CS241 Principles and Practice of Problem Solving

Lecture 5: Writing a program

Yuye Ling, Ph.D.

Shanghai Jiao Tong University
John Hopcroft Center for Computer Science

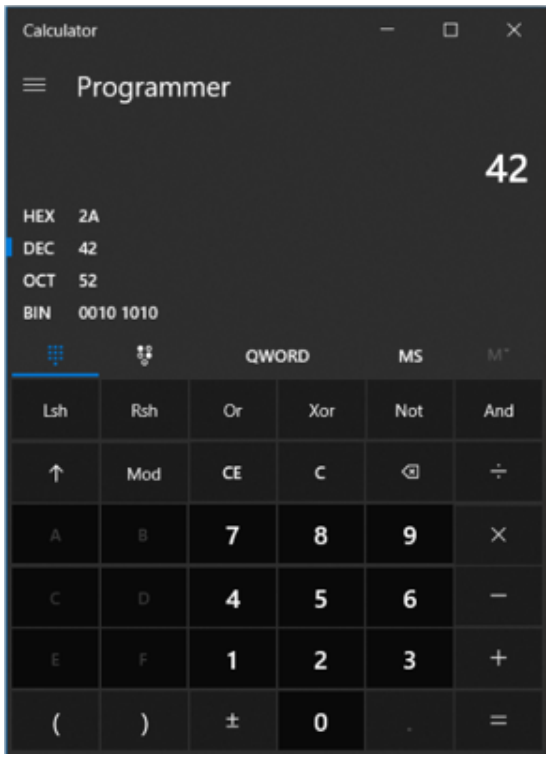
September 23, 2019

Copyright Notice

A large portion of the contents in the following pages come from Prof. Bjarne Stroustrup's slides.
The original contents could be found here.

Spoiler

This lecture and the next will describe the process of designing a program through the example of a "simple" desk calculator.

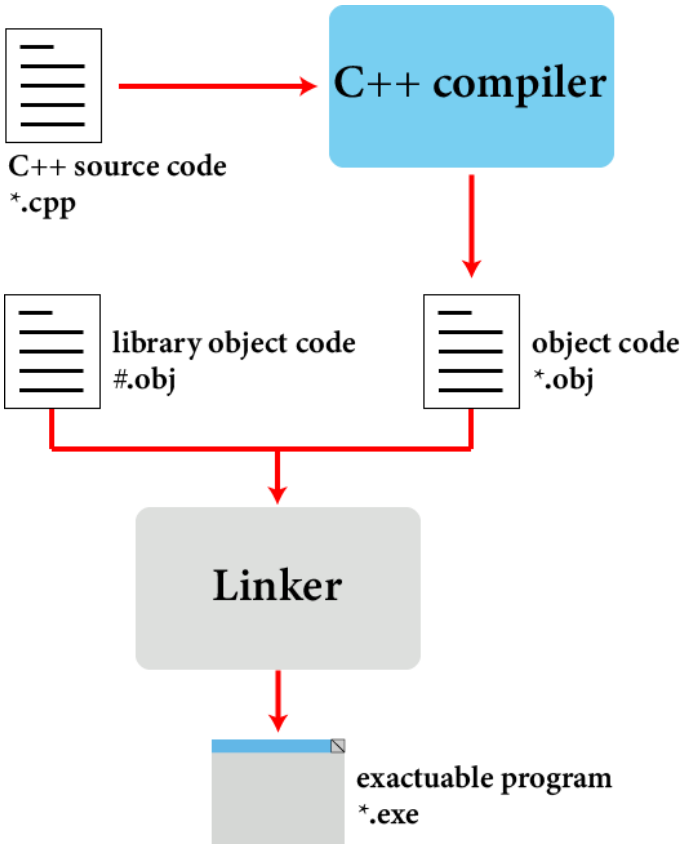


What do we want from the desk calculator?

- We should have a User Interface
- The calculator should be able to do simple arithmetics

How do you feel? Is this a simple task?

Spoiler (Cont'd)



We have learned a little bit about compiler

What is a compiler?

What about asking you to write a compiler?

