Week4 :

Scientific Computing
Writing and completing a program

Blair Jamieson

University of Winnipeg

Class 4

Outline

Writing a Program

Completing a program

Outline

Writing a Program
- A Problem
- A Simple Calculator
- Grammars
- Turning grammar into code
- Program Structure

Completing a program

A problem

Today's class we will work through a programming problem that is worked through in the textbook Chapters 6 and 7. We picked a problem that:

- Illustrates design and programming techniques
- Doesn't require too many new language constructs
- Is complicated enough to require some thought
- Allows many variations in solution
- Allows many variations in its solutions
- Solves an easily understood problem
- Is short enough to present (in 3 hours)

The problem : a simple calculator

- Get the computer to do ordinary arithmetic on expressions the user types in, eg:
- You enter: $2 + 3.1 * 4$
- The program should respond: 14.4
- Thinking about the problem:
  1. First think about how you would like to interact with the program
  2. Often a design will evolve after you start working on it – our example will do the same
  3. The journey through the steps from intital design to final product is as important to learn as the technical language details

Stages of development

- *Analysis*: Describe current understandign of the problem – can be called requirements or specification.
- *Design*: Create an overall structure for the system and how the parts will communicate. Pick any libraries of code that can be used.
- *Implementation*: Write the code, debug it, and test it.

Strategy in programming

1. Ask yourself some strategy questions:
   - What is the problem to be solved?
   - How would I like to interact with this program?
   - Is the problem statement clear? – Often it is better to start by not asking too much of the program, and later add to it in an updated 2.0 verstion.
   - Does the problem seem manageable given time, skills, and tools available?
2. Try breaking the problem into manageable parts.
   - Do you know any tools/libraries that might help? (for sure use C++ standard library!)
   - Look for parts of a solution that can be seperately described (eg. vector, string, iostream) – in this class we will see `Token` and `Token_stream`
   - Analogy to building a car, break into parts: wheels, engine, seats, etc.
3. Build a small, limited version of the program first
   - This will bring out problems in our understanding

A simple calculator

- We will use `cin` and `cout` to interact with the calculator (that's all we've learned about so far)
- Example inputs and outputs:

```
Expression: 2+2
Result: 4
Expression: 2+2*3
Result: 8
Expression: 2+3-25/5
Result: 0
```

- Expressions are input by the user, results are produced by the program
- Pseudo-code for program's operation:

```
read_a_line();
calculate();
write_result();
```

A first code

firstcalculator.cpp

```
#include "std_lib_facilities.h"
int main(){
  cout << "Please enter expression (we can handle +
      and -): ";
  int lval = 0, rval = 0;
  char op = ' ';
  int res = 0;
  cin >> lval >> op >> rval;
  if ( op == '+' ){
    res = lval + rval;
  } else if ( op == '-' ) {
    res = lval - rval;
  } else {
    cerr << "Unknown op " << op << endl;
  }
  cout << "Result: " << res << endl;
  return 0;
}
```

Notes on firstcalculator.cpp

- First code basically works. Lets clean it up and add more features.
- Lets also check inputs as we go (instead of getting them all in one input line).
- We will add multiplication and division, and ability to handle multiple operands.
- Ie. $1 + 2 * 3$

A second version – secondcalculator.cpp

```
1  int main() {
2    cout << "Enter expression with +,-,*,/ (add x to
         end):"<<endl;
3    int lval = 0, rval = 0;
4    cin >> lval;
5    if (!cin) error("No first operand");
6    for ( char op; cin >> op; ){
7      if ( op != 'x' ) cin >> rval;
8      if (!cin) error("No second operand");
9      switch ( op ){
10       case '+':  lval += rval;  break;
11       case '-':  lval -= rval;  break;
12       case '*':  lval *= rval;  break;
13       case '/':  lval /= rval;  break;
14       default:
15       cout << "Result: "<< lval << endl;
16       return 0;
17     }
18   }
19   error ("bad expression");
20 }
```

Notes on secondcalculator.cpp

- This version, we notice some problems.
- For example we type: $1 + 2 * 3$, and the program responds: 9
- If it followed ordinary order of operation, we expected to get 7
- What went wrong?
- Problem is that we can't just evaluate from left to right – need to look for * and / before + and -
- We need to consider:
  1. How to handle user input over several lines
  2. How do we search for *, (, or / among +, - and numbers?
  3. How do we remember where the * was?
  4. How do we handle evaluation that isn't left to right (1+2*3)?

