器了！

好吧，我承认上面的代码的功能实在是太弱了。但是你别小看它啊。这个小小的编译器所做的事情，其实都是大型编译器所做的事情：它正确的识别合法的程序语句，然后输出正确的可以执行的汇编代码。而且同样重要的是，我们写的这个小小的编译器能够识别不合法的程序语句，然后给出一个有意义的错误信息。你还想要啥自行车？随着我们不断的扩展我们的语法分析器，我们最好能够确保以上两点永远没问题。

上面写的小程序有一些其他的特点值得聊一下。首先，你会看到我们并没有将语法分析和代码生成分开成不同的模块。一旦语法分析器知道我们想要的工作已经完成，就会立即生成目标汇编代码。在一个真实的编译器中，`GetChar`会从磁盘上读取文件，然后输出到另一个磁盘文件。但我们所用的方法很容易进行测试和实验。

同时也要注意，一个表达式一定会产生一个求值结果，并将求值结果存放到某个地方。我选择的地方是68000芯片的D0寄存器。我可能应该选其他的地方来存放求值结果，但D0也很好。

2.3. 二元表达式

现在我们已经上路了，让我们继续往前开车。必须要承认的是，一个表达式只包含一个数字，够呛能满足我们的需求。所以让我们看一下如何来扩展我们的代码。假设我们想处理下面这种形式的表达式：

或者 1+2
或者 4-3
或者，更一般的形式， `<term> +/ - <term>`



其实上面的最后一行就是巴科斯-诺尔范式，或者简称BNF。

我们需要写一个程序来识别上面所写的 `term` 然后将计算结果存放在某个地方，然后还得写一个程序来识别 `+` 和 `-`，然后输出我们想要的汇编代码。但是如果表达式将计算结果保存在 `D0` 寄存器，那我们将 `Term` 的计算结果保存在哪里？答案就是：同样的地方 `D0`。在我们得到 `Term` 的下一个计算结果之前，我们将会把 `Term` 的第一个计算结果存放在某个地方。

好吧，我们想做的事情基本就是写一个 `Term` 程序，它要做的事情就是我们之前写的 `Expression` 程序要做的事情。所以将 `Expression` 程序重命名成 `Term` 就行了。然后编写新版本的 `Expression` 程序如下：

```
{ 对表达式进行语法分析和翻译 }
```

```
procedure Expression;
begin
    Term;
    EmitLn('MOVE D0,D1');
    case Look of
        '+': Add;
        '-': Subtract;
    else Expected('Addop');
    end;
end;
```

紧接着，在 `Expression` 程序上面写如下两个程序：

```
{ 识别和翻译加法 }
```

```
procedure Add;
begin
    Match('+');
    Term;
    EmitLn('ADD D1,D0');
end;
```

```
{ 识别和翻译减法 }
```

```
procedure Subtract;
begin
    Match('-');
    Term;
    EmitLn('SUB D1,D0');
end;
```

当你完成了以上工作，现在各个程序的顺序应该如下：

- `Term` (老版本的 `Expression`)
- `Add`
- `Subtract`
- `Expression`

现在运行程序。尝试一下你能够想到的所有的两个单字符数字所组成的排列组合，用 `+` 和 `-` 进行分割。你每次运行应该能够得到4行汇编代码。现在尝试一下能够出现错误的一些表达式。我们的语法分析器捕获到这些错误了吗？

看一下我们程序产生的汇编代码。有两个地方需要注意。第一，生成的代码并不是我们自己会写的那种汇编代码。下面的代码：

```
MOVE #n, D0  
MOVE D0, D1
```

很低效。如果我们手写汇编代码，我们肯定会直接将数据 `#n` 加载到 `D1` 寄存器中啊。

这里还反映出一种信息：那就是我们的语法分析器产生的汇编代码比我们手写的汇编代码效率要低。习惯它吧。在本系列教程中，一直都是这样的。其实，在某种程度上，所有的编译器都是这样的。一些计算机科学家终其一生都在研究代码优化，他们所做的工作确实改进了生成的代码的质量。一些编译器做的很好，但这样做会付出很大的代价，编译器代码的复杂度会很高。而且这也是一场注定会失败的战争，可能永远不会出现一种情况，那就是一个好的汇编程序员无法打败编译器生成的汇编代码。在这个系列教程结束之前，我会提几句可以对编译器做的一点优化。仅仅是为了告诉你做一些简单的优化也不太难。但是要记住，我们要学习的不是代码的优化。现在，通过阅读这一系列的教程，我们会忽略掉优化方面的东西，重点学习如果生成能运行的汇编代码。

还要说的一点是：我们的代码有问题，是错的！当然产生的汇编代码可以运行，减法程序会从 `D0` 寄存器（存放的是第二个参数）的值减去 `D1` 寄存器（存放的是第一个参数）的值。这种方式是错误的，因为我们产生的结果的正负是有问题的。所以让我们来修复一下 `Subtract` 程序的bug，我们用改变结果的正负性的方式就可以解决这个问题，代码如下：

```
{ 识别和翻译减法 }  
  
procedure Subtract;  
begin  
    Match( '-' );  
    Term;  
    EmitLn( 'SUB D1, D0' );  
    EmitLn( 'NEG D0' );  
end;
```

现在我们的代码更加低效了，但最起码能够输出正确的结果了！不幸的是，程序中表示表达式中的 `term` 的顺序看起来很别扭。这就是我们生活的真相啊。当我们实现除法时，又会碰到同样的问题。

好吧，现在我们已经拥有了一个语法解析器能够识别两个数字的和或者差。之前，我们的程序只能识别一个单个的数字。但是真正的表达式可以拥有两种形式中的一种（单个数字或者加减法表达式）。现在你可以运行程序然后输入一个单个的字符 '`1`'，看看能处理之前的表达式形式吗？

是不是无法工作了？为什么无法工作了？我们完成的语法解析器目前只能识别这样的表达式：那就是有两个 `term` 的加减表达式。我们必须重写 `Expression` 方法，让它能做更多的事情。而这才是一个真正的语法分析器开始的地方。

2.4. 一般表达式

在一个真实世界里，一个表达式可以包含一个或者多个 `term`，用加减运算符进行分割。在BNF中，写做下

面的形式：

```
<expression> ::= <term> [<addop> <term>]*
```

我们可以在 **Expression** 方法中添加一个简单的循环，来适配上面的定义：

```
{ 对表达式进行语法分析和翻译 }
```

```
procedure Expression;
begin
    Term;
    while Look in ['+', '-'] do begin
        EmitLn('MOVE D0,D1');
        case Look of
            '+': Add;
            '-': Subtract;
            else Expected('Addop');
        end;
    end;
end;
```

现在我们又前进了一步。这个版本的程序可以处理任意数量的 **term**，而只耗费了我们两行额外的代码。当我们继续前进时，我们会发现这就是自顶向下语法分析器的特点…只需要添加几行代码就可以适配编程语言的扩展。注意，**Expression** 方法和BNF定义是多么的匹配啊！这同样是自顶向下语法分析器的一个特点。当你熟练掌握了这种方法，你会发现将BNF定义转换成语法分析器的代码是非常容易的！

好吧，现在可以尝试一下我们最新版本的语法分析器了。验证一下会发现我们的代码可以处理各种合法的表达式，还会对非法的表达式输出一个有意义的错误信息。很整洁吧？你可能会发现在我们测试的时候，任何错误信息都会嵌在我们产生的汇编代码里。但是记住，这是因为我们使用 **CRT** 作为我们的输出文件。在一个可用的产品里，这两种输出是分开的…一个输出到屏幕，一个输出到文件中。

2.5. 使用栈

现在我将会打破我不引入任何复杂性的原则。因为这里引入复杂性是绝对必要的。我们需要指出代码中的一个问题。现在代码的逻辑是，语法分析器将会使用 **D0** 寄存器来作为 **主要** 寄存器，**D1** 寄存器作为存储部分和的地方。现在程序工作起来还比较好，因为我们只需要处理的运算符是加号和减号。任何新的 **term** 一旦被发现都会被累加。但在一般情况下，就不好使了。例如下面的表达式：

```
1+(2-(3+(4-5)))
```

