Writing a Program

Lecture 6

Hartmut Kaiser

https://teaching.hkaiserorg/spring2024/csc3380/

STEM Careers at the NSA and Quantum Computing

- **Event Details:**

- Date: 02/08/2024 (today)
- Time: 2:00 pm CST
- Location: CCT-Digital Media Center (Theater)
- Speaker: Sean Nemetz-MA

- This talk is specially designed for students, professionals, and anyone interested in the cutting-edge developments in STEM. We will discuss opportunities for a STEM career at the agency. This will be followed by a more technical talk about quantum computing, its immediate application in public key cryptography, and the potential impact of quantum computing on the NSA's mission.

Software Development Notes

The Design Process

- Software System Architect
  - Postulates a solution
  - Models it in a design framework
  - Establishes and maintains the vision for the solution
  - Evaluates design against original requirements
- Primary responsibility of the Software System Architect
  - Specify a solution to a given problem (usually expressed as a functional specification)
  - Implementation independent

The Design Process

- Software Designer
  - Designs the internal working of system components
    - Defines subsystems
    - Crafts process logic
    - Details data flow between and within system components and external sources and interfaces
  - Produces a specification of the design, detailed enough that
    - A programmer can implement it
    - A tester can test it
    - A technical writer can document it

Objectives of the Design Process

- Primary responsibility of the Software Designer
  - Produce a set of specifications that describe the intended form of the implementation for the software system
- The design specifications
  - Describe
    - The form (structure) of the solution
    - The way that the components are to fit together
  - Act as a set of "blueprints" that show how the system is to be constructed

Desirable Features…

- Fitness for purpose
  - The system must work, and work correctly
  - It should
    - perform the required tasks
    - in the specified manner and
    - within the specified constraints
    - of the specified resources
- Robustness
  - The design should be stable against changes such as file and data structures, user interface, etc.

Desirable Design Features

- Simplicity
  - The design should be as simple as possible, but no simpler
- Separation of concerns
  - The different concepts and components should be separated out (modular)
- Information hiding
  - Information about the detailed form of objects such as data structures and device interfaces should
    - be kept local to a module or unit
    - Not be directly "visible" outside that unit

Undesirable Features

- Having too much retained state information spread around the system
- Using interfaces that are too complex
- Containing excessively complex control structures
- Involving needless replication

Design Strategies

- Top-down
  - Functional decomposition
  - Stepwise refinement at the component level
- Bottom up
  - Composition
  - Design pieces in isolation before deciding how they will fit together as a whole
- Stylized
  - Pattern (re)use
  - Good solution already exists, in part or in whole
- There is a place for all of these strategies in software and software system design
  - Start with top-down architecture design
  - Components are handed off to development team for bottom-up software design
  - Patterns are reused at both the architecture and software design levels, where appropriate

Building a Calculator

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: **`iostream`**s, **`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 our initial version

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) {                 // what's a line?
        analyze the expression           // what does that mean?
        evaluate the expression
        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