Tokens

- We can look what people have already done in other calculator programs.
- Typically "tokenize" the inputs.
- For example if user enters: $45 + 11.5/7$
- We make a list of tokens: 45, +, 11.5, /, 7
- A token is something we consider a "unit", such as a number or operator
- We will use three kinds of tokens:
  1. floating point literals, eg. 3.1415, 0.27e2, 42
  2. operators: +, -, /, *, %
  3. parentheses: (and)
- Need a way to store "tokens" that have a (kind, value) pair

Token

We define a new type `Token` to represent tokens:

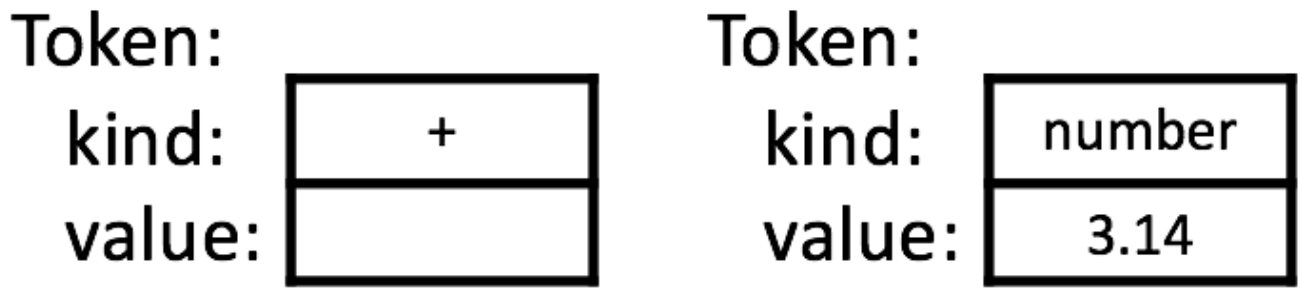


Figure: `Token` a type to represent (kind, value) pairs.

The code for this user defined type is:

```
1 class Token {
2   // a user defined type (kind, value) pair
3   public:
4   char kind;
5   double value;
6 };
```

- The keywork `class` is used to define a new user type
- We use a char to represent the token type

Using Token

We can use the `Token` type like this:

```
Token t;        // t is a Token
t.kind  = '+'; // t represents a '+'
Token t2;       // t2 is a token
t2.kind = '8'; // we use '8' to represent numbers
t2.value = 3.14'
```

- We use the dot "." notation `object_name.member_name` to access a member of a class
- We can copy tokens:

```
Token tt = t;          // copy initialization
if ( tt.kind != t.kind ) error("Impossible!");
t = t2;                // assigment
cout << t.value ;      // prints 3.14
```

Using Tokens

- Using Tokens, we can represent $(1.5 + 4) * 11$ using 7 tokens:

7 Tokens representing (1.5+4)*11

kind:	(	8	+	8	)	*	8
value:		1.5		4			11

Figure: `Token`s representing $(1.5 + 4) * 11$

- We can write a function to get a token (declared as):

```
Token get_token();
```

Using Tokens

- We can collect `Tokens` in a vector:

```
1  int main(){
2    vector< Token > tok;
3    while (cin ){
4      Token t = get_token();
5      tok.push_back( t );
6    }
7    // Now we have all tokens in a vector, do all *
         first?
8    for (int i=0; i<tok.size(); ++i ){
9      if ( tok[i].kind == '*' ){
10       double d = tok[i-1].value * tok[i+1].value;
11       // now what?
12     }
13   }
14   return 0;
15 }
```

Grammars

- So how to we decide what to do with our vector of Tokens?
- Again look at previous calculator code – we use a *grammar*
- A simple grammar for our calculator looks like:

```
Expression:                ( an experssion is either:
    Term                     a term,
    Expression + Term        an expression + a term, or
    Expression - Term        an expression - a term )
Term:                      ( a term is either:
    Primary                  a primary,
    Term * Primary           a term * a primary,
    Term / Primary           a term / a primary, or
    Term % Primary           a term % a primary )
Primary:                   ( a primary is either :
    Number                   a number, or
    ( Expression )           an expression in brackets )
Number:
    floating-point         ( a number is a C++ floating po
```

Example 1 – reading the grammar

- Suppose we enter 2. Is 2 an expression?
- An expression must be a term, or another expression + or - a term. If we look at term, it must be a primary or another term *, / or % a primary. If we look at primary, we see it is a number or an expression in brackets. If we look at number, we find that 2 is a floating-point literal.
- Since 2 is a floating point number, it is a number, which is a primary, which is a term, which is an expression.
- So, yes, 2 is an expression.