Some thoughts on software development

Write a calculator: just do it

- Define the problem
- Divide the problem
- First attempt

Detour: compiler's analogy

- The similarity between the calculator and the compiler
- How does the compiler works
- Implement the calculator by the heuristics from the compiler

Some thoughts on software development

Write a calculator: just do it

- Define the problem
- Divide the problem
- First attempt

Detour: compiler's analogy

- The similarity between the calculator and the compiler
- How does the compiler works
- Implement the calculator by the heuristics from the compiler

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 still

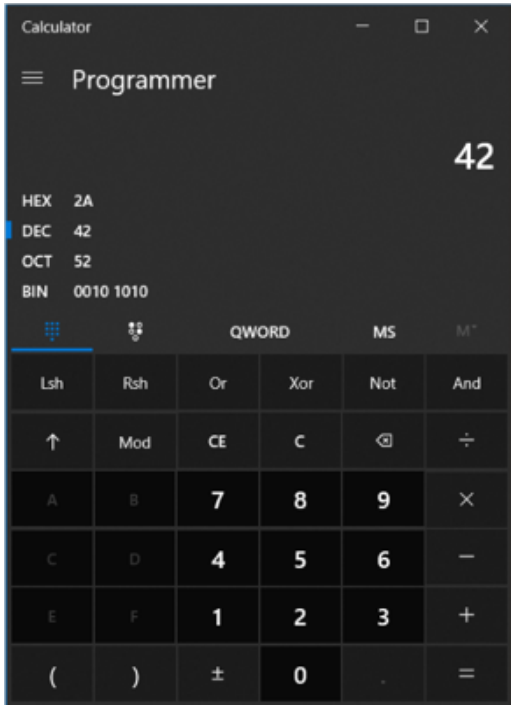
- We learn by example
  - Not by just seeing explanations of principles
  - Not just by understanding programming language rules
- The more and the more varied examples the better
  - You won't get it right the first time
  - "You can't learn to ride a bike from a correspondence course"

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"

What's the problem to be solved

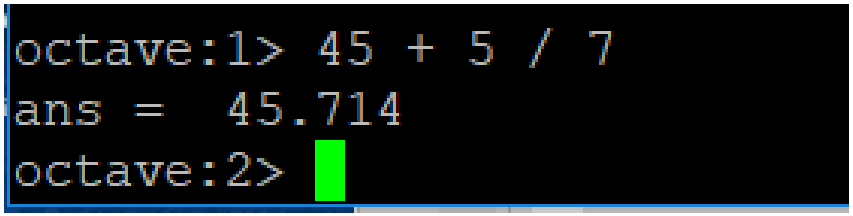
- Ideally, we are expecting this



- Since we don't know how to write GUI
  something like this is also acceptable

```
octave:1> 45 + 5 / 7
ans =  45.714
octave:2>
```

Pseudo code



A first idea

```
int main()
{
    variables
    while (get a line) {
        analyze the expression
        evaluate the expression
        print the result
    }
}
```

Anyway, let's implement the addition and subtraction

```
1  #include <isosteam>
2  using namespace std;
3
4  int main(){
5      cout << "Please enter expression (we can handle + and -):";
6      int lval = 0;
7      int rval;
8      char op;
9      int res;
10     cin >> lval >> op >> rval;
11
12     if (op == '+')
13         res = lval + rval;
14     else if (op == '-')
15         res = lval - rval;
16
17     cout << "Result:" << res << '\n';
18     return 0;
19 }
20
```

Demo

It worked! But is it useful?

But we do want a more sophisticated calculator now

1. Add multiplication and division
2. Add the ability to handle more than one operand

Advanced version

```
#include <isostream>
using namespace std;

int main(){
    cout << "Please enter expression (we can handle +, -, * and /):";
    cout << "Add an x to end expression (e.g., 1+2*3x)";
    int lval = 0;
    int rval;
    cin >> lval;
    if (!cin) error("no first operand");
    for (char op; cin >> op; ){
        if (op != 'x') cin >> rval;
        if (!cin) error("no second operand");
        switch (op){
        case '+':
            lval += rval;
            break;
        case '-':
            lval -= rval;
            break;
        // to be continued...

```

Advanced version (Cont'd)

```
1            case '*':
2                lval *= rval;
3                break;
4            case '/':
5                lval /= rval;
6                break;
7            default:
8                cout << "Result:" << lval << '\n';
9            return 0;
10           }
11       }
12       error("bad expression");
13   }
14
```

This version works to some extend.

What about 45 + 5 / 7?

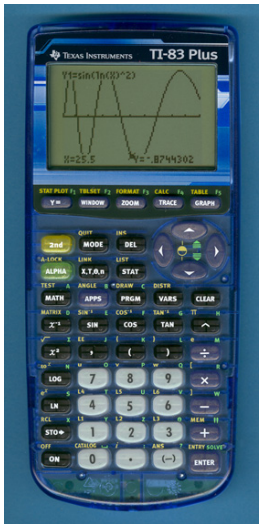
It couldn't handle the concept of "priority"

Quick summary

Taking 45 + 5 / 7 as an example. The previous approach is process oriented.

1. How do we represent 45+5/7 as data?
2. How do we identify 45, +, 5, /, and 7 in an input stream?
3. How do we make sure that 45+5/7 means 45+(5/7) rather than (45+5)/7?

Writing a calculator is not trivial!



The essence of the "simple calculator"

The calculator we want is indeed a "compiler" (or "translator")

- We input math expressions in natural language
- The proposed program has to translate the expressions into C++ language

Before we can implement the calculator, we have to learn some ABCs about the compiler/translator.

Compiler as a translator

What is compiler?
What could be the design goals for compilation/translation?

- Compilation time
- Performance of the generated codes
- Correctness

Direct translation?

Different language have different organizations

- "subject + verb + object" versus "subject + object + verb "
- One vocabulary might have multiple meanings
  - You have to put the word into context

We need a grammar to help us!

Divide and conquer I: Lexical analysis

Assume we want to translate/compile

```
position = initial + rate * 60
```

Let's analyze the problem and divide it into smaller steps.

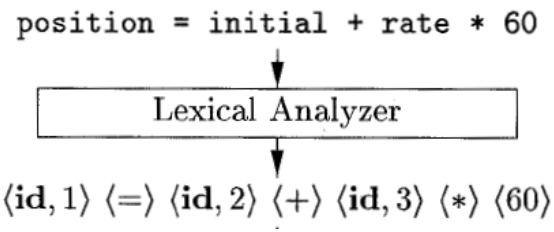
Think about ourselves. How do we perform the translation?

- start with the alphabets
- putting alphabets into words
  - Recognizing (not understanding) words is not trivial
  - Try this
  - w hat isthis se ntence?
  - For emaxlpe, it deson't mttaer in waht oredr the ltteers in a wrod aepapr, the olny iprmoatnt tihng is taht the frist and lsat ltteer are in the rghit pcale. The rset can be a toatl mses and you can sitll raed it wouthit pobelrm.

Lexical analyzer

Recall that in the first lecture, computer is not as smart as you.

1. They must know what the word separators are: The language must define rules for breaking a sentence into a sequence of words
2. The lexical analyzer breaks a sentence into a sequence of words or tokens:
   - `position = initial + rate * 60`
   - Sequence of words (total 7 words)

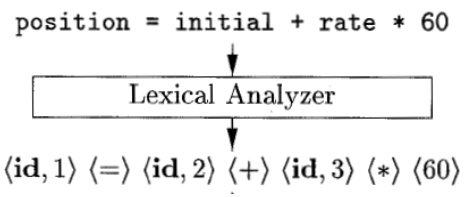


What is token?

Token is an elementary symbols of the language defined by the grammar.

Example

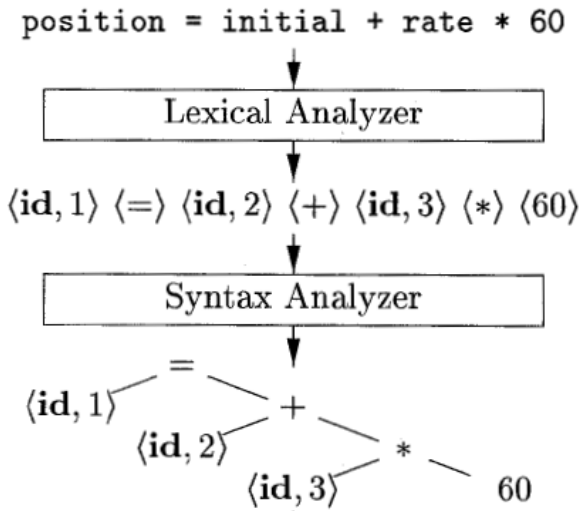
- 1 + 4*(4.5-6)
- 9 tokens: 1, +, 4, *, (, 4.5, -, 6,)
- 6 types: number, +, *, (, -,)
- To design the token, we want it to have two parts
  - One part to describe its attribute
  - The other stores its value



Divide and conquer II: Syntax analysis

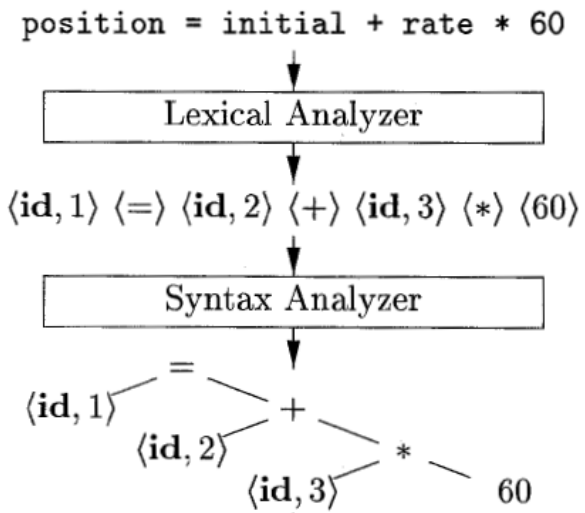
Once the words are understood, the next step is to understand the structure of the sentence.

This process is called syntax analysis: the tokens produced by the lexical analyzer are used create a tree-like intermediate representation that depicts the grammatical structure of the token stream.



Parsing: just a fancier name for syntax analysis

The operation of parsing is essentially traversing the parse-tree



Question: how do we traverse a tree?

We could do this recursively.

Example: arithmetics

Three levels of hierarchies:

- Expression: for "+" and "-"
- Term: for "*", "/", and "%"
- Primary: numbers

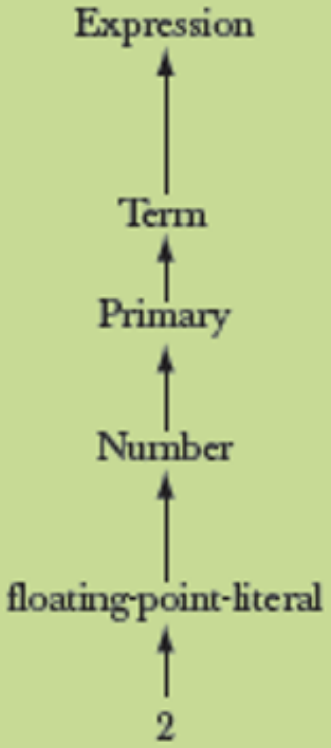
Explicit grammar

```
Expression:
    Term
    Expression "+" Term
    Expression "-" Term
Term:
    Primary
    Term "*" Primary
    Term "/" Primary
    Term "%" Primary
Primary:
    Number
    "("Expression")"
Number:
    Float
```

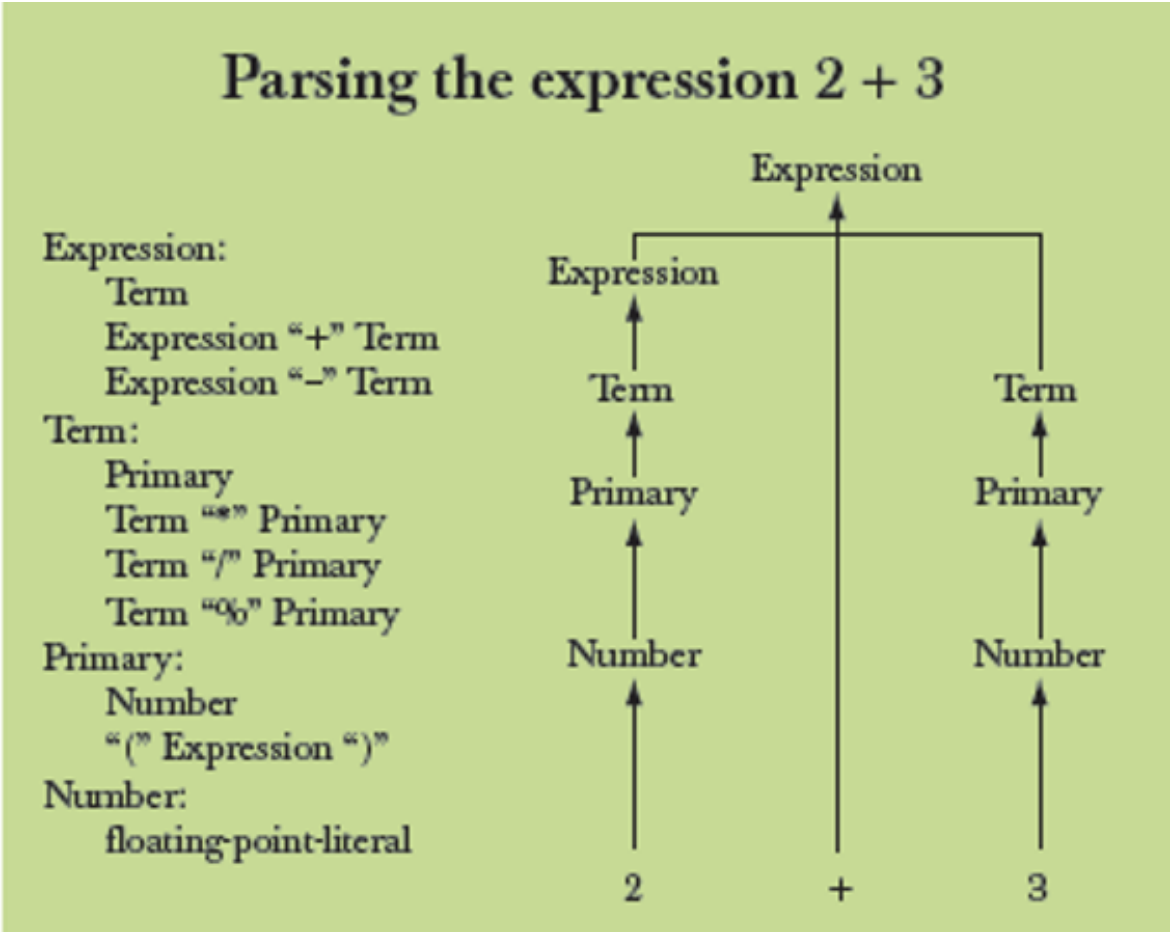
Example: arithmetics (Cont'd)

Parsing the number 2

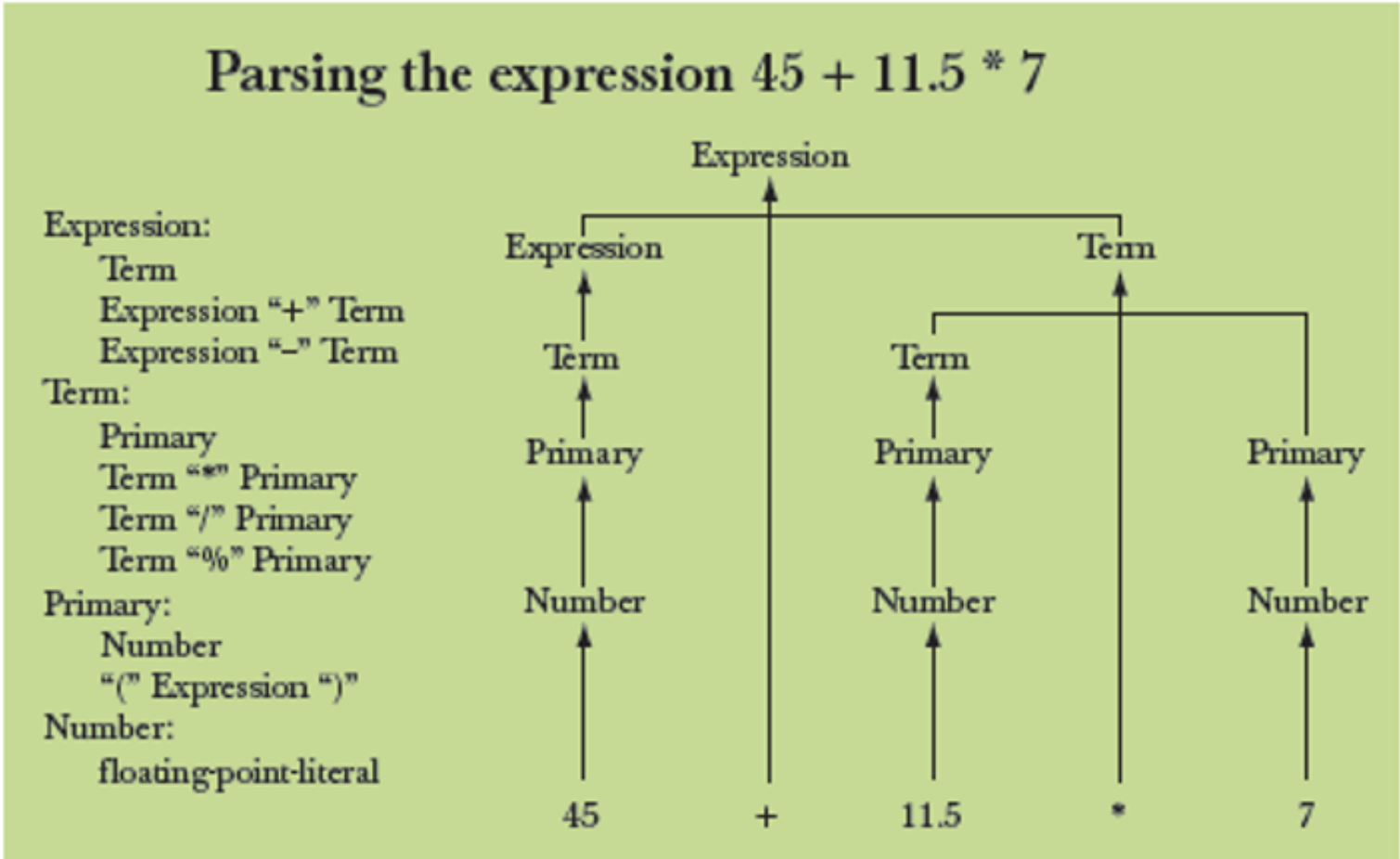
Expression:
    Term
    Expression "+" Term
    Expression "–" Term
Term:
    Primary
    Term "*" Primary
    Term "/" Primary
    Term "%" Primary
Primary:
    Number
    "(" Expression ")"
Number:
    floating-point-literal



Example: arithmetics (Cont'd)



Example: arithmetics (Cont'd)



Example: arithmetics (Cont'd)

What about 45 + 11.5 * 7 + 6.2?

Divide and conquer III: Semantic analysis

Once the structure is identified we try to understand the meaning of the sentence.

Or, we try to let the computer understand the meaning of the sentence.

By how?

Implement it in codes.

A revisit to the "simple" calculator

Now we can come back to our original task: write a calculator.

Potential steps

- Read in expressions: try to separate words, and put them into Tokens (lexical analysis)
- Organize the tokens: into a parse-tree (syntax analysis)
- Execute the operations: implement the specific arithmetic operations (semantic analysis)

Parsing is the key

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()

Recursive

Dealing with + and -

```
Expression:
    Term
    Expression '+' Term
    Expression '-' Term
```

Implementation

```
1  double expression() // read and evaluate: 1   1+2.5   1+2+3.14   etc.
2  {
3      double left = term();  // get the Term
4      while (true) {
5          Token t = get_token(); // get the next token
6          switch (t.kind) { //  and do the right thing with it
7              case '+':     left += term(); break;
8              case '-':     left -= term(); break;
9              default:     return left; // return the value of the expression
10         }
11     }
12 }
13
```

Dealing with *, /, and %

```
1  double term() // exactly like expression(), but for *, /, and %
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7              case '*':     left *= primary(); break;
8              case '/':     left /= primary(); break;
9              case '%':   left %= primary(); break;
10             default:     return left; // return the value
11         }
12     }
13 }
14
```

Dealing with * and /

```
Term:
    Primary
    Term '*' Primary
    Term '\' Primary
```

Implementation

```
1  double term() // exactly like expression(), but for *, and /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7              case '*':     left *= primary(); break;
8              case '/':     left /= primary(); break;
9              default:     return left; // return the value
10         }
11     }
12 }
13
```

Dealing with divide by 0

```
1  double term() // exactly like expression(), but for * and  /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7          case '*':
8              left *= primary();
9              break;
10 case '/':
11             { double d = primary();
12             if  (d==0) error("divide by zero");
13             left /= d;
14             break;
15             }
16         default:
17             return left; // return the value
18         }
19     }
20 }
21
```