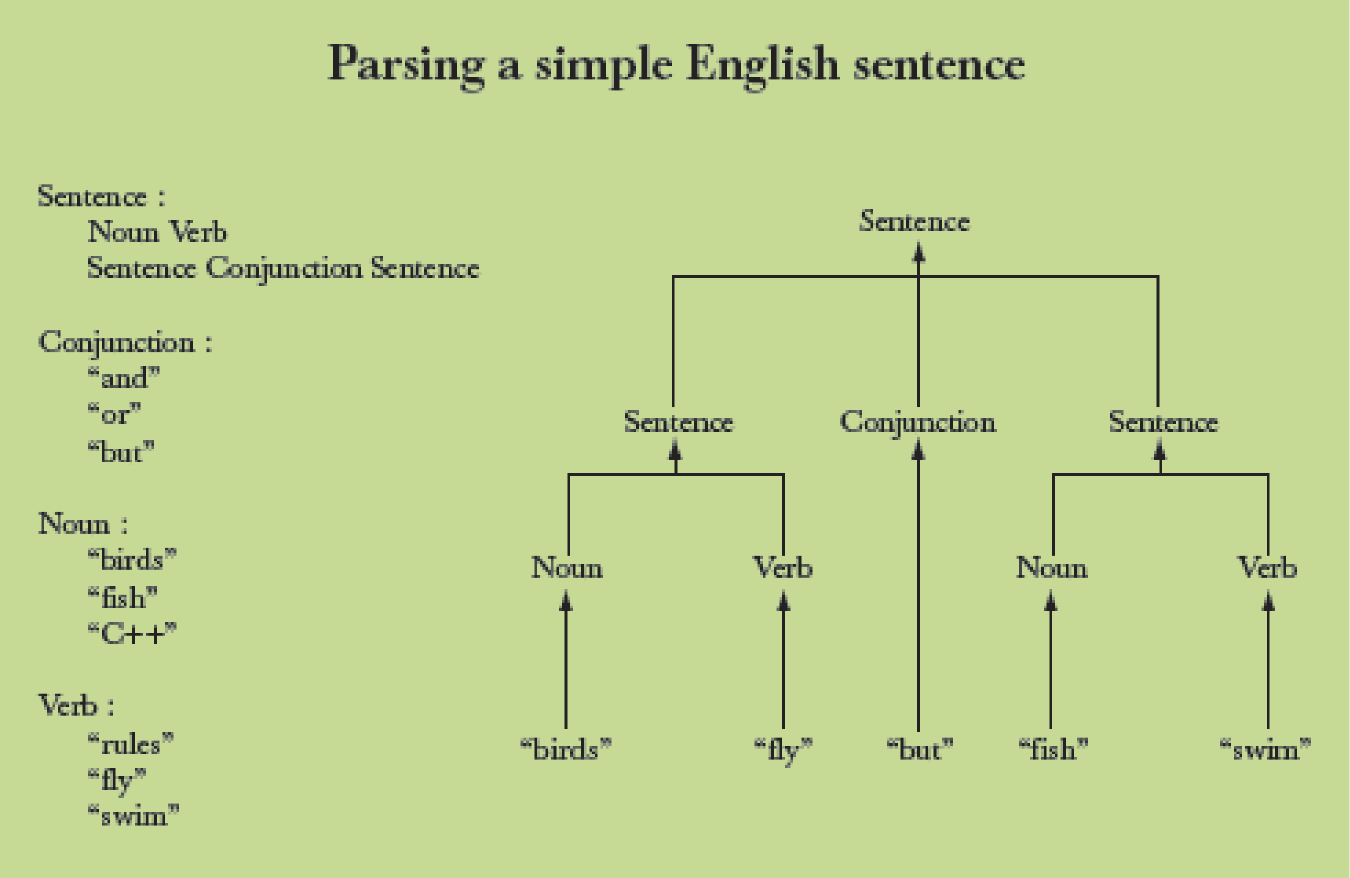
- Wait!
  - We are just about to reinvent the wheel!
  - Read Chapter 6 for more examples of dead-end approaches[1]
- What would the experts do?
  - Computers have been evaluating expressions for 50+ years
  - There *has* to be a solution!
  - What *did* the experts do?
    - Reading is good for you
    - Asking more experienced friends/colleagues can be far more effective, pleasant, and time-effective than slogging along on your own

[1] Bjarne Stroustrup, Programming – Principles and Practice using C++

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 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"
  - Why is it right/wrong?
  - How do we know?
  - How can we teach what we know to a computer?

Grammars – “English”