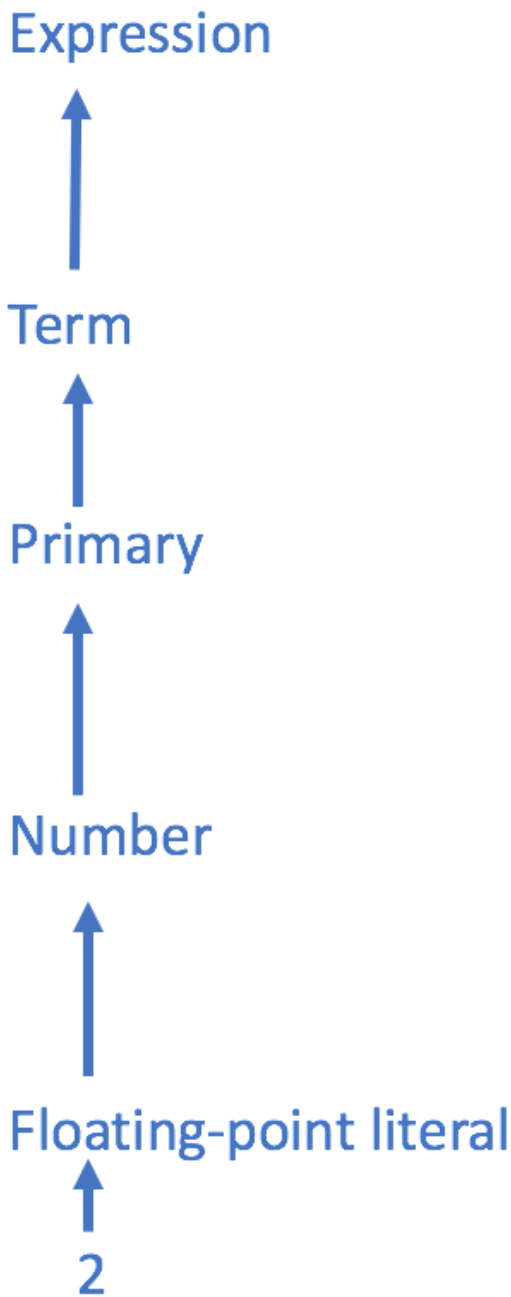
Grammar example 1 – graphically

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

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

Primary:
- Number
- (Expression)

Number:
- floating point literal



Grammar example 2 – graphically

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

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

Primary:
- Number
- (Expression)

Number:
- floating point literal

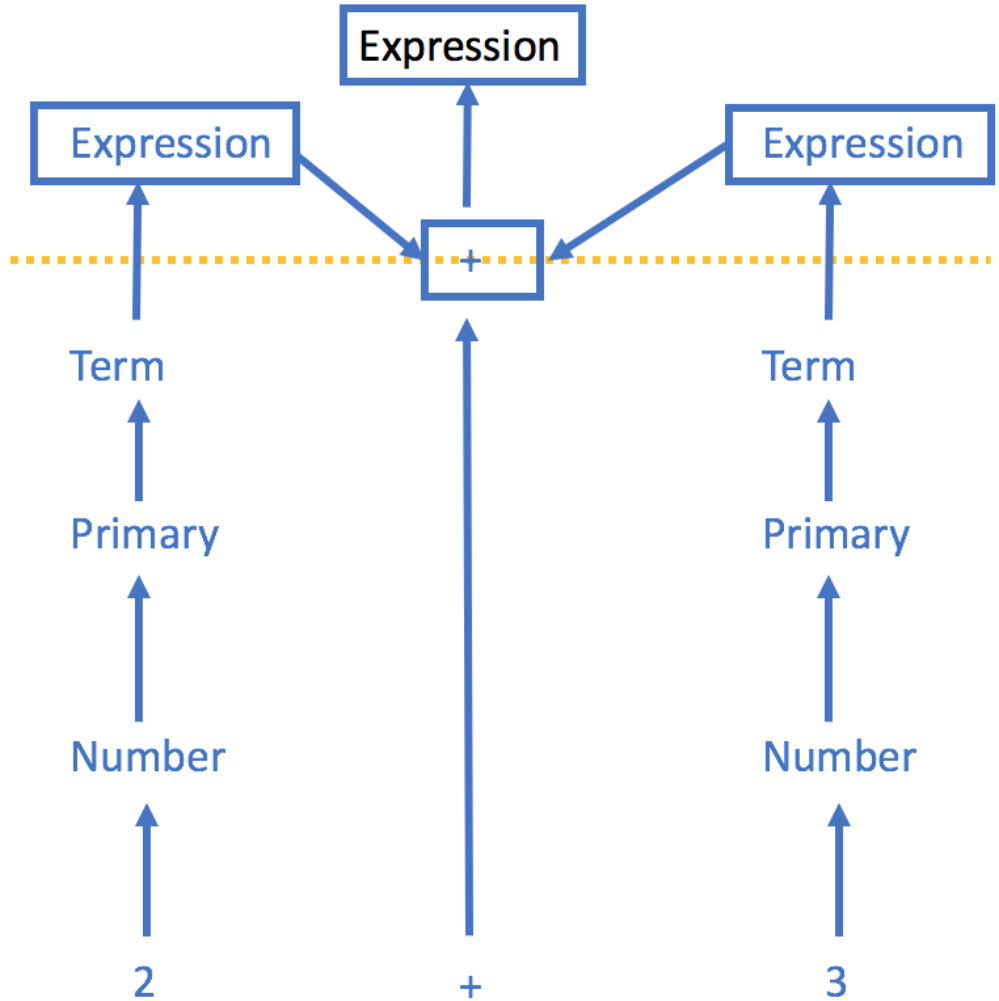


Figure: Evaluating user entry of $2 + 3$ in grammar.

Grammar example 3 – graphically

