USTC

Chapter 6
Writing a Program

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章和下一章将通过一个“桌面计算器”来描述程序设计的主要过程
  - 软件开发的一些思想 (principle)
  - 计算器的设计思路 (idea)
  - 使用文法 (grammar)
  - 表达式计算 (expression)
  - 程序组织 (organization)

建立一个程序

- 分析 Analysis
  - 精炼我们对问题的理解
    - 思考一下，程序最终的用途是什么
- 设计 Design
  - 创建程序的总体结构
- 实现 Implementation
  - 编写代码
  - 调试
  - 测试
- 必要时，重复以上过程……

编写一个程序：策略

- 要解决的问题是什么?
  - 问题定义清楚了吗?
  - 看上去问题是可以处理的，但时间、技巧和工具是否足够?
- 将程序划分为可分别处理的多个部分
  - 你知道有哪些工具、函数库或者其他辅助手段吗
    - 是的，我们前面已经用到了: **iostreams**, **vector**, etc.
- 实现一个小的、有限的版本来解决问题的关键部分
  - 引出我们在理解、思想和工具中存在的问题
  - 看看能否改变问题描述的一些细节，使其更加容易处理
- 如果它不能工作
  - 抛弃第一个版本，实现另外一个有限的版本
  - 继续实现它，直到获得能够令我们满意的一个版本
- 实现一个完整的解决方案
  - 最好能够利用最初版本中的组件

编写一个程序：示例

- 分步建立一个程序，这一过程中，我们会犯一些常见的错误
  - 即使是有经验的程序员也会犯错 (mistake)
    - 错误可能很多，但它是我们学习必不可少的部分
  - 设计一个好的程序，本质上说是困难的
  - 让编译器去检查一些明显的错误，而无需第一次就尽量做好每一个细节
    - 将主要精力放在重要的设计选择上
  - 建立一个简单、不完整的版本进行实验，并从中获得反馈信息
    - 好的程序是“不断增长的”

一个简单的计算器

- 从键盘给定表达式作为输入，计算表达式的值并返回结果
  - For example
    - Expression: 2+2
    - Result: 4
    - Expression: 2+2*3
    - Result: 8
    - Expression: 2+3-25/5
    - Result: 0

需求的简单描述

  - 有利于我们理解程序设计过程和编译系统的原理

- 下面，让我们更细化一些……

伪代码

❖ 第一个思路：

```
int main()
{
   variables                          // pseudo code
   while (get a line) {               // what's a line?
         analyze the expression       // what does that mean?
         evaluate the expression
         print the result
   }
}
```

用程序表达人的思想

- 如何将45+5/7 表示为数据？
- 如何从输入字符串中获取 45 + 5 / 和 7？
- 如何保证45+5/7 的意思是45+(5/7) 而不是(45+5)/7？
- 允许浮点数计算吗？ (sure!)
- 允许使用变量吗？ v=7; m=9; v*m ？

一个简单的计算器

- 等一下！
  - 只是重新发明一个车轮！
  - 课本Chapter 6上有更多“dead-end”方式的示例
- 专家们会做什么呢？
  - 计算机计算这种表达式已经超过50年
  - 一定有一个成熟的解决方案！
  - 专家们以前是怎么做的呢？
    - 阅读代码对你是有益的
    - 咨询有经验的朋友/同学/老师可能是一种更有效的方式，而不是自己进行艰难摸索

我们提倡：

- 对程序设计而言，交流并借鉴已有方案是最有效的方式！

表达式— 文法

❖ 这是专家们经常使用的东东 — 写一个文法 *grammar*：

Expression :
- **Term**
- **Expression ‘+’ Term** — *e.g.*, **1+2**, **(1-2)+3**, **2*3+1**
- **Expression ‘-’ Term**

Term :
- **Primary**
- **Term ‘*’ Primary** — *e.g.*, **1*2**, **(1-2)*3.5**
- **Term ‘/’ Primary**
- **Term ‘%’ Primary**

Primary :
- **Number** — *e.g.*, **1**, **3.5**
- **‘(‘ Expression ‘)’** — *e.g.*, **(1+2*3)**

Number :
- **floating-point literal** — *e.g.*, **3.14**, **0.274e1**, or **42** – as defined for C++