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++
```

- An expression is built out of Tokens (*e.g.*, numbers and operators).

Grammars - Expression

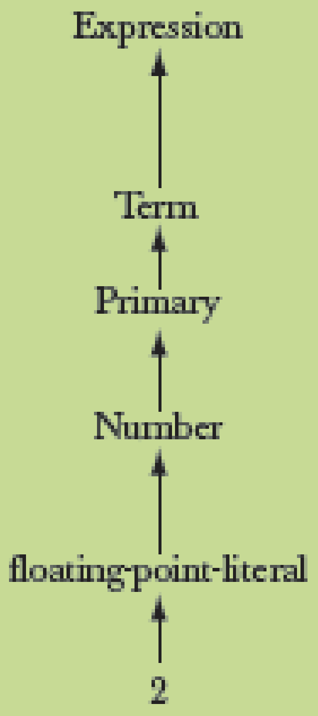
Parsing the number 2

Expression:
- Term
- Expression "+" Term
- Expression "–" Term

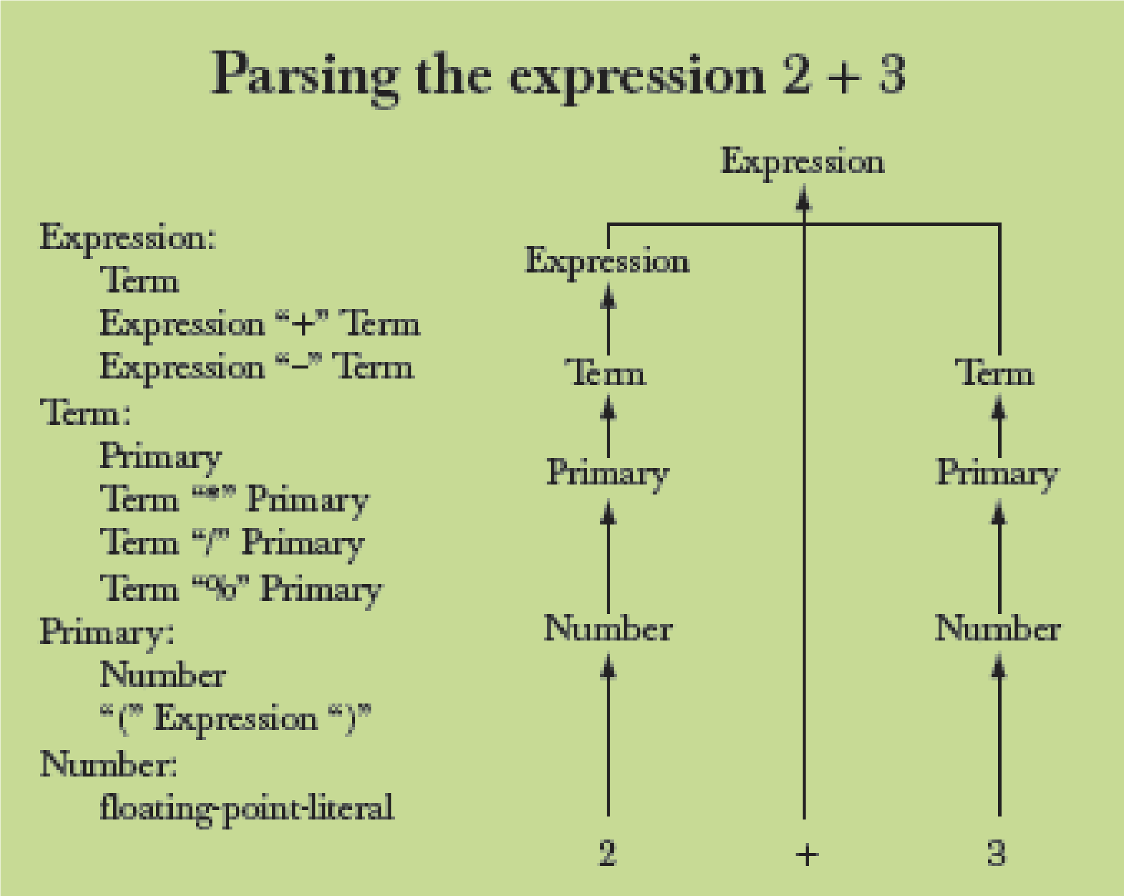
Term:
- Primary
- Term "*" Primary