如果我们将 **1** 放入 **D1** 寄存器中，那我们把 **2** 放在哪里？因为一个一般的表达式可能有任意复杂度。所以我们将会很快用完所有寄存器！

幸运的是，有一个简单的解决方法。就像所有现代的微处理器一样，68000处理器也有一个栈。栈是一个用来存储一堆东西的完美的地方。所以无需将 **term** 从 **D0** 移动到 **D1** 这么麻烦，我们直接将 **term** 压入栈就可

以了。对于不熟悉68000处理器的读者，我们说一下如何压栈，如下汇编就可以：

```
压栈操作,      -(SP)
弹栈操作,      (SP)+ .
```

所以让我们更改一下 `Expression` 方法中的 `EmitLn` 代码：

```
EmitLn('MOVE D0,-(SP)');
```

然后更改两个数的加减操作的代码 `Add` 和 `Subtract`：

```
EmitLn('ADD (SP)+,D0')
```

以及

```
EmitLn('SUB (SP)+,D0'),
```

现在重新编译尝试一下语法分析器，会发现并没有搞崩代码。

我们的代码比之前的更加低效了，但这是一个必要的步骤，你会看到的！

2.6. 乘法和除法

现在让我们来做一些真正的复杂的工作。很明显你知道，除了加减运算符还有其他的数学运算符，表达式需要有乘除法。你已经知道了有一个隐含的运算符叫做 **优先级**，或者叫做等级。在表达式中优先级很重要，就像下面的表达式：

```
2 + 3 * 4,
```

我们都应该先做乘法运算，然后再做加法运算。（知道我们为什么需要栈了吗？）

在编译器技术的早期，人们会使用一些超级复杂的技术来保证运算符的优先级被遵守。后来发现，这些超级复杂的技术是完全没有必要的。运算符优先级的规则可以很好的被我们的自顶向下语法分析技术所适配。而直到现在，我们考虑的 `term` 还只是一个单字符的数字。

更加一般的方式是将 `term` 定义为多个 `FACTOR` 的 **乘积**，例如：

```
<term> ::= <factor> [ <mulop> <factor> ]*
```

什么是 `factor`？现在，它就是一个单字符数字的 `term`。

注意到对称性了吗？一个 `term` 和一个表达式的形式是一样的。实际上，我们可以对代码做一些重命名和拷贝的工作。但为了避免混淆，下面的代码是语法分析器的所有代码。（注意我们处理除法运算符的方式）

```
{ 对数学因子 (Factor) 进行语法分析和翻译 }
```

```
procedure Factor;  
begin  
    EmitLn('MOVE #' + GetNum + ', D0')  
end;
```

```
{ 识别和翻译乘法 }
```

```
procedure Multiply;  
begin  
    Match('*');  
    Factor;  
    EmitLn('MULS (SP)+, D0');  
end;
```

```
{ 识别和翻译除法 }
```

```
procedure Divide;  
begin  
    Match('/');  
    Factor;  
    EmitLn('MOVE (SP)+, D1');  
    EmitLn('DIVS D1, D0');  
end;
```

```
{ 对数学Term进行语法分析和翻译 }
```

```
procedure Term;  
begin  
    Factor;  
    while Look in ['*', '/'] do begin  
        EmitLn('MOVE D0, -(SP)');  
        case Look of  
            '*': Multiply;  
            '/': Divide;  
            else Expected('Mulop');  
        end;  
    end;  
end;
```

```
{ 识别和翻译加法 }
```

```
procedure Add;  
begin
```

```

Match('+');
Term;
EmitLn('ADD (SP)+, D0');
end;

{ 识别和翻译减法 }

procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB (SP)+, D0');
EmitLn('NEG D0');
end;

{ 对表达式进行语法分析和翻译 }

procedure Expression;
begin
Term;
while Look in [ '+', '-' ] do begin
  EmitLn('MOVE D0, -(SP)');
  case Look of
    '+' : Add;
    '-' : Subtract;
  else Expected('Addop');
  end;
end;
end;

```

来抽一根！一个非常整洁的语法分析器或者说翻译器已经完成了，只用了55行Pascal代码！输出已经开始看起来有那么一点儿用了。当然你得忽略掉生成的汇编代码很低效。记住，我们从来不打算生成紧凑高效的代码！

2.7. 括号

我们可以将这部分的语法解析器改装成可以处理带括号的表达式的解析器。你知道的，括号主要用来强制规定运算符的优先级。比如下面的表达式：

2*(3+4)

括号强制使加法运算发生在乘法运算之前。更为重要的是，括号让我们可以定义任意复杂度的表达式，例如下面：

```
(1+2)/((3+4)+(5-6))
```

将括号处理机制引入我们的语法分析器的关键在于：要意识到无论被括号括住的表达式多么的复杂，对于这个世界来说，它看起来就像是一个简单的 **factor**。也就是说，**factor** 的一种形式如下：

```
<factor> ::= (<expression>)
```

递归来了！一个表达式可以包含一个 **factor**，而这个 **factor** 可以包含其他的表达式，而这个表达式又可能包含了一个 **factor**，可以无限搞下去。

无论复杂与否，我们都得处理这种情况。当然只需要在 **Factor** 方法中添加几行代码就可以了：

```
{ 对数学因子 (Factor) 进行语法分析和翻译 }

procedure Expression; Forward;

procedure Factor;
begin
  if Look = '(' then begin
    Match('(');
    Expression;
    Match(')');
    end
  else
    EmitLn('MOVE #' + GetNum + ',D0');
end;
```

再次注意一下，我们扩展语法分析器是多么的容易啊。我们的Pascal代码和BNF语法也特别的适配。

像之前那样，编译一下最新写的程序，然后保证它能够正确的解析合法的输入，以及能够对非法输入正确的报错。

2.8. 一元运算符负号的处理

现在，我们的语法分析器已经能够处理任意的表达式了，是吗？好吧，试一下下面的输入：

```
-1
```

又废了！不能工作了，是吧？**Expression** 方法期望的输入是以整数开始的输入，而我们的输入是以负号开始的。所以你会发现 **+3** 同样不会工作，下面的表达式也不会工作：

```
-(3-2)
```

其实有很多方法可以搞定这个问题。最简单的方法（当然不一定是最好的方法）是将一个 `0` 添加到这种类型的表达式的最前面。所以 `-3` 变成了 `0-3`。我们可以轻松的将这个补丁打到现在的 `Expression` 方法的代码里面：

```
{ 对表达式进行语法分析和翻译 }

procedure Expression;
begin
  if IsAddop(Look) then
    EmitLn('CLR D0')
  else
    Term;
  while IsAddop(Look) do begin
    EmitLn('MOVE D0,-(SP)');
    case Look of
      '+': Add;
      '-': Subtract;
      else Expected('Addop');
    end;
  end;
end;
```

我和你说过修改代码很简单吧！只需要我们添加3行新的Pascal代码就可以了。注意一下对新的方法 `IsAddop` 的调用。因为对加减法运算符的检测出现过两次，所以我决定将它抽出来成为一个单独的函数。`IsAddop` 方法的形式很明显来自于 `IsAlpha`。下面是代码：

```
{ 识别加减法符号 }

function IsAddop(c: char): boolean;
begin
  IsAddop := c in ['+', '-'];
end;
```

好的，把以上修改完成然后重新编译代码。你可以将 `IsAddop` 方法添加到你的摇篮代码的最底下。因为后面我们还会需要它。现在再尝试输入一下 `-1`，可以工作了！当然代码的效率还是很差的，哈哈。我们用了6行代码才将一个常量加载成功。但最起码它是正确的。记住，我们并没有想要取代Turbo Pascal。

现在我们已经完成了表达式的语法分析器的主要结构。这个版本的程序应该可以正确的解析和编译任意你想输入的表达式了。当然我们的程序还是局限在只能处理单个字符的数字这种 `term`。但我希望现在你能够为语法分析器添加微小的改动，就可以适配对表达式语法规的扩展了。当你听到一个变量或者甚至一个函数调用也只不过是一个 `factor` 时，请不要太惊讶。

在下一篇文章中，我将会向你展示扩展我们的语法分析器来适配以上的扩展是相当简单的。我还会想你展示如何去处理多字符的数值以及变量名。所以看到了吧，我们离一个真正有用的语法分析器已经不远了。

2.9. 有关优化的一点东西

之前的教程，我向你保证过我会给你一些提示，也就是如何去改进生成的汇编代码的质量的方法。像我所说的那样，生成高质量的汇编代码并不是本系列教程的主要目标。但你起码需要知道我们不想在执行汇编代码的时候因为低质量代码的原因浪费时间。实际上，我们可以修改语法分析器来产生更高质量的代码，且并不需要抛弃我们之前写的所有代码。通常情况下，一些优化并不是那么的难做。也就是只需要在语法分析器中添加一些额外的代码就可以了。

有两种主要的方法可以使用：

- 在汇编代码产生之后再去优化生成的汇编代码：这个通常叫做 **窥孔优化**。通常来讲，我们会知道生成的汇编指令的组合顺序，我们也知道哪些汇编代码很糟糕（例如针对 `-1` 产生的汇编代码）。所以我们需要做的就是扫描生成的汇编代码，然后看一下这些组合序列，然后将它们替换成更好的代码就可以了。这有点像宏展开这种技术。只是和宏展开的方向是反的，只需要进行模式匹配就好了。唯一的复杂性在于有大量的汇编代码组合需要去搜索。这种技术叫做窥孔优化的原因就是因为我们一次只能搜索一小组汇编指令的组合。窥孔优化对于代码质量会有惊人的提升。而且窥孔优化无需更改大量的代码。所以这种代价值得付出。生成的汇编代码的运行速度，代码的行数，以及编译器实现的复杂度都值得我们做这种优化。将所有的汇编指令组合都找出来需要很多的IF测试，因为每一个优化都可能是错误的来源。而且，这种测试比较费时间。在经典的窥孔优化器的实现中，窥孔优化会作为编译器的第二个阶段。编译器生成的汇编代码会存放在磁盘上。然后窥孔优化器读取汇编代码文件，然后做优化，优化后的汇编代码继续存放在磁盘上。实际上，你可以将窥孔优化器看成是一个不同于编译器的独立的程序。因为优化器只会从一个小的“窗口”中去窥探生成的汇编代码。一个更好的实现方式是，缓存一些要输出的汇编代码，然后在每一次 `EmitLn` 之后去扫描缓存。
- 尝试在第一次生成汇编代码的时候就生成更好的代码：这种方法要求我们在 `Emit` 汇编代码之前就找到一些特定的情况来进行优化。举个小例子，我们应该可以识别出表达式中常量0和别的数进行相加，所以我们只需要 `Emit` 一个 `CLR`，或者干脆什么都不做。又比如，如果我们在 `Factor` 中（注意，不是在 `Expression` 中）识别出一个一元运算符负号，我们可以将 `-1` 这样的常量直接作为普通的常量，而不是通过正数来生成这样的常量。这些事情都不难。他们只需要在代码中额外添加一些代码就可以了。所以我不想把这些优化代码添加到我的代码中。我的观点是，一旦我们将写的编译器跑起来，能够产生能用的汇编代码，我们再回头去折腾一些优化方面的东西，会比较好。这也是为什么世界上会存在发布2.0版本这种事情的原因。

还有一种类型的优化值得说一下，这种方法似乎会产生非常紧凑的代码，也不会引起很大的争论。这算是我的发明吧，因为我没在其他出版物中看到过。当然，我觉得这应该不是我的原创。

我的这种方法避免了大量使用栈，而是会更好的去使用CPU的寄存器。我们之前只做了加减法，所以我们使用的寄存器是 `D0` 和 `D1`，而不是栈，还记得吗？它可以工作，因为只有两个数需要运算，所以这个隐形的栈从来也没有操作过超过两个数。

而68000处理器有八个数据寄存器。为什么不将它们用做一个私有管理的栈？关键点在于任何时候，语法分析器都知道在栈上的元素数量是多少。所以我们需要妥善的管理这些元素。我们可以定义一个私有的“栈指针”，这个“栈指针”会跟踪我们现在在栈的哪一层，然后访问对应的寄存器。例如 `Factor` 程序，并不会将数据加载到 `D0` 寄存器中，而是会加载到当前的“栈顶”寄存器中。

我们要做的事情实际上是将CPU的内存上的栈替换成自己管理的栈，而这个自己管理的栈是由寄存器模拟出来的。对于大部分表达式而言，栈的层次数量并不会超过8，所以我们可以生成质量较高的汇编代码。当然，我们需要处理栈的深度超过8的情形，但这也不是什么大问题。我们只需要将我们自己用寄存器模拟出来的栈存不下的数据溢出到CPU的栈中去，就可以了。对于栈深度超过8的情况，代码不会比我们现在生成的代码更加糟糕，对于栈深度小于8的情况，产生的代码更好。

上面的这个优化，我已经自己实现过了，只是为了确保这种优化能工作，这样不会对你产生讲解错误。它确实可以工作。在实践中，你不能真把栈的8层都用完。你至少需要一个寄存器用来翻转除法的两个操作数的顺序（真希望68000有一个XTHL，就像8080那样）。对于包含函数调用的表达式，我们还需要一个寄存器来留给它们使用。当然，对于大部分的表达式而言，这种优化将会缩小产生的汇编代码的规模。

所以你可以看到，优化出更好的汇编代码并没有那么困难，但优化确实会增加我们的翻译器的复杂度。我们现在的水平还处理不了这种复杂度。因为这个原因，我强烈建议我们继续忽略掉生成的代码的效率的问题。这样可以保证我们不会为了优化代码而把之前写的代码都扔掉。

下一篇文章，我们将会处理变量这种 **factor** 以及函数调用。我也会向你展示处理多字符 **token** 和输入中的空格是多么的简单。

3. 更多表达式

3.1. 简介

在上一部分，我们分析了用于一般数学表达式的语法分析和翻译技术。我们以一个可以处理满足以下两个约束的任意复杂表达式的小型语法分析器来结束上一章节，不过有两个限制：

1. 只有数值Factor，没有变量
2. 数值Factor限制为单个数字

在这一章节，我们将除去以上约束。我们将扩展我们已做的一切，包括赋值语句和函数调用。记住，虽然第二个约束是我们自己定的…一个让我们更方便，更容易设计，更能集中基本原理的约束。就如你接下去所见的，这个约束是很容易删除的，所以不要太过担心它。我们使用这个技术是为了我们服务，请你相信当我们做好准备时就能把约束去掉。

3.2. 变量

在实际中，我们经常看到许多含有变量的表达式，例如：

```
b * b + 4 * a * c
```

难以想像不能处理含有变量表达式的语法分析器会有多好。幸运地是，这很容易实现的。

请回想我们当前的语法分析器，它允许有两种factor：整数常量和具有圆括号的表达式。用BNF语法表述如下：

```
<factor> ::= <number> | (<expression>)
```

这里，**|** 代表 **or** (或)，意味着对于factor两种形式的任一种形式都是合法的。应该也记得，对于识别这两种不同形式我们并没有困难。向前看字符判断 **(** 为一种情形，而一个数字则属于另一种情形。

大概你不会再吃惊，一个变量也是另一种形式的factor。所以我们扩展上面的BNF语法如下：

```
<factor> ::= <number> | (<expression>) | <variable>
```

同样，这样不会产生二义性：如果向前看字符是一个字母，我们就可知接下来的是一个变量；如果是一个数字，我们得到的是一个数字。当我们翻译一个数时，我们就生成一条加载这个数的代码，就如把一个立即数送入 D0 寄存器。现在我们也是一样，只是加载的是一个变量。

一个在代码生成中兼有的复杂性起源于这样一个事实：大多数68000操作系统，包括我所用的SK*DOS都要求把代码写成"position-independent"(位置独立)形式，这意味着所有一切都是PC相关的。

加载一个变量的汇编语言形式如下：

```
MOVE X(PC),D0
```

这里 X 当然是一個变量名。为了增加语法分析器分析变量表达式的能力，让我们把当前版本的Factor函数改为：

```
{ 对数学Factor进行语法分析和翻译 }

procedure Expression; Forward;

procedure Factor;
begin
  if Look = '(' then begin
    Match('(');
    Expression;
    Match(')');
    end
  else if IsAlpha(Look) then
    EmitLn('MOVE ' + GetName + '(PC),D0')
  else
    EmitLn('MOVE #' + GetNum + ',D0');
end;
```

我在前面也讲过扩展语法分析器是多么容易的一件事，因为方法具有固定结构的。你可以看到在这里同样适用。这次它花费总共只有2行额外代码。也应注意，if-else-else结构是如何精确地表述BNF的语法方程的。

好，编译和测试这个新版本的编译器。应该不会有太大的错误，对吧？

3.3. 函数

这里还有一种许多编程语言支持的常见factor类型：函数调用。对于我们来说要处理好函数问题现在还为时过早，因为我们还不能处理参数传递问题。甚至，一个“真实”的语言包含着支持超过一种类型的机制，其中一种类型就是函数类型。我们也还不能处理这个问题。但出于以下两个理由，我仍想现在就实现函数：首先，它可以让我们汇总语法分析程序，它在某些方面与最终的语法分析程序形式很相近，第二，它也引出了

一个新的十分有价值去讨论的问题。

直到现在，我们已经有能力写一个称为“predictive parser”(预测语法分析器)的程序。这就是说，无论在任何一点上，我们都能根据向前看字符来正确的知道接下来要做什么。但是当我们加入函数后，它就不适用了。因为每种语言都有其命名规则来构造一个合法的标识符。现在，我们简单把标识符规定了一个字母'a'...'z'。问题就在于一个变量名和一个函数名有着相同的命名规则。那么我们怎样区分是标识符还是函数呢？一种方法是在他们使用之前都要先声明。Pascal语言采用的就是这种方法，另一种方法是我们可以要求一个函数后跟一个(也许是空)的参数列表。而这种规则被C语言采用。

因为我们设计中至今没有一个声明类型的机制，所以我们采用C语言的规则。由于我们也没有处理参数的机制，我们只能处理空参数列表的函数，因此函数调用将有已下形式：

```
x()
```

因为我们不处理参数，所有什么也不用做，除了调用函数，我们所要做的是用一个BSR(子程序调用)命令来取代一个MOVE。

既然在Factor函数的测试中，当向前看字符是一个字母时存在着两个可能的分支，所有我们把它分开成两个独立的过程。修改Factor函数如下：

```
{ 对数学Factor进行语法分析然后翻译 }

procedure Expression; Forward;

procedure Factor;
begin
  if Look = '(' then begin
    Match('(');
    Expression;
    Match(')');
  end
  else if IsAlpha(Look) then
    Ident
  else
    EmitLn('MOVE #' + GetNum + ',D0');
end;
```

并在Factor过程前插入一个新的过程：Ident

```

{ 语法分析和翻译一个标识符 }

procedure Ident;
var Name: char;
begin
  Name := GetName;
  if Look = '(' then begin
    Match('(');
    Match(')');
    EmitLn('BSR ' + Name);
  end
  else
    EmitLn('MOVE ' + Name + '(PC),D0')
end;

```

好，编译然后测试这个版本。它能分析所有合法的表达式吗？它能正确地标志一个错误的形式吗？

我们应注意最重要的一点是即使我们不再有一个预测语法分析器，对于我们采用的递归下降方法也不会增添任何复杂性。这样，当Factor函数发现一个标识符(字母)，它也不知道它是一个变量名还是一个函数名，这并不是它所真正关心的。Factor函数只是简单地把这个问题传给Ident函数，并让它去判断。Ident函数则依次读入标识符，并多读一个字符去决定它现在处理的标识符是哪种类型。

紧记这个方法。这是一个非常有用的概念，而且无论什么时候当你遇到二义性情形要求先行扫描时，它都应该被采用。即使你不得不先行扫描几个token，这个原理依然可以适用。

3.4. 更多有关错误处理

当我们在谈论基本原理时，这里还有另一个重要的问题应指出：错误处理。注意到虽然我们做的语法分析器可以正确地拒绝(译：almost，几乎，下面会有解释为什么用almost)每一个我们送给它的畸形表达式，并有一个有意义的出错信息，我们本不用做太多工作让其发生。事实上，整个语法分析程序本质上(由Ident到Expression)只有两个有关错误程序调用。甚至这些都是不必要的…如果你再看看Term和Expression代码，你会发现这些相关的语句都是不可达的。我把它们放入只是早期出于保险考虑，但现在它们不再需要。为什么你现在不删除它们呢？

那么我们如何更自由地获得好的错误处理呢？这很简单，我已经小心地避免直接用函数GetChar读一个字符。取代直接使用GetChar，在错误处理上我依靠GetName，GetNum，和Match去为我完成错误检测。仔细的读者也应该注意到一些Match调用(例如，在Add和Subtract中)其实是不需要的。因为我们已经知道我们得到的字符会是什么字符…但是让它们留在那里会让结构更为对称，而且一般用Match代替GetChar是一个好的设计规则。

我在上面用了一个"almost"。有一种情形是我们错误处理想解决的。迄今为止，我们还没有让我们编译器知道一行结束的特征是什么，也没有告诉当嵌入空格时编译器该如何做。所以一个空白符(或其它不属于可识别字符集的其它字符)都会使我们的编译器忽略还没识别的字符而终止，在这一点上它也许可以被证明是一个合理的行为。但是在一个真正的编译器中，通常有另一个语句跟在一个可以工作的语句后，以至任何一个我们认为是我们表达式一部分的字符将被使用或是被拒绝为下个表达式。

但它仍然是非常简单的修改，即使它只是一个临时的。我们不得不断言表达式应该以行结束符而结束，例如

, 一个回车符。为了了解我正在讨论的, 尝试输入一行:

```
1+2 <space> 3+4
```

看一下语法分析器是如何把空格看成一个终结符的? 现在, 为了让编译器可以适当地标记, 在主函数Main中, 仅在Expression调用后加入一行:

```
if Look <> CR then Expected('Newline');
```

它可以捕捉留在输入流中的一切。不要忘记增加一个常数语句定义CR:

```
CR = ^M;
```

和以往一样, 重编译程序并验证它可以做它所能支持的。

3.5. 赋值语句

好, 我们已经有一个可以工作得非常好的编译器了。我想指出的是, 不包括摇篮代码我们只用了88行可执行代码。但编译的对象文件异常大, 占4752字节。但这并不坏, 想想我们并不难保存这些源代码和对象文件。我们必须坚持KISS原则。

当然, 分析一个表达式之后如果不进行处理它, 这并不是太好。表达式通常(但不是总是)出现在赋值语句中, 如下形式

```
<Ident> = <Expression>
```

其实, 我们离可以有能力分析一个赋值语句只有一瞬之差, 所以让我们把这最后一步完成。仅仅在过程Expression之后加入如下新的过程:

```
{ 语法分析和翻译一个赋值语句 }

procedure Assignment;
var Name: char;
begin
  Name := GetName;
  Match('=');
  Expression;
  EmitLn('LEA ' + Name + '(PC),A0');
  EmitLn('MOVE D0,(A0)')
end;
```

再一次留意到, 代码正好与BNF语法一致。进一步可留意到错误检测并不难, 全交由GetName和Match完成。

出于要求构造PC相关的代码，两行汇编译代码不得不在68000中特殊处理。

现在只要在主函数main中把Expression调用改为Assignment调用。如此而已。

太爽了！实际上我们正在编译赋值语句。如果一个编程语言中只有这一种类型的语句，那么我们就可以把它放入一个循环中而且我们也就有一个完全的编译器了。

当然，一个编程语言中不可能只有一种类型的语句。还有一些如控制语句(条件语句和循环语句)，函数，声明等等。但令人振奋的是，我们已经处理的算术表达式是一个语言中最有挑战性的。相对我们已经做的，控制语句将是十分容易的。我将会把它们补充在第15章节。而其它语句也将同步完成，只要我们记住KISS原则。

3.6. 多字符token

在这一系列的教程中，我已经很小心限制我们所做的一切都是单字符token，并一直让你确信把它扩展成多字符token是不太困难的。我不清楚你是否相信我…如果你过去曾有一点怀疑，我真的不想责备你…在接下来的章节里我会继续用这方法，因为它帮助我们避开了复杂性。但我乐意补充这些向你保证过的代码，这样你就知道扩展一个语法分析器是多么的容易了。在这当中，我们也将为代码中嵌入的空白符作准备。在你接下来改动代码之前，虽然只有一小部分改动，请用另一个文件名来保存当前版本的语法分析器。我们会在后面的部分多次使用它，且我们也将继续在单字符token版本的程序中做开发。

许多编译器把处理输入流分成一个独立的模块称为词法分析器。其主要思想是词法分析器处理一个接一个的字符输入，并返回一个在流中的分离单元(token)。当我们想这样处理时，可以实现它，但我们现在并不需要。我们只需要对GetName和GetNum进行很小的局部修改就可以使其处理多字符记号。

一个标识符通常定义为开头字符是一个字母，而余下为字母数字式的串(字母或数字)。为了完成它，我们需要另一个识别函数：

```
{ 判断一个字符是数字还是字母 }

function IsAlNum(c: char): boolean;
begin
  IsAlNum := IsAlpha(c) or IsDigit(c);
end;
```

把上面的函数加入到你的语法分析器中。我把它放在IsDigit之后。当你实现时，最好也把它作为摇篮代码中永久的一员(译：就是作为模版的一部分)。

现在我们需要修改函数GetName的返回值一字符代替为一字符串：

```

{ 获取标识符 }

function GetName: string;
var Token: string;
begin
    Token := '';
    if not IsAlpha(Look) then Expected('Name');
    while IsAlNum(Look) do begin
        Token := Token + UpCase(Look);
        GetChar;
    end;
    GetName := Token;
end;

```

简单地，把GetNum修改为：

```

{ 获取数值 }

function GetNum: string;
var Value: string;
begin
    Value := '';
    if not IsDigit(Look) then Expected('Integer');
    while IsDigit(Look) do begin
        Value := Value + Look;
        GetChar;
    end;
    GetNum := Value;
end;

```

令人惊讶的是这就是语法分析程序实质上需要改动的全部地方。在函数Ident和Assignment中的局部变量Name，原来声明为char类型，现在必须声明为string[8](显然，我们可以选择让字符串长度更长，但许多汇编程序在某种程度上都限制了长度。完成这些改动，并重编译和测试。现在你相信这是一个简单的改动了吧？

3.7. 空白符

在我们暂时抛开这个语法分析器之前，让我们看看空白符问题。就现在的情况来看，当我们输入一个空白字符时，语法分析器将崩溃。这很不友好。所以让我们进一步开发以消除以上的限制。

使处理空白符容易的关键就在于提出一个简单的规则来规定语法分析器应该如何对待输入流，并能使得这个规则在任何地方都可以执行。直到现在，因为空白符是不允许的，我们就可以假定在每个语法分析行为之后，向前看字符Look都包含着下一个有意义的字符，所以我们可以立即对Look进行测试。我们的设计是基于这个原则的。

对于我来说它仍为一个好的原则，所以它也是我们以后将延用的规则。这意味着所有先行预测输入流的例程必须跳过所有的空白符，并把下一个非空白符保存在Look中。幸运的是，我们已经小心地采用GetName，GetNum，和Match来处理大部分的输入。这里仅三个例程序(加上Init)需要我们修改。

不用惊讶，我们仍以一个新识别例程开始修改：

```
{ 判断是否为空白字符 }

function IsWhite(c: char): boolean;
begin
  IsWhite := c in [ ' ', TAB];
end;
```

我们也需要一个函数去吃掉空白字符，直到找到一个非空白字符：

```
{ 忽略空白字符 }

procedure SkipWhite;
begin
  while IsWhite(Look) do
    GetChar;
end;
```

现在，在Match，GetName，和GetNum中加入对SkipWhite的调用。

```

{ 匹配一个特定的输入字符 }

procedure Match(x: char);
begin
  if Look <> x then Expected('' + x + '')
  else begin
    GetChar;
    SkipWhite;
  end;
end;

{ 获取一个标识符 }

function GetName: string;
var Token: string;
begin
  Token := '';
  if not IsAlpha(Look) then Expected('Name');
  while IsAlNum(Look) do begin
    Token := Token + UpCase(Look);
    GetChar;
  end;
  GetName := Token;
  SkipWhite;
end;

{ 获取一个数值 }

function GetNum: string;
var Value: string;
begin
  Value := '';
  if not IsDigit(Look) then Expected('Integer');
  while IsDigit(Look) do begin
    Value := Value + Look;
    GetChar;
  end;
  GetNum := Value;
  SkipWhite;
end;

```



这里我重新编排了一下Match的语句顺序，但没用改变其功能。

最后，我们在Init函数中需要跳过所有空白字符。

```
{ 初始化 }

procedure Init;
begin
    GetChar;
    SkipWhite;
end;
```

完成以上改动并重新编译程序。你将发现为了避免Pascal编译器的出错信息，你将不得不把Match移到SkipWhite之后。和以往那样测试程序保证它可以正常工作。

因为在这小节中我们已经做了许多改动，我重现整个语法分析程序如下：

```
program parse;

{ 声明常量 }

const TAB = ^I;
      CR = ^M;

{ 声明变量 }

var Look: char;           { 向前看字符 }

{ 从输入流中读取一个新的字符 }

procedure GetChar;
begin
    Read(Look);
end;

{ 报告一个错误 }

procedure Error(s: string);
begin
    Writeln;
    Writeln(^G, 'Error: ', s, '.');
end;

{ 报告错误然后终止程序 }

procedure Abort(s: string);
begin
    Error(s);
    Halt;
end;
```

```
end;

{ 打印预期的信息 }

procedure Expected(s: string);
begin
    Abort(s + ' Expected');
end;

{ 识别一个字母 }

function IsAlpha(c: char): boolean;
begin
    IsAlpha := UpCase(c) in ['A'..'Z'];
end;

{ 识别一个十进制数字 }

function IsDigit(c: char): boolean;
begin
    IsDigit := c in ['0'..'9'];
end;

{ 识别一个数字或者字母的字符 }

function IsAlNum(c: char): boolean;
begin
    IsAlNum := IsAlpha(c) or IsDigit(c);
end;

{ 识别加减操作符 }

function IsAddop(c: char): boolean;
begin
    IsAddop := c in ['+', '-'];
end;

{ 识别空白字符 }

function IsWhite(c: char): boolean;
begin
    IsWhite := c in [' ', TAB];
end;

{ 跳过空白字符 }
```

```

procedure SkipWhite;
begin
  while IsWhite(Look) do
    GetChar;
end;

{ 匹配一个特定的输入字符 }

procedure Match(x: char);
begin
  if Look <> x then Expected('' + x + '')
  else begin
    GetChar;
    SkipWhite;
  end;
end;

{ 获取一个标识符 }

function GetName: string;
var Token: string;
begin
  Token := '';
  if not IsAlpha(Look) then Expected('Name');
  while IsAlNum(Look) do begin
    Token := Token + UpCase(Look);
    GetChar;
  end;
  GetName := Token;
  SkipWhite;
end;

{ 获取一个数值 }

function GetNum: string;
var Value: string;
begin
  Value := '';
  if not IsDigit(Look) then Expected('Integer');
  while IsDigit(Look) do begin
    Value := Value + Look;
    GetChar;
  end;
  GetNum := Value;
  SkipWhite;

```

```

end;

{ 输出带制表符缩进的字符串 }

procedure Emit(s: string);
begin
    Write(TAB, s);
end;

{ 输出带制表符缩进和换行符 (CRLF) 的字符串 }

procedure EmitLn(s: string);
begin
    Emit(s);
    WriteLn;
end;

{ 语法分析和翻译一个标识符 }

procedure Ident;
var Name: string[8];
begin
    Name:= GetName;
    if Look = '(' then begin
        Match('(');
        Match(')');
        EmitLn('BSR ' + Name);
        end
    else
        EmitLn('MOVE ' + Name + '(PC),D0');
    end;

{ 语法分析和翻译数学Factor }

procedure Expression; Forward;

procedure Factor;
begin
    if Look = '(' then begin
        Match('(');
        Expression;
        Match(')');
        end
    else if IsAlpha(Look) then
        Ident

```

```

else
    EmitLn('MOVE #' + GetNum + ',D0');
end;

{ 识别和翻译乘法操作 }

procedure Multiply;
begin
    Match('*');
    Factor;
    EmitLn('MULS (SP)+,D0');
end;

{ 识别和翻译除法操作 }

procedure Divide;
begin
    Match('/');
    Factor;
    EmitLn('MOVE (SP)+,D1');
    EmitLn('EXS.L D0');
    EmitLn('DIVS D1,D0');
end;

{ 语法分析和翻译数学Term }

procedure Term;
begin
    Factor;
    while Look in ['*', '/'] do begin
        EmitLn('MOVE D0,-(SP)');
        case Look of
            '*' : Multiply;
            '/' : Divide;
        end;
    end;
end;

{ 识别和翻译加法运算 }

procedure Add;
begin
    Match('+');
    Term;
    EmitLn('ADD (SP)+,D0');

```

```

end;

{ 识别和翻译减法运算 }

procedure Subtract;
begin
    Match( '-' );
    Term;
    EmitLn( 'SUB (SP)+, D0' );
    EmitLn( 'NEG D0' );
end;

{ 语法分析和翻译一个表达式 }

procedure Expression;
begin
    if IsAddop(Look) then
        EmitLn( 'CLR D0' )
    else
        Term;
    while IsAddop(Look) do begin
        EmitLn( 'MOVE D0, -(SP)' );
        case Look of
            '+' : Add;
            '-' : Subtract;
        end;
    end;
end;

{ 语法分析和翻译一个赋值语句 }

procedure Assignment;
var Name: string[8];
begin
    Name := GetName;
    Match( '=' );
    Expression;
    EmitLn( 'LEA ' + Name + ' (PC), A0' );
    EmitLn( 'MOVE D0, (A0)' )
end;

{ 初始化代码 }

procedure Init;
begin

```

```

GetChar;
SkipWhite;
end;

{ 主程序 }

begin
Init;
Assignment;
If Look <> CR then Expected('NewLine');
end.

```

现在语法分析程序已经完成。它已具有我们可以放入一个直线型“编译器”的所有特征。把它收藏在一个安全的地方。下一次，我们将开始一个新的主题，但一会儿我们也仍将讨论表达式。下一部分，我打算讲述与编译器不同的解释器，并向你展示当我们改动行为的种类时语法分析器的结构变动。即使你对解释器不感兴趣，但获取这些信息为我们以后服务是很有好处的。下次再见。

4. 解释器

4.1. 简介

在前三个教程中，我们看了一下如何对数学表达式进行语法分析和编译。然后我们处理了非常简单的单个term，单个字符的表达式。最终完成了一个非常完整的语法分析器，能够对完整的赋值语句进行语法分析和翻译，而且支持多字符的token，可以跳过空白字符，以及支持函数调用。这篇教程，我将带着你再走一遍之前的旅程，只是这次我们会解释执行目标代码，而不是编译目标代码。

我们这一系列的教程不是一个编译器教程吗？为什么还要折腾解释器呢？仅仅是因为我想让你看到语法分析器的本质而已。我也想统一一下两种类型的翻译器的概念，所以你看到的不仅是两种概念的差别，更多的是它们的相似性。

考虑下面的赋值语句：

```
x = 2 * y + 3
```

在一个编译器中，我们想让目标机器的CPU去执行这个赋值语句，而且是在执行编译好的可执行程序时去执行这个赋值语句。翻译器并没有做任何数学运算…翻译器只是输出了汇编代码，然后CPU来一句一句的运行汇编代码。对于上面的例子，编译器将输出计算等号右边的表达式的汇编代码，然后将计算结果保存到变量 `x` 中。

对于解释器而言，不会有汇编代码产生。表达式的计算是立即进行的，也就是边进行语法分析，边进行计算。对于上面的例子，当赋值语句的语法分析进行完毕，`x` 就会有一个新的值了。

我们整个教程讨论的编译方法其实叫做“语法制导翻译”（syntax-driven translation）。你现在其实也发现了，语法分析器的程序结构和BNF语法是非常相似的。我们构建的语法分析器程序识别了BNF中定义的每一条语法规则。和每一个函数对应的是一个BNF中的一个“动作（action）”。当我们碰到一个动作时，就为它编写一个程序。在我们现在编写的编译器程序中，每一个动作对应的程序都会输出汇编代码。这些汇编

代码由目标机器的CPU执行。在一个解释器中，对于每一个动作而言，我们都会立即去解释执行这个动作。

我想让你看到的就是，编写解释器时，我们的语法分析器的程序结构并不会改变。只是程序中的动作变了，不是输出汇编代码，而是解释执行程序。所以如果你可以为一个编程语言写一个解释器，那么你就能为这个编程语言写一个编译器，反过来也是如此。当然，你也会看到它们之间的差异。因为动作是不一样的，识别程序也会有所不同。特别是，在一个解释器中，识别程序是一个函数，这个函数会返回给调用它的程序一个数值（解释执行的结果）。而我们之前编译器中的语法分析器程序并没有这样做。

事实上，我们的编译器是一个“纯粹的”编译器。每当一个BNF的语法构建规则被识别时，目标机器的汇编代码立即就生成了。（这也是生成的汇编代码不是很高效的一个原因）。我们这里构建的解释器是一个纯粹的解释器，也就是说没有任何的翻译过程，例如针对源代码进行“词法分析”。这样就展示了除了翻译器的两个极端。在真实的生产环境中，翻译器并不会如此的纯粹，而是既有编译的技术又有解释的技术。

我可以举几个例子。我已经提过一个了：大部分解释器，例如微软的BASIC解释器，将会把源代码进行词法分析，然后翻译成一种中间表示形式，这样就可以很容易地进行实时的解释运行。

另一个例子是汇编器（assembler）。汇编器的目标是生成目标机器的二进制的机器代码，通常的做法是针对每一行汇编代码转换成二进制机器代码。但几乎每个汇编器都允许表达式作为参数传递。在这种情况下，表达式通常是常量表达式，所以显然汇编器不会为常量表达式生成二进制代码。而是解释执行表达式，然后将计算结果直接作为二进制代码输出。

事实上，我们也可以运用一点解释器技术。我们在之前所构建的翻译器针对一些复杂的表达式生成了汇编代码，即使每一个term都是常量也是如此。在这种情形下，我们完全可以使用解释器技术来将全是常量的表达式计算成一个常量结果，再输出汇编代码。

在编译器理论中有一个叫做“惰性”翻译的概念。这个理念主要的意思就是没必要针对每一个动作都生成汇编代码。事实上，极端一点，你可以不生成任何汇编代码，除非在你真正需要产生汇编代码的时候。为了实现这一点，和语法分析过程所关联的BNF中的动作，并不仅仅会生成汇编代码。有时候会生成，而经常情况下它们仅仅将信息返回给调用者。有了这些信息，调用者可以针对接下来的事情作出更好的决策。

例如，给定如下语句：

```
x = x + 3 - 2 - (5 - 4)
```

我们之前写的编译器将会为这个语句生成18条汇编语句：将每一个参数加载到寄存器中，执行算术运算，以及保存计算结果。而一个惰性求值将会识别出算术运算中包含的常量的计算可以在编译期完成求值。然后将表达式归约为以下形式：

```
x = x + 0
```

一个更加惰性的求值将会更加聪明，会识别出上面的语句等价于以下语句：

```
x = x
```

而上面的这条语句什么也没有做。所以我们可以将18条汇编语句化简为0条汇编语句！

注意以上的优化在我们的编译器中是无法起作用的。因为我们的每一个动作都立即生成了汇编代码。

比起我们的编译器，惰性表达式求值策略可以产生好的多的汇编代码。但我要警告你，这样的策略将会极大的增加我们的语法分析器的代码的复杂度，因为每一个过程都要决定是否输出汇编代码。惰性求值其名字的来源并不是因为编译器的开发者可以偷懒了！

因为我们一直遵循KISS原则。所以我不想深入探讨这个话题。我只想让你明白将编译技术和解释执行技术结合起来可以做很多的优化。你需要知道的是在语法分析的过程中，一个更加聪明的翻译器将会把东西返回给它们的调用者，然后期待事情不会被搞砸。这是本节教程要过一遍解释器技术的主要原因。

4.2. 解释器

好，你现在知道我们为什么要讲解释器了。我们现在开干吧。为了更好的实践，我们准备重新构建一组摇篮代码然后重新写一遍翻译器。这次，我们可以写的快一点。

因为我们需要做算术计算，所以首先我们先修改GetNum方法，到现在为止，这个方法只能返回一个字符或者字符串。现在，它可以返回一个整型数值。将摇篮代码拷贝一份，而不要直接修改原来的摇篮代码！然后代码如下：

```
function GetNum: integer;
begin
  if not IsDigit(Look) then Expected('Integer');
  GetNum := Ord(Look) - Ord('0');
  GetChar;
end;
```

现在，重写Expression方法：

```
function Expression: integer;
begin
  Expression := GetNum;
end;
```

最后，在main函数的最后插入以下语句，然后编译并测试。

```
WriteLn(Expression);
```

现在语法分析器程序可以将一个单字符的整数进行语法分析然后翻译一个整数表达式了。当然，你需要保证程序能够对数字 0..9 都运行正确。然后输入其他字符，可以报错。不需要你再做任何其他事情了。

接下来，让我们扩展上面的程序，可以处理加减运算符。修改Expression：

```

function Expression: integer;
begin
  if IsAddop(Look) then
    Value := 0
  else
    Value := GetNum;
  while IsAddop(Look) do begin
    case Look of
      '+': begin
        Match('+');
        Value := Value + GetNum;
      end;
      '-': begin
        Match('-');
        Value := Value - GetNum;
      end;
    end;
  end;
  Expression := Value;
end;

```

表达式的结构和我们之前写的编译器是平行的，所以我们不需要花很多时间来做调试。我们已经有了很大的进展了，不是吗？Add和Subtract方法都被去掉了。因为在我们上面的程序中，已经处理了加减法。我们其实可以把这两个方法保留下来，然后将表达式所需要的参数传进去，然后求值，求得的值就是Value。但是对我来说，将Value作为一个局部变量，实现起来更加的清晰。我们把Add和Subtract方法中的代码放进Expression中了。这个结果表明了，当我们把我们简单的翻译方案实现为以上清晰而优美的代码时，我们可能就不需要和惰性求值打交道了。这点我们可能需要为了后面的内容而记在心里。

好，翻译器能工作吗？让我们进行下一步。我们的Term程序修改起来也不会太难。在Expression中，将每个对GetNum的调用都改为对Term的调用，Term代码如下：

```

function Term: integer;
var Value: integer;
begin
  Value := GetNum;
  while Look in ['*', '/'] do begin
    case Look of
      '*': begin
        Match('*');
        Value := Value * GetNum;
      end;
      '/': begin
        Match('/');
        Value := Value div GetNum;
      end;
    end;
  end;
  Term := Value;
end;

```

现在再试一下。不要忘记两件事情：首先，我们处理的是整数的除法，例如， $1/3$ 应该求值为0。第二，即使我们输出的是多数字的结果，我们的输入也应该限制在单个数字。

这貌似是一个非常愚蠢的限制，因为我们已经知道GetNum的扩展是非常容易的。所以让我们进一步，把这个限制去掉。新版本的GetNum如下：

```

{ 获取数值 }

function GetNum: integer;
var Value: integer;
begin
  Value := 0;
  if not IsDigit(Look) then Expected('Integer');
  while IsDigit(Look) do begin
    Value := 10 * Value + Ord(Look) - Ord('0');
    GetChar;
  end;
  GetNum := Value;
end;

```

如果你已经编译并且测试了这个版本的解释器，那么下一步就是编写Factor函数了，以及可以处理带括号的表达式。我们现在还不打算处理变量名的问题。首先，在Term函数中修改对GetNum的调用，让我们可以直接调用Factor。现在，编写新版本的Factor程序：

```

{ 对数学因子 (Factor) 进行语法分析和翻译 }

function Expression: integer; Forward;

function Factor: integer;
begin
  if Look = '(' then begin
    Match('(');
    Factor := Expression;
    Match(')');
    end
  else
    Factor := GetNum;
  end;

```

很简单吧？我们已经写出了一个能用的解释器了！

4.3. 一点简单的哲学

在往前开车之前，有些事情我想多说两句，注意了啊！有一个概念，我们一直在教程中使用，但我从来没有明确的提过。我想现在是时候了，因为这个概念非常有用，而且非常的强大，就是这个概念将一个简单的语句解析器和一个超级复杂的语句解析器区分开来。

在编译器技术的早些时候，人们有一个非常痛苦的时期，就是如何处理类似于运算符优先级之类的事情…例如乘法和除法的优先级比加法和减法的优先级要更高。我还记得一个三十年前的大学同学，他搞定如何处理优先级的时候，简直不要太开心！他使用了两个栈，然后将运算符和操作数各种压栈。每个运算符都会关联一个优先级，规则要求我们如果栈顶的运算符的优先级是正确的，我们就可以执行运算（将栈进行归约）。为了让生活更有趣一些，像) 这样的运算符也是有自己的优先级的，取决于这个运算符是否在栈上。在将这个运算符放到栈上之前，我们必须给这个运算符一个值，然后还得给这个运算符另一个值，来决定什么时候将运算符从栈顶拿走。为了体验一下，很多年前我自己也实现了一下上面这种技术，我可以告诉你，很蛋疼。

我们现在不再需要上面这坨事儿了。事实上，现在我们发现算术语句的语句解析跟小孩子玩游戏一样简单。为什么我们如此幸运？优先级的栈跑哪儿去了？

在我们的解释器实现里面，也有相似的事情。我们已经知道为了对算术语句做计算（和对它们的语句解析对应），需要将数值压栈到某一个栈上。那这个栈在哪里呢？

在编译器的教材里面，有很多地方讨论了栈和其他数据结构。在另一种主流的语句解析技术（LR）中，需要显式的使用栈。事实上，LR技术很像之前我们讨论过的我大学同学使用过的很麻烦的技术。另一个概念就是语句分析树的概念。教材的作者们很喜欢画一个语句中的token所组成的流程图。然后将token画进一颗树形结构中，运算符是树形结构的内部节点，数值是叶子节点。在我们使用的技术里面，栈和树形结构在哪里呢？我们之前的实现没有看到任何以上这些结构。答案就是，这些结构都是隐形存在的，并没有显式的编写代码。在每个编程语言的实现里面，我们每次调用一个子程序，都需要栈的参与。每当一个子程序被调用，子程序的返回地址就会压入到CPU栈的栈顶位置。当子程序调用结束时，返回地址将被弹出，然后控制流就转到了原来调用子程序的位置后面。在一个支持递归的编程语言中，例如Pascal，还需要将局部数据压到栈顶上面，当然也会在需要的时候弹出栈。

举个例子，Expression函数包含了一个局部变量叫做Value，当我们调用Term函数时，会用到这个变量。假设，在下一次调用Term函数时，Term函数会调用Factor函数，而Factor函数会再一次递归的调用Expression。而Expression函数的一个实例会获取一个Value的拷贝，那第一个Value会怎么样呢？答案：第一个Value还在栈上面，而且一直会在栈上，直到我们的调用返回，才会从栈上弹出。

换句话说，我们的代码看起来如此简单的原因是因为我们最大限度的使用了Pascal编程语言的资源。层级结构（栈）和语法解析树一直都在那里，只是他们被隐藏在了语法解析器的结构中（递归调用在底层会使用栈）。也就是说我们的函数调用序列这个过程的底层，有栈和语法解析树这样的东西。我们之前已经这么写代码了，所以你很难想象还有其他方式来实现语法解析器。但我要告诉你，我们之前的写法耗费了研究编译器理论的科学家很多年，才发明出我们在教程中的那种写法。早期的编译器的实现复杂的无法想象。应该庆幸现在的写法是多么的简单啊！

我讲上面的这些东西，既是一个收获也是一个警告。收获：当我们正确的做事情时，一切可以很简单。警告：好好看看你写的代码。如果你自己折腾一个编译器，然后发现需要引入栈或者树这样的数据结构，那么你就要思考一下，你是在按照正确的方法写代码吗？可能你并没有用到编程语言给你提供的很多功能，而是自己在造轮子。

下一步就是添加变量名这样的特性了。现在我们有一个小问题需要解决。对于编译器的编写来讲，处理变量名这个问题很简单…我们直接在汇编代码中生成变量名就好了，而对变量名所需要的内存分配，我们教给了程序的其他部分。在这篇教程中，我们需要能够根据变量名来获取所对应的值，然后在Factor函数中返回值时，将变量名所对应的值作为返回值返回。

在个人计算机的早期时代，Tiny BASIC解释器就已经存在了。Tiny BASIC解释器拥有26个可能的变量：每个变量都是26个字母中的一个。这正好和我们的单字符变量名很吻合，所以我们打算使用同样的技巧。在我们实现的解释器的开始，也就是在Look变量的后面，插入下面的语句：

```
Table: Array['A'..'Z'] of integer;
```

我们需要初始化这个数组，所以添加下面的函数：

```
{ 初始化变量 }

procedure InitTable;
var i: char;
begin
  for i := 'A' to 'Z' do
    Table[i] := 0;
end;
```

我们需要在Init函数中插入对InitTable的调用。别忘记做这个，否则会报错！

现在我们拥有了一个变量的数组，我们可以通过修改Factor来使用这个数组。由于我们并没有对变量进行赋值，所以Factor函数针对这些变量名会一直返回0，让我们再对代码扩展一下。下面是代码的新版本：

```

{ 对数学因子 (Factor) 进行语法分析和翻译 }

function Expression: integer; Forward;

function Factor: integer;
begin
  if Look = '(' then begin
    Match('(');
    Factor := Expression;
    Match(')');
    end
  else if IsAlpha(Look) then
    Factor := Table[GetName]
  else
    Factor := GetNum;
end;

```

编译然后测试这个版本的代码。即使所有的变量现在都是0,但至少我们可以正确的对整个表达式进行语法解析，也会对错误的表达式报错。

我想你已经意识到下一步要做什么了：我们需要添加赋值语句，这样我们可以对变量进行赋值。现在，让我们专注在单行赋值语句，很快我们就会处理多行语句。

赋值语句和我们之前在编译器中编写的处理赋值语句的代码很相似：

```

{ 对赋值语句进行语法分析和翻译 }

procedure Assignment;
var Name: char;
begin
  Name := GetName;
  Match('=');
  Table[Name] := Expression;
end;

```

为了测试，我在main函数中添加了一个临时的写操作语句，用来打印A的值。然后我对各种各样的赋值语句进行了测试。

当然，一个解释型的编程语言只能接收一行程序没什么太大的价值。我们需要能够处理多行语句。这只需要我们对调用Assignment方法的代码包裹一层循环就可以了。让我们现在实现一下。但是循环退出条件是什么呢？很高兴你提出了这个问题，因为我们之前写代码一直忽略了这一点。

在任何翻译器中，最难处理的事情之一就是何时跳出给定的结构 (for, while, if)，然后跳转到另一个地方。目前这还是一个问题，因为我们并没有引入任何控制结构。只引入了表达式和赋值语句。当我们添加控制结构时，我们需要非常小心，来让控制结构正确的结束执行。如果我们将解释器的代码放入一个循环中

，我们需要一种方法来退出循环。在输入中换行来结束循环不是一种好的方法，因为我们换行应该是到下一行才对。我们可以输入一个语法解析器无法识别的字符，来让我们退出程序，但以报错的方式退出程序，看起来也不是很爽。

我们需要的是一个终止符号。我喜欢Pascal的终止符号`.`。一个稍微复杂一点的地方在于，Turbo Pascal在每一行的末尾会有两个字符，一个是`CR`字符，一个是`LF`字符。在每行的末尾，我们需要消费掉这两个字符，才能处理下一行的内容。我们可以在Match方法中做这件事情，只是Match的报错信息会打印字符，而打印`CR`和`LF`字符看起来不是个好主意。所以我们需要一个特定的程序来处理这种情况，因为这个处理程序我们会反复使用很多次。下面是代码：

```
{ 识别然后跳过CRLF字符 }

procedure NewLine;
begin
  if Look = CR then begin
    GetChar;
    if Look = LF then
      GetChar;
  end;
end;
```

将上面的程序放在一个合适的地方吧…我把它放在了Match函数的后面。现在，重写Main函数如下：

```
{ 主程序 }

begin
  Init;
  repeat
    Assignment;
    NewLine;
  until Look = '.';
end.
```

注意对`CR`字符的测试这里没了，而在NewLine函数中，也没有错误处理。很好，尽管…在下一个赋值语句的开始，会捕获剩余的输入字符。

好的，现在我们已经有一个可以工作的解释器了。但并不是太好，因为我们还没办法读取数据或打印数据。所以我们必须有一些I/O功能。

让我们总结一下这节课，然后添加I/O程序。由于我们始终坚持单字符的token，我将使用`?`符号来等待输入语句，使用`!`来输出。这两个符号后面输入的单字符符号作为“参数列表”，下面是程序：

```

{ 输入程序 }

procedure Input;
begin
    Match('?');
    Read(Table[GetName]);
end;

{ 输出程序 }

procedure Output;
begin
    Match('!');
    WriteLn(Table[GetName]);
end;

```

上面的代码并不是很惊艳，我承认…例如，在输入时没有提示符，但至少能工作。

main函数中对应的改变在下面的代码中。可以看到我们使用了基于Look的case语句，来决定下一步干什么。

```

{ 主程序 }

begin
    Init;
    repeat
        case Look of
            '?': Input;
            '!': Output;
            else Assignment;
        end;
        NewLine;
    until Look = '.';
end.

```

我们现在已经完成了一个真正的可以工作的解释器了。解释器包括三种类型的程序语句，26个变量名，以及I/O语句。它所欠缺的是控制语句，子程序，以及函数功能。因为我们并不是要构建一个产品级的解释器，而仅仅是为了学习，所以以上部分差不多就够了。下一节课我们将研究如何编写控制语句的编译器代码。后面还会编写子程序的编译器代码。我很想马上开始，所以我们先把这个粗糙的解释器留在这个地方吧。

我希望现在你应该知道单字符变量名和空格的处理是很简单的，上一节课我们已经讲过了。这次，如果你想扩展以上解释器，它们应该是“课后作业”了。下次见。