程序可基于分词 (Token) 进行构建 (*e.g.*, 数字和运算符)

文法简述

- 什么是文法 *grammar*?
  - 是关于表达式的一个语法规则集合
  - 规则描述了如何分析一个表达式
  - 文法看起来好像没有意义？(对人)
    - 比如，你知道下面是正确的：
      - **2*3+4/2**
      - **birds fly but fish swim**
    - 你知道下面是错误的：
      - **2 * + 3 4/2**
      - **fly birds fish but swim**
  - 那么，它为什么是正确的(错误的)呢？
  - 你是如何知道的呢？
  - 我们如何让计算机知道呢？

“English”文法示例

语法分析：从顶层开始，搜索与输入单词匹配的规则.....

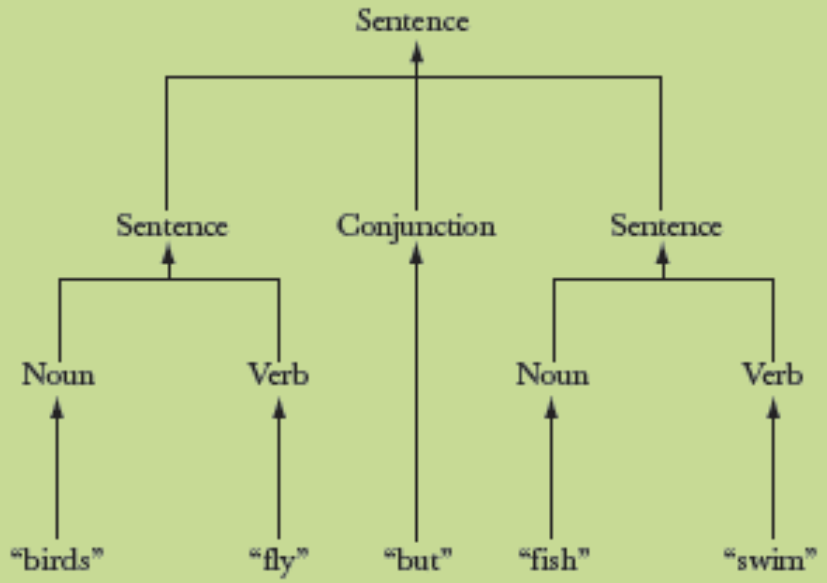
Parsing a simple English sentence

Sentence :
- Noun Verb
- Sentence Conjunction Sentence

Conjunction :
- “and”
- “or”
- “but”

Noun :
- “birds”
- “fish”
- “C++”

Verb :
- “rules”
- “fly”
- “swim”



表达式文法 — “2”

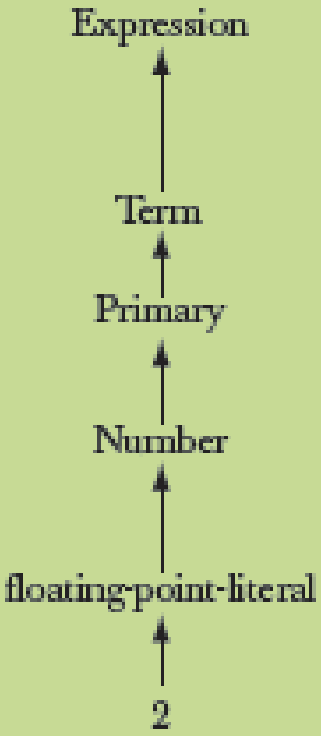
Parsing the number 2

Expression:
- Term
- Expression “+” Term
- Expression “–” Term

Term:
- Primary
- Term “*” Primary
- Term “/” Primary
- Term “%” Primary

Primary:
- Number
- “(” Expression “)”

Number:
- floating-point-literal



表达式文法 — “2+3”

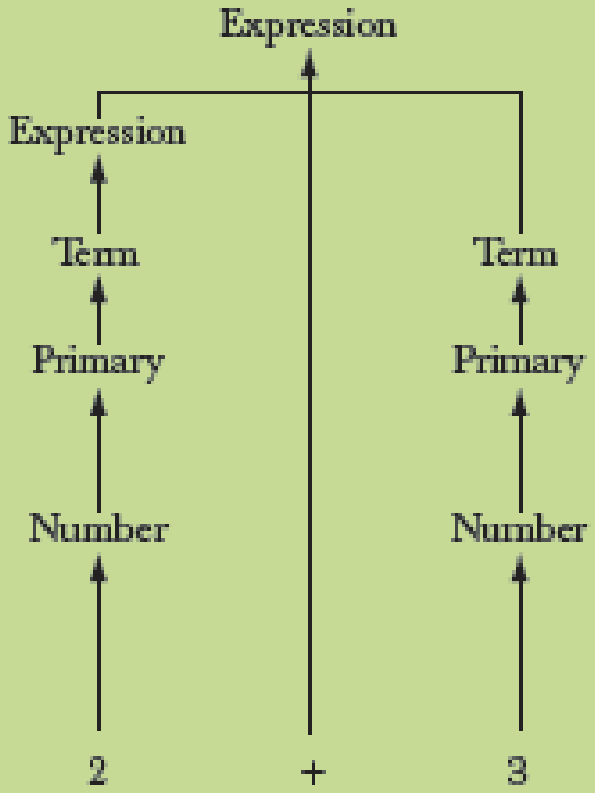
Parsing the expression 2 + 3

Expression:
- Term
- Expression “+” Term
- Expression “–” Term

Term:
- Primary
- Term “*” Primary
- Term “/” Primary
- Term “%” Primary

Primary:
- Number
- “(” Expression “)”

Number:
- floating-point-literal



表达式文法 — “45+11.5*7”

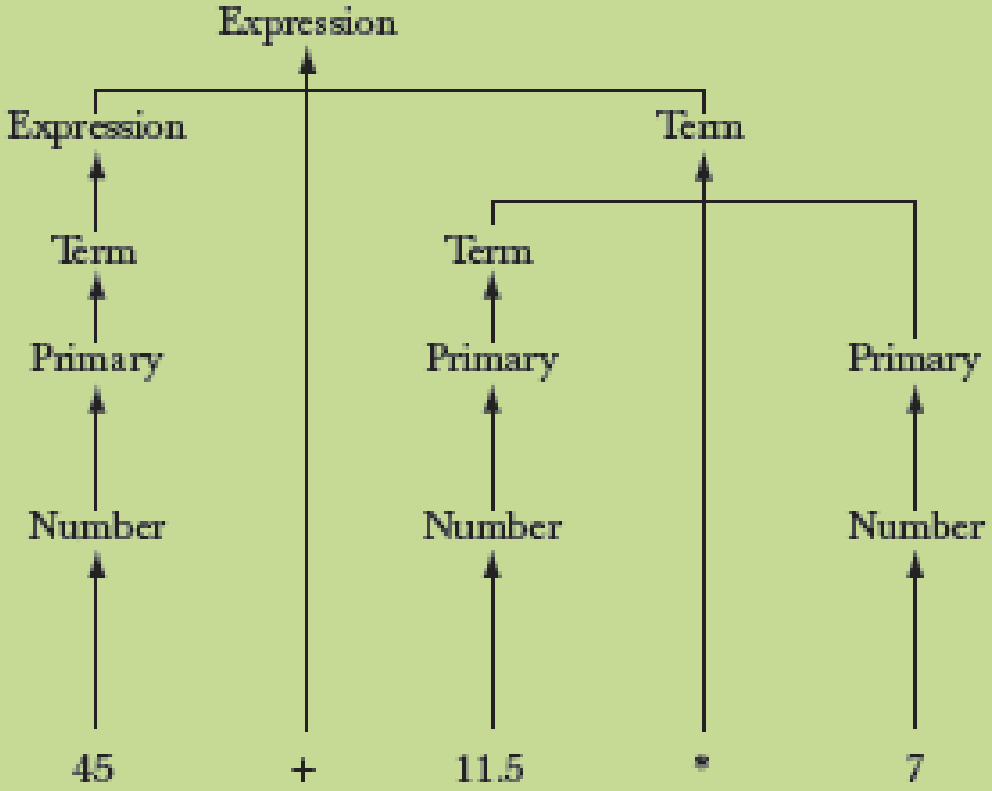
Parsing the expression 45 + 11.5 * 7

Expression:
- Term
- Expression “+” Term
- Expression “–” Term

Term:
- Primary
- Term “*” Primary
- Term “/” Primary
- Term “%” Primary

Primary:
- Number
- “(” Expression “)”

Number:
- floating-point-literal



分析函数

我们需要函数来匹配文法规则：

```
get()        // read characters and compose tokens
             // calls cin for input

expression() // deal with + and –
             // calls term() and get()

term ()      // deal with *, /, and %
             // calls primary() and get()

primary()    // deal with numbers and parentheses
             // calls expression() and get()
```

注意：

每个函数只处理表达式指定的部分，而将其余部分留给其他的函数进行处理——这能够从根本上简化每个函数

类比：多人在处理一个复杂的问题时，每个人只处理自己擅长的问题，而将其余的留给自己的合作伙伴

函数返回类型

- 分析函数应该返回什么呢？
  - 计算结果？

```
Token get();          // read characters and compose tokens
double expression();  // deal with + and –
                      //        return the sum (or difference)
double term ();       // deal with *, /, and %
                      //        return the product (or ...)
double primary();     // deal with numbers and parentheses
                      //        return the value
```

- 上面的Token代表什么呢？

What is a token?

- 我们期望把输入作为一个 tokens 流
  - 读取字符串 **1 + 4*(4.5-6)** (总共13 个字符，包括2个空格)
  - 有 9 个 tokens： **1 + 4 * (4.5 - 6)**
  - 6 种类型的tokens： number + * (-)
- 每个 token 有两个部分
  - 类型“kind”， e.g., number
  - 值 value， e.g., **4**
- 我们需要一个类型来表示上述“Token”的想法
  - 将在下一章建立这样一个类型，现在我们只需：
    - **get_token()** 从输入中获取下一个
    - **t.kind** 提供token的类型
    - **t.value** 提供token的值

处理 + 和 -

Expression:
    Term
    Expression ‘+’ Term   // *Note: every Expression starts with a Term*
    Expression ‘-’ Term

将递归转换为循环进行处理

```
double expression()          // read and evaluate: 1   1+2.5   1+2+3.14  etc.
{
    double left = term();                       // get the Term
    while (true) {
        Token t = get_token();                  // get the next token…
        switch (t.kind) {                       // … and do the right thing with it
            case '+':       left += term(); break;
            case '-':       left -= term(); break;
            default:        return left;        // return the value of the expression
        }
    }
}
```

处理 *, / 和 %

```
double term()       // exactly like expression(), but for *, /, and  %
{
    double left = primary();            // get the Primary
    while (true) {
        Token t = get_token();                  // get the next Token…
        switch (t.kind) {
        case '*':     left *= primary(); break;
        case '/':     left /= primary(); break;
        case '%':   left %= primary(); break;
        default:      return left;              // return the value
        }
    }
}
```

- Oops: 不能编译通过
  - % 对浮点数没有定义!

仅处理*和/

Term :
    Primary
    Term '*' Primary   // *Note: every Term starts with a Primary*
    Term '/' Primary

```
double term()       // exactly like expression(), but for *, and /
{
    double left = primary();            // get the Primary
    while (true) {
        Token t = get_token();                  // get the next Token
        switch (t.kind) {
        case '*':    left *= primary(); break;
        case '/':    left /= primary(); break;
        default:     return left;               // return the value
        }
    }
}
```