- Term "/" Primary
- Term "%" Primary

Primary:
- Number
- "(" Expression ")"

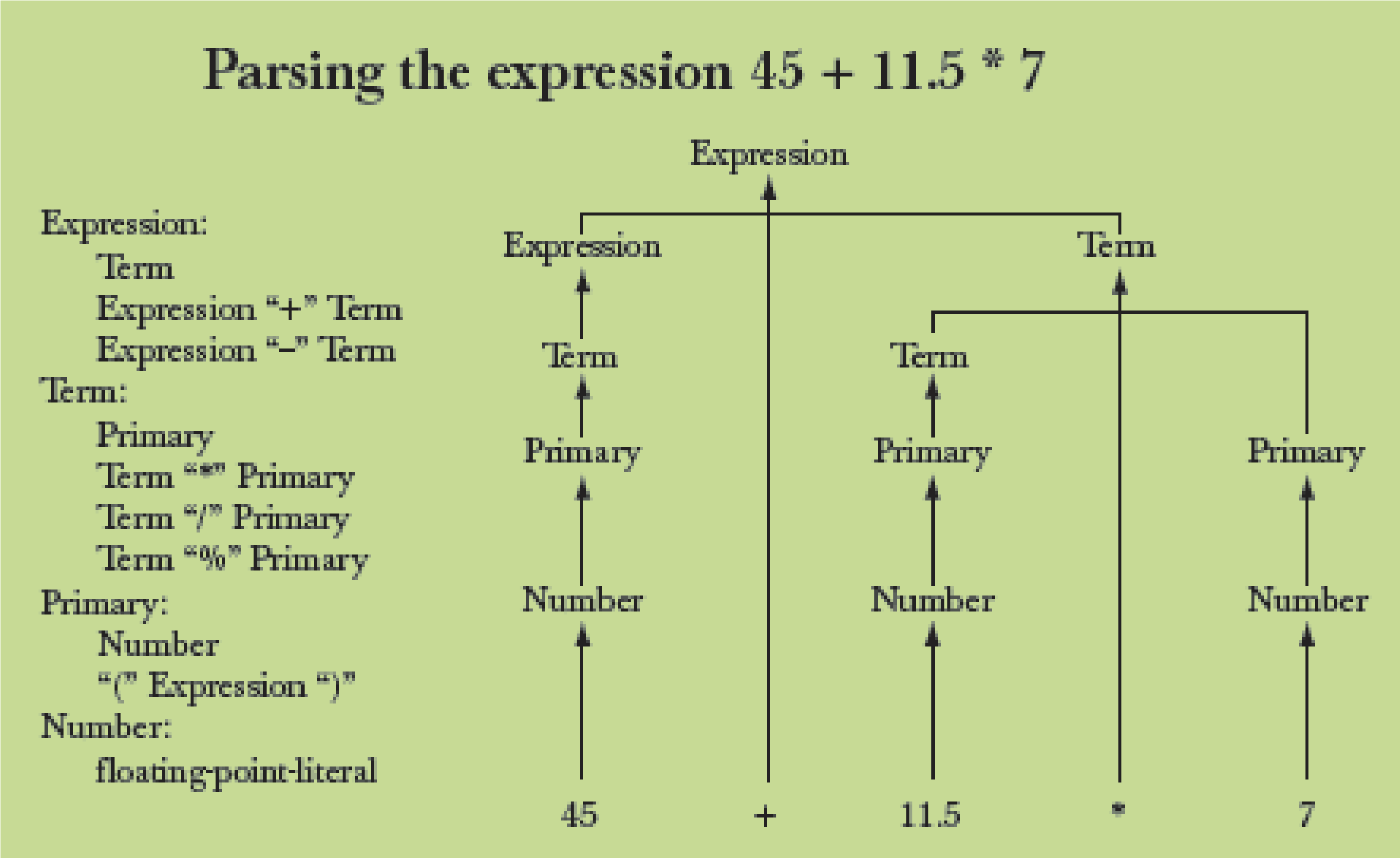
Number:
- floating-point-literal



Grammars - Expression



Grammars - Expression



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

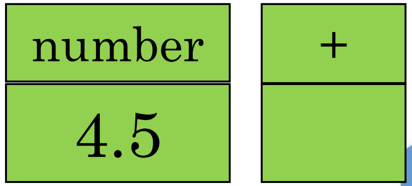
- What should the parser functions return?
  - How about the result?

