

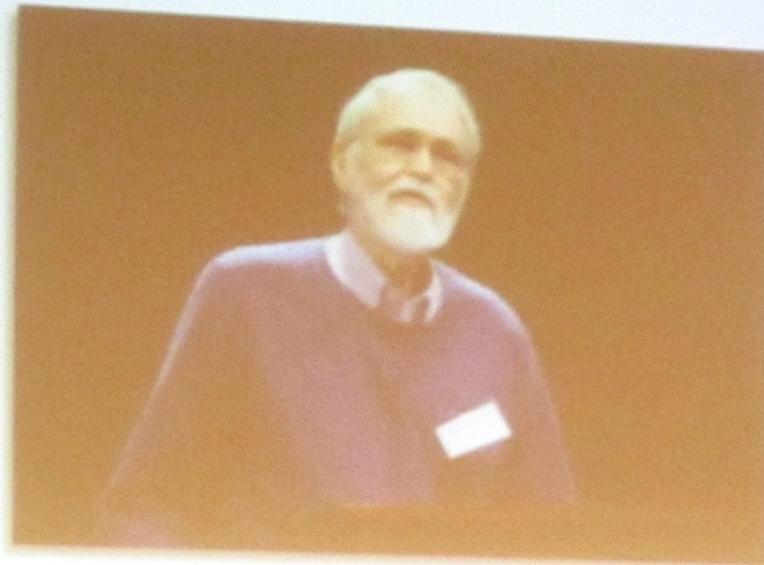
问题求解与实践 ——实现一个计算器

主讲教师：陈雨亭、沈艳艳

Overview

- Some thoughts on software development
- The idea of a calculator
- Using a grammar
- Expression evaluation
- Program organization

`/* You are Not Expected
to Understand This */`



Brian Kernighan
Princeton University

Building a program

- Analysis
 - Refine our understanding of the problem
 - Think of the final use of our program
- Design
 - Create an overall structure for the program
- Implementation
 - Write code
 - Debug
 - Test
- Go through these stages repeatedly

Writing a program: Strategy

- What is the problem to be solved?
 - Is the problem statement clear?
 - Is the problem manageable, given the time, skills, and tools available?
- Try breaking it into manageable parts
 - Do we know of any tools, libraries, etc. that might help?
 - Yes, even this early: **iostreams**, **vector**, etc.
- Build a small, limited version solving a key part of the problem
 - To bring out problems in our understanding, ideas, or tools
 - Possibly change the details of the problem statement to make it manageable
- If that doesn't work
 - Throw away the first version and make another limited version
 - Keep doing that until we find a version that we're happy with
- Build a full scale solution
 - Ideally by using part of your initial version

Programming is also a practical skill

- We learn by example
 - Not by just seeing explanations of principles
 - Not just by understanding programming language rules

- The more and the more varied examples the better
 - You won't get it right the first time
 - "You can't learn to ride a bike from a correspondence course"

Writing a program: Example

- I'll build a program in stages, making lot of "typical mistakes" along the way
 - Even experienced programmers make mistakes
 - Lots of mistakes; it's a necessary part of learning
 - Designing a good program is genuinely difficult
 - It's often faster to let the compiler detect gross mistakes than to try to get every detail right the first time
 - Concentrate on the important design choices
 - Building a simple, incomplete version allows us to experiment and get feedback
 - Good programs are "grown"

A simple calculator

- Given expressions as input from the keyboard, evaluate them and write out the resulting value
 - For example
 - Expression: $2+2$
 - Result: 4
 - Expression: $2+2*3$
 - Result: 8
 - Expression: $2+3-25/5$
 - Result: 0
- Let's refine this a bit more ...

Pseudo Code

- A first idea:

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

- How do we represent **45+5/7** as data?
- How do we find **45 + 5 /** and **7** in an input string?
- How do we make sure that **45+5/7** means **45+(5/7)** rather than **(45+5)/7**?
- Should we allow floating-point numbers (sure!)
- Can we have variables? **v=7; m=9; v*m** (later)

A simple calculator

- Wait!
 - We are just about to reinvent the wheel!
 - Read Chapter 6 for more examples of dead-end approaches
- What would the experts do?
 - Computers have been evaluating expressions for 50+ years
 - There *has* to be a solution!
 - What *did* the experts do?
 - Reading is good for you
 - Asking more experienced friends/colleagues can be far more effective, pleasant, and time-effective than slogging along on your own
 - “Don’t re-invent the wheel”

Expression Grammar

- This is what the experts usually do – write a *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++

A program is built out of Tokens (*e.g.*, numbers and operators).

A side trip: Grammars

- What's a *grammar*?
 - A set of (syntax) rules for expressions.
 - The rules say how to analyze (“parse”) an expression.
 - Some rules seem hard-wired into our brains
 - Example, you know what this means:
 - $2*3+4/2$
 - **birds fly but fish swim**
 - You know that this is wrong:
 - $2 * + 3 4/2$
 - **fly birds fish but swim**
 - How can we teach what we know to a computer?
 - Why is it right/wrong?
 - How do we know?

Grammars – “English”

Parsing a simple English sentence

Sentence :

Noun Verb
 Sentence Conjunction Sentence

Conjunction :

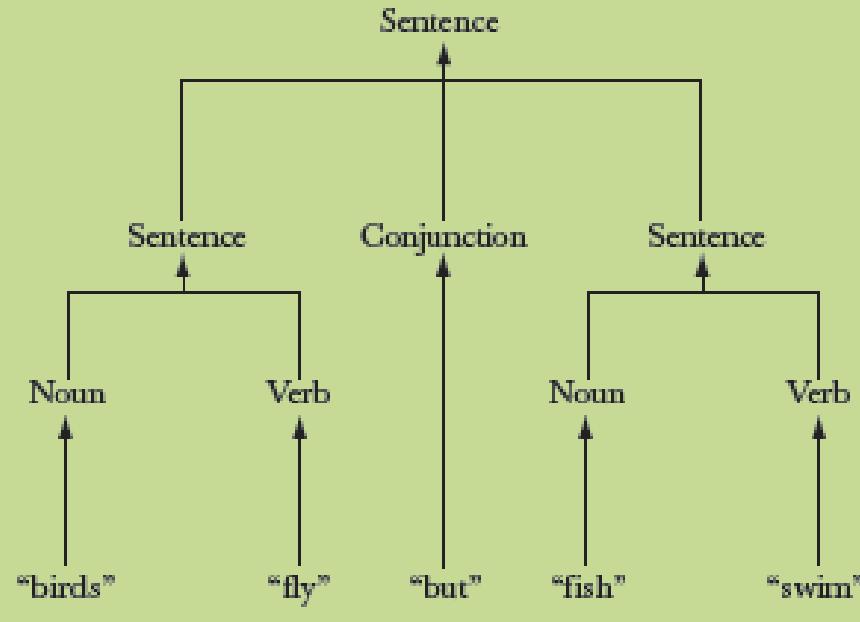
“and”
 “or”
 “but”

Noun :

“birds”
 “fish”
 “C++”

Verb :

“rules”
 “fly”
 “swim”



Grammars - expression

Parsing the number 2

Expression:

- Term
- Expression "+" Term
- Expression "-" Term

Term:

- Primary
- Term "*" Primary
- Term "/" Primary
- Term "%" Primary

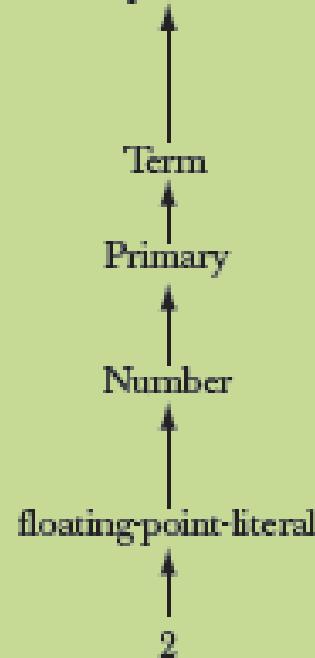
Primary:

- Number
- "(" Expression ")"

Number:

- floating-point-literal

Expression



Grammars - expression

Parsing the expression $2 + 3$

Expression:

Term

Expression "+" Term

Expression "-" Term

Term:

Primary

Term "*" Primary

Term "/" Primary

Term "%" Primary

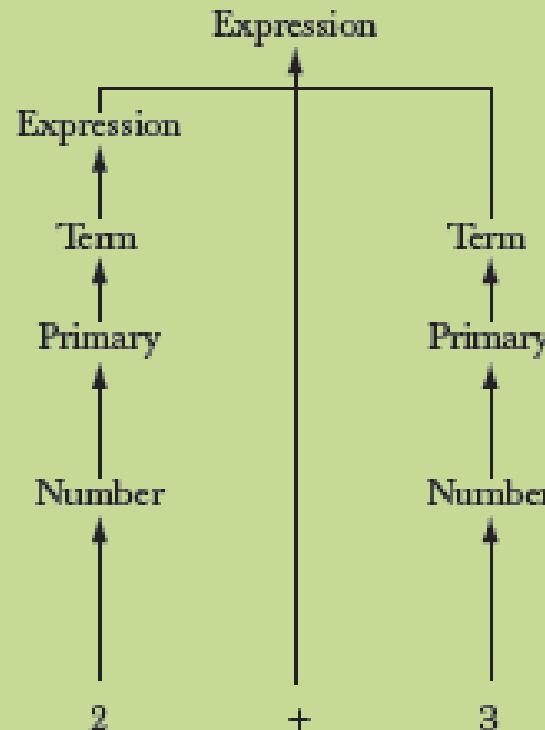
Primary:

Number

"(" Expression ")"

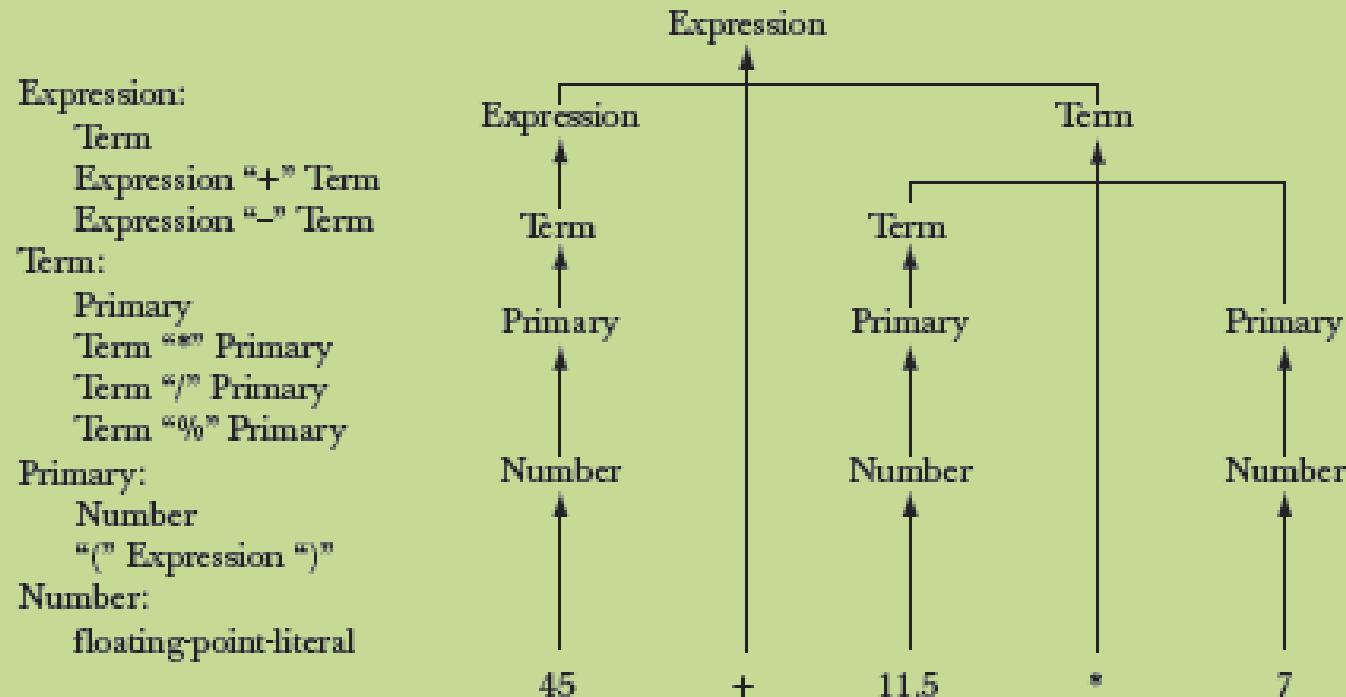
Number:

floating-point-literal



Grammars - expression

Parsing the expression $45 + 11.5 * 7$



Functions for parsing

We need functions to match the grammar rules

```
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()
```

Note: each function deals with a specific part of an expression and leaves everything else to other functions – this radically simplifies each function.

Analogy: a group of people can deal with a complex problem by each person handling only problems in his/her own specialty, leaving the rest for colleagues.

Function Return Types

- What should the parser functions return?
 - How about the result?

```
Token get_token(); // 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
```

- What is a Token?

The program – main()

```
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;
}
```

number
4.5

What is a token?



- We want to see input as a stream of tokens
 - We read characters **1 + 4*(4.5-6)** (That's 13 characters incl. 2 spaces)
 - 9 tokens in that expression: **1 + 4 * (4.5 - 6)**
 - 6 kinds of tokens in that expression: **number + * (-)**
- We want each token to have two parts
 - A “kind”; e.g., number
 - A value; e.g., 4
- We need a type to represent this “Token” idea
 - We'll build that in the next lecture, but for now:
 - **get_token()** gives us the next token from input
 - **t.kind** gives us the kind of the token
 - **t.value** gives us the value of the token

Dealing with + and -

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) {  
            case '+':   left += term(); break;  
            case '-':   left -= term(); break;  
            default:    return left;      // return the value of the expression  
        }  
    }  
}
```

Dealing with *, /, and %

```
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: doesn't compile
 - % isn't defined for floating-point numbers

Dealing with * and /

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
        }
    }
}

```

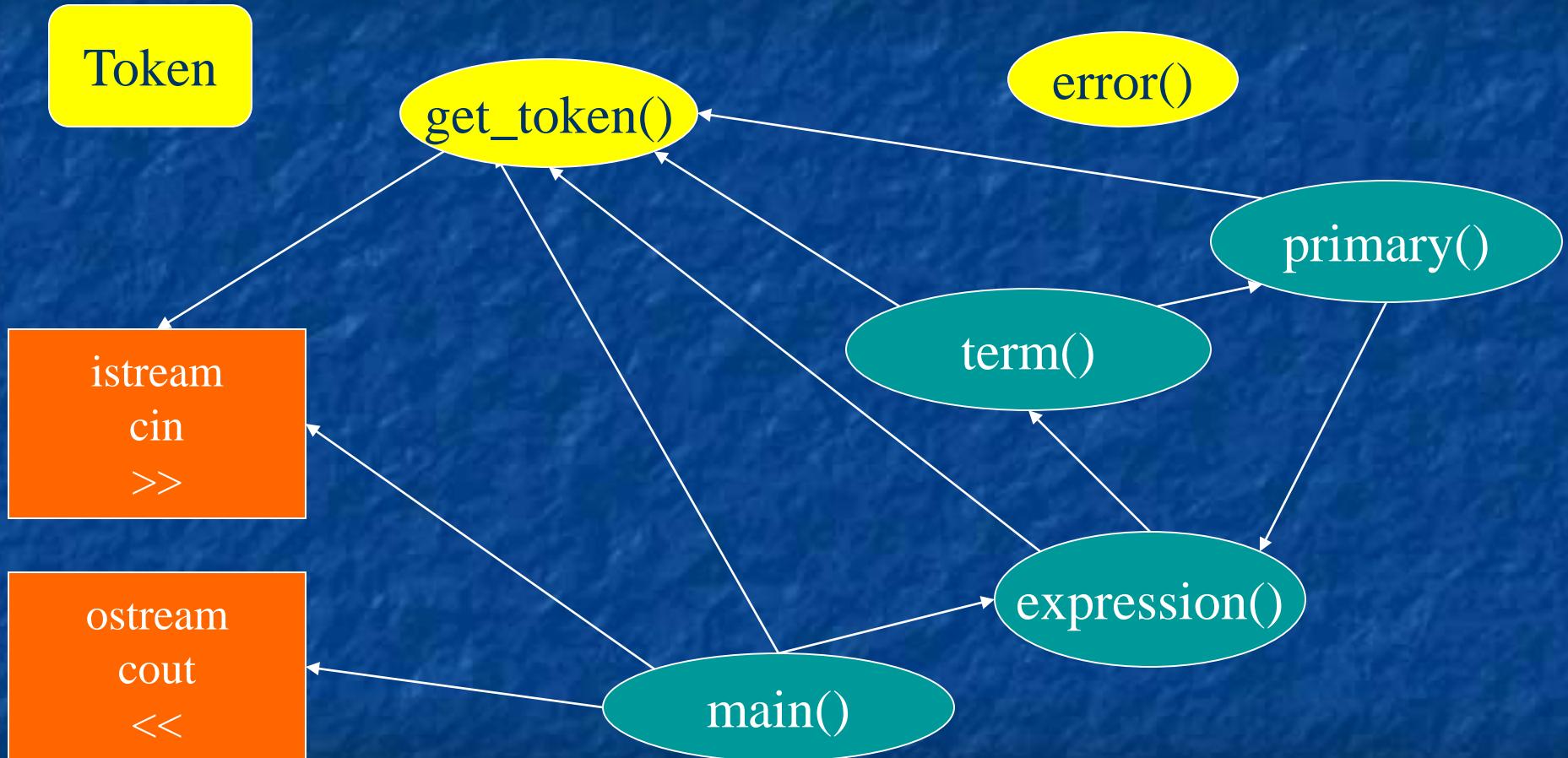
Dealing with divide by 0

```
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
        }
    }
}
```

Dealing with numbers and parentheses

```
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");
    }
}
```

Program organization



- Who calls whom? (note the loop)

The program

```
#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
```

The program – main()

```
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;
}
```

A mystery

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

A mystery

- Expect “mysteries”
- Your first try rarely works as expected
 - That’s normal and to be expected
 - Even for experienced programmers
 - If it looks as if it works be suspicious
 - And test a bit more
 - Now comes the debugging
 - Finding out why the program misbehaves
 - And don’t expect your second try to work either

A mystery

- 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! Our program “eats” two out of three inputs
 - How come?
 - Let’s have a look at expression()

Dealing with + and -

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"*

}

}

Dealing with + and -

- So, we need a way to “put back” a token!
 - Put back into what?
 - “the input,” of course: we need an input stream of tokens, a “token stream”

```
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;
        }
    }
}
```

Dealing with * and /

- Now make the same change to term()

```
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;
        }
    }
}
```

The program

- It “sort of works”
 - That’s not bad for a first try
 - Well, second try
 - Well, really, the fourth try; see the book
 - But “sort of works” is not good enough
 - When the program “sort of works” is when the work (and fun) really start
- Now we can get feedback!

Another mystery

- 2 3 4 2+3 2*3
 - 2 an answer
 - 3 an answer
 - 4 an answer
 - 5 an answer
- What! No “6” ?
 - The program looks ahead one token
 - It’s waiting for the user
 - So, we introduce a “print result” command
 - While we’re at it, we also introduce a “quit” command

The main() program

```
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 == ';')
            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 ...
```

Now the calculator is minimally useful

- 2;
 - 2 an answer
 - 2+3;
 - 5 an answer
 - $3+4*5$;
 - 23 an answer
 - q

Next lecture

- Completing a program
 - Tokens
 - Recovering from errors
 - Cleaning up the code
 - Code review
 - Testing

问题求解与实践 ——实现一个计算器