先去掉%运算，下一章再考虑该问题

处理除“零”问题

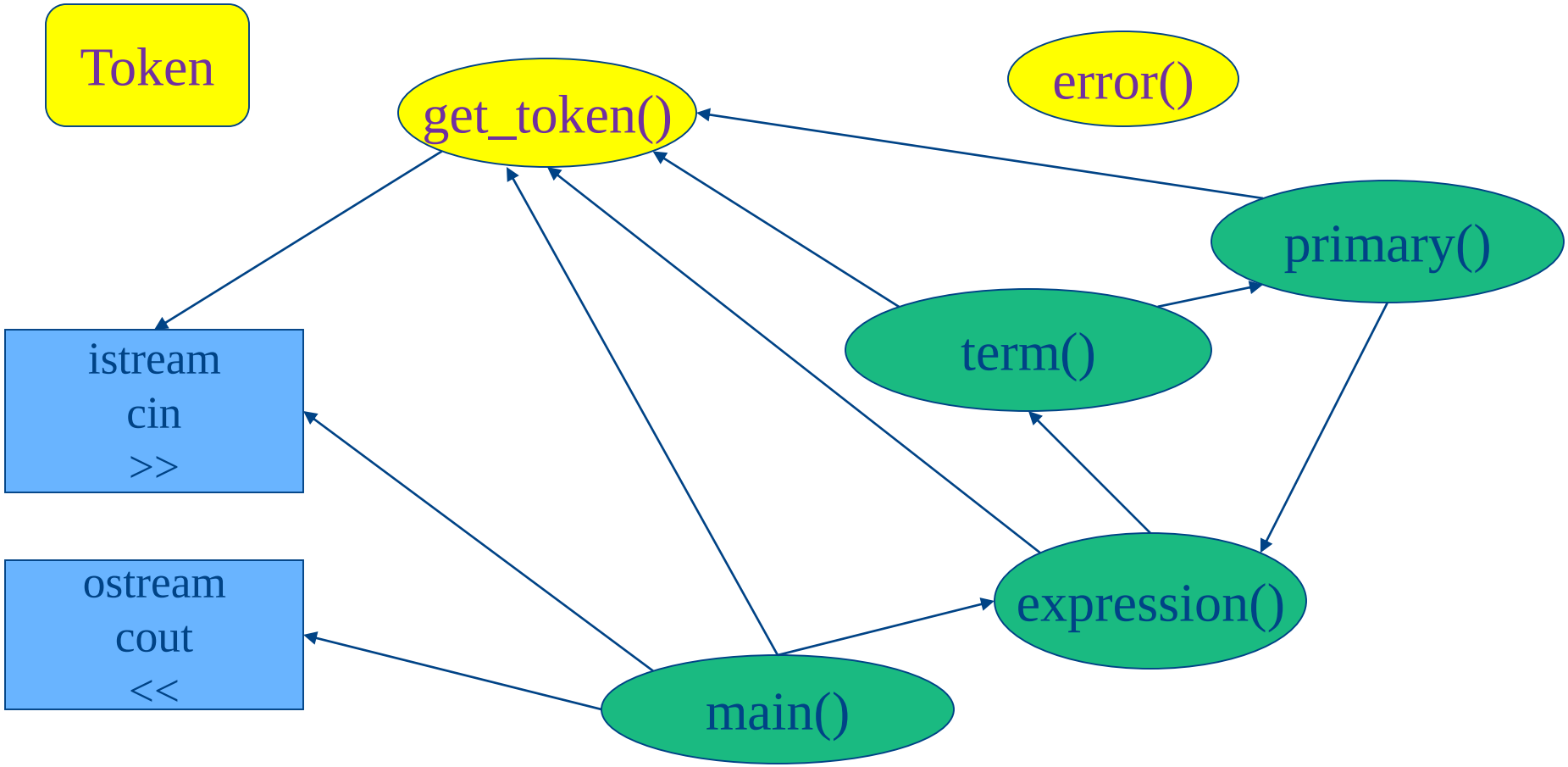
```
double term()          // exactly like expression(), but for * and  /
{
    double left = primary();                    // get the Primary
    while (true) {
        Token t = get_token();                  // get the next Token
        switch (t.kind) {
        case '*':
                left *= primary();
                break;
        case '/':
                double d = primary();
                if  (d==0) error("divide by zero");
                left /= d;
                break;

        default:
                return left;                    // return the value
        }
    }
}
```

报错功能，在头文件中有定义

处理数字和括号

```
double primary()     // Number or '(' Expression ')'
{
    Token t = get_token();
    switch (t.kind) {
    case '(':                                   // handle '('expression ')'
        double d = expression();
        t = get_token();
        if (t.kind != ')') error("')' expected");
        return d;
    case '8':                   // we use '8' to represent the "kind" of a number
        return t.value;         // return the number's value
    default:
        error("primary expected");
    }
}
```

程序的组织



- 谁调用了谁？注意，有一个循环

程序全貌

```
#include "std_lib_facilities.h"

// Token stuff (explained in the next lecture)

double expression(); // declaration so that primary() can call expression()

double primary() { /* ... */ } // deal with numbers and parentheses
double term() { /* ... */ }          // deal with * and / (pity about %)
double expression() { /* ... */ }    // deal with + and –

int main() { /* ... */ }             // on next slide
```

main( )函数

```
int main()
try {
    while (cin)
        cout << expression() << '\n';
    keep_window_open();                    // for some Windows versions
}
catch (runtime_error& e) {
    cerr << e.what() << endl;
    keep_window_open ();
    return 1;
}
catch (…) {
    cerr << "exception \n";
    keep_window_open ();
    return 2;
}
```

运行——诡异

运行程序后，输入及结果：

- 2
-
- 3
- 4
- 2　　an answer
- 5+6
- 5　　an answer
- X
- Bad token　　an answer (finally, an expected answer)

运行——诡异

- 1 2 3 4+5 6+7 8+9 10 11 12
- 1 an answer
- 4 an answer
- 6 an answer
- 8 an answer
- 10 an answer

- Aha! 我们的程序“吃掉”了三个表达式中的两个
  - 接下来怎么办?
  - 重新看一下expression( )函数

处理 + 和 -

Expression:
    Term
    Expression ‘+’ Term   *// Note: every Expression starts with a Term*
    Expression ‘-’ Term

```
double expression()          // read and evaluate: 1   1+2.5   1+2+3.14  etc.
{
    double left = term();                          // get the Term
    while (true) {
        Token t = get_token();                     // get the next token…
        switch (t.kind) {                          // … and do the right thing with it
        case '+':     left += term(); break;
        case '-':     left -= term(); break;
        default:      return left;                 // <<< doesn't use "next token"
        }
    }
}
```

获取一个其他字符后，直接返回，该字符后续无法再使用

处理 + 和 -

- 因此，需要一种方式，能够将token能够“吐”回去！
  - “吐”回哪里去?
  - “输入”当然可以，那么，我们需要tokens的输入流来实现

```
double expression()          // deal with + and -
{
    double left = term();
    while (true) {
        Token t = ts.get();      // get the next token from a "token stream"
        switch (t.kind) {
        case '+':     left += term(); break;
        case '-':     left -= term(); break;
        default:      ts.putback(t);  // put the unused token back
                      return left;
        }
    }
}
```

返回到输入流中，其他函数可以继续使用！

处理*和/

- 对term( )做相同的修改

```
double term()          // deal with * and /
{
    double left = primary();
    while (true) {
        Token t = ts.get();     // get the next Token from input
        switch (t.kind) {
        case '*':
                // deal with *
        case '/':
                // deal with /
        default:
                ts.putback(t);          // put unused token back into input stream
                return left;
        }
    }
}
```

程序

- 它能够“工作一点点”了
  - 我们使用第一个测试还不错
    - 那么，第二个呢?
    - 那么，第四个呢? (教材上有实际结果)
  - 然而，“工作一点点”是不够的!
  - 当程序开始能够“工作一点点”时，我们真正有趣的编程工作才刚刚开始
- 现在，回过头来重新看一下!

再运行一次

- 2 3 4 2+3 2*3
- 2 an answer
- 3 an answer
- 4 an answer
- 5 an answer

❖ 怎么了？第6个呢？

- 程序看起来“预测”下一个token的到来
  - 它在等待用户输入
- 那么，我们引入一个“print result”命令
- 同时也引入一个“quit”命令

main( ) 函数

```
int main()
{
    double val = 0;
    while (cin) {
        Token t = ts.get();     // rather than get_token()
        if (t.kind == 'q') break;                    // ‘q’ for “quit”
        if (t.kind == ';')                           // ‘;’ for “print now”
            cout <<  val << '\n';  // print result
        else
            ts. putback(t);          // put a token back into the input stream
        val = expression();     // evaluate
    }
    keep_window_open();
}
// … exception handling …
```

开始使用计算器

- 现在，计算器程序能够使用了(最简单的方式)
  - 2;
  - 2　　an answer
  - 2+3;
  - 5　　an answer
  - 3+4*5;
  - 23　　an answer
  - q

在实际程序中，这还不够.....

Next

- 完成这个程序
  - 定义Tokens
  - 错误恢复
  - 清理代码
  - 代码 review
  - 测试