```
token get();                // 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’?

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 later, but for now:
    - `t = get()` 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          e.g., 1+2,   (1-2)+3,   2*3+1
  Expression '-' Term
```

```
// read and evaluate: 1, 1+2.5, 1+2+3.14 etc., return the sum (or difference)
double expression()
{
    double left = term();          // get the Term
    while (true)
    {
        token t = get();           // 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
        }
    }
}
```

Dealing with *, /, and %

```
Term :
  Primary
  Term '*' Primary                    e.g., 1*2,   (1-2)*3.5
  Term '/' Primary
  Term '%' Primary
```

```
// exactly like expression(), but for *, /, and  %
double term()
{
    double left = primary();    // get the Primary
    while (true) {
        token t = get();         // 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                    e.g., 1*2,   (1-2)*3.5
  Term '/' Primary
```

```
// exactly like expression(), but for * and /
double term()
{
    double left = primary();    // get the Primary
    while (true) {
        token t = get();        // 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

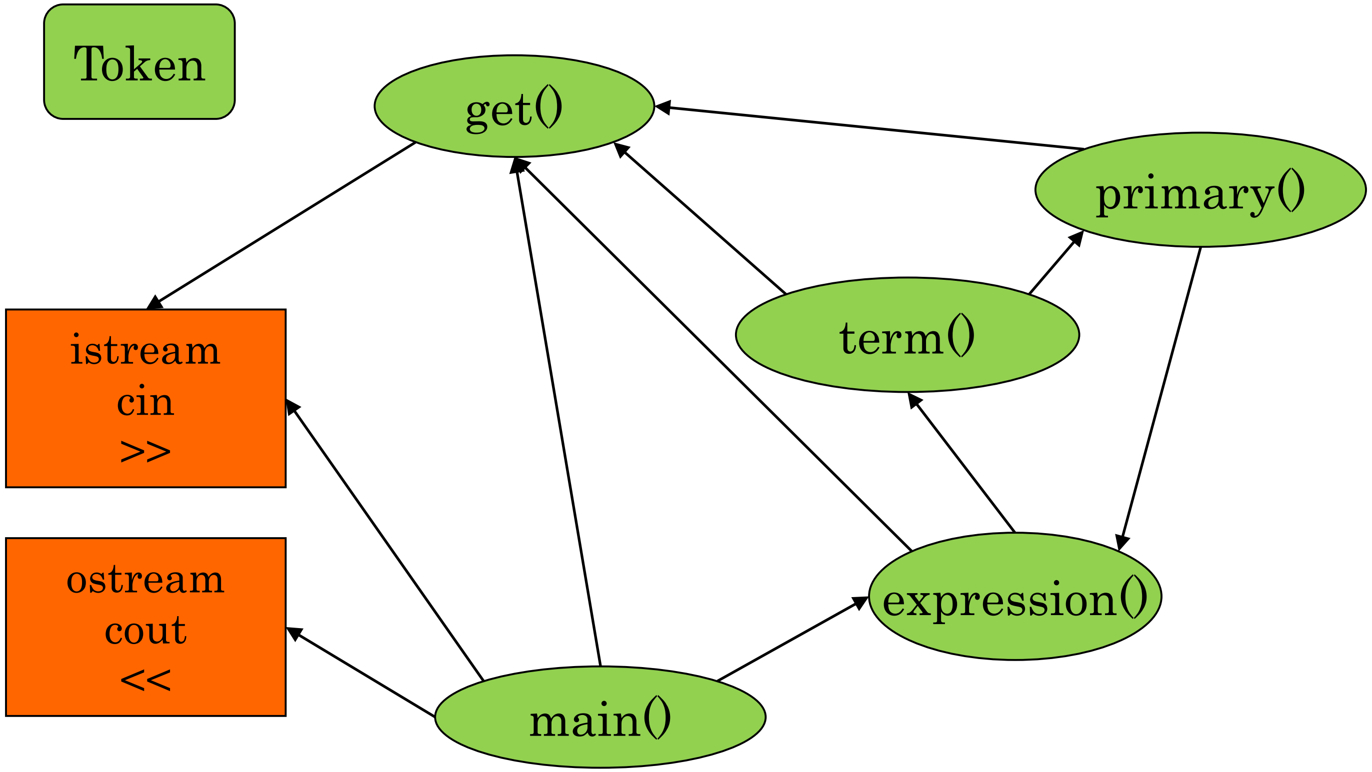
```
// exactly like expression(), but for * and /
double term()
{
    double left = primary();    // get the Primary
    while (true) {
        token t = get();        // get the next Token ...
        switch (t.kind) {
        case '*': left *= primary(); break;
        case '/': {
            double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

Dealing with numbers, ‘(‘and ‘)’

```
double primary()    // Number or ‘(‘ Expression ‘)’
{
    token t = get();
    switch (t.kind) {
    case '(': {                 // handle ‘(’ expression ‘)’
        double d = expression();
        t = get();
        if (t.kind != ')') throw std::runtime_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:
        throw std::runtime_error("primary expected");
    }
}
```

Program Organization

- Who calls who? (note the loop)



The program

```
#include <iostream>
#include <string>

// Token stuff (explained in next lecture)

// declaration so that primary() can call expression()
double 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 (std::cin)
            std::cout << expression() << '\n';
        return 0;
    }
    catch (std::runtime_error& e) {
        std::cerr << e.what() << std::endl;
        return 1;
    }
    catch (...) {
        std::cerr << "exception\n";
        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

- 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          e.g., 1+2,   (1-2)+3,   2*3+1
  Expression '-' Term
```

```
// read and evaluate: 1, 1+2.5, 1+2+3.14 etc., return the sum (or difference)
double expression()
{
    double left = term();          // get the Term
    while (true)
    {
        token t = get();           // 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", discards it
        }
    }
}
```

Dealing with + and -

- So, we need a way to "put back" a token!
  - Back into what?
  - "the input," of course; that is, we need an input stream of tokens

```
// read and evaluate: 1, 1+2.5, 1+2+3.14 etc., return the sum (or difference)
double expression()
{
    double left = term();          // get the Term
    while (true)
    {
        token t = ts.get();        // get the next token ...
        switch (t.kind)  {         // ... and do the right thing with it
        case '+': left += term(); break;
        case '-': left -= term(); break;
        default: ts.putback(t);    // <<< put the unused token back into the token stream
                 return left;
        }
    }
}
```

Dealing with * and /

- Now make the same change to **term()**

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

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

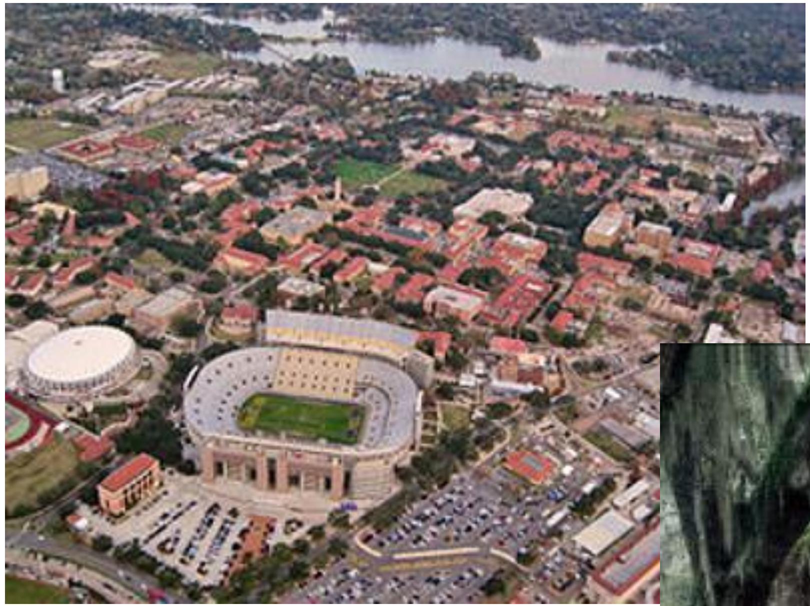
```
int main()
{
    double val = 0;
    while (std::cin)
    {
        token t = ts.get();
        if (t.kind == 'q')
            break;                          // ‘q’ for “quit”
        if (t.kind == ';')                  // ‘;’ for “print now”
            std::cout << val << '\n';       // print result
        else
            ts.putback(t);
        val = expression();                 // evaluate
    }
    return 0;
}
// exception handling ...
```

Now the calculator is minimally useful

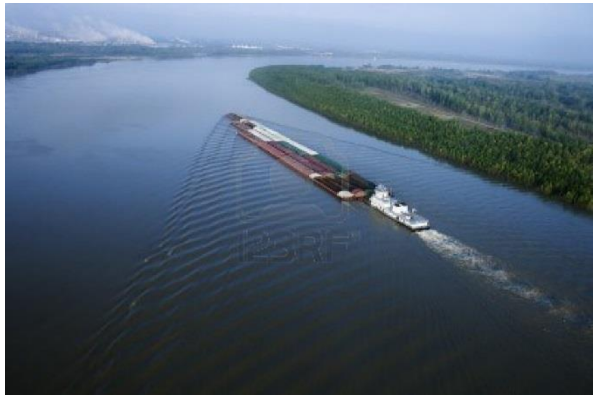
- 2;
- 2 an answer
- 2+3;
- 5 an answer
- 3+4*5;
- 23 an answer
- q

Next Lecture

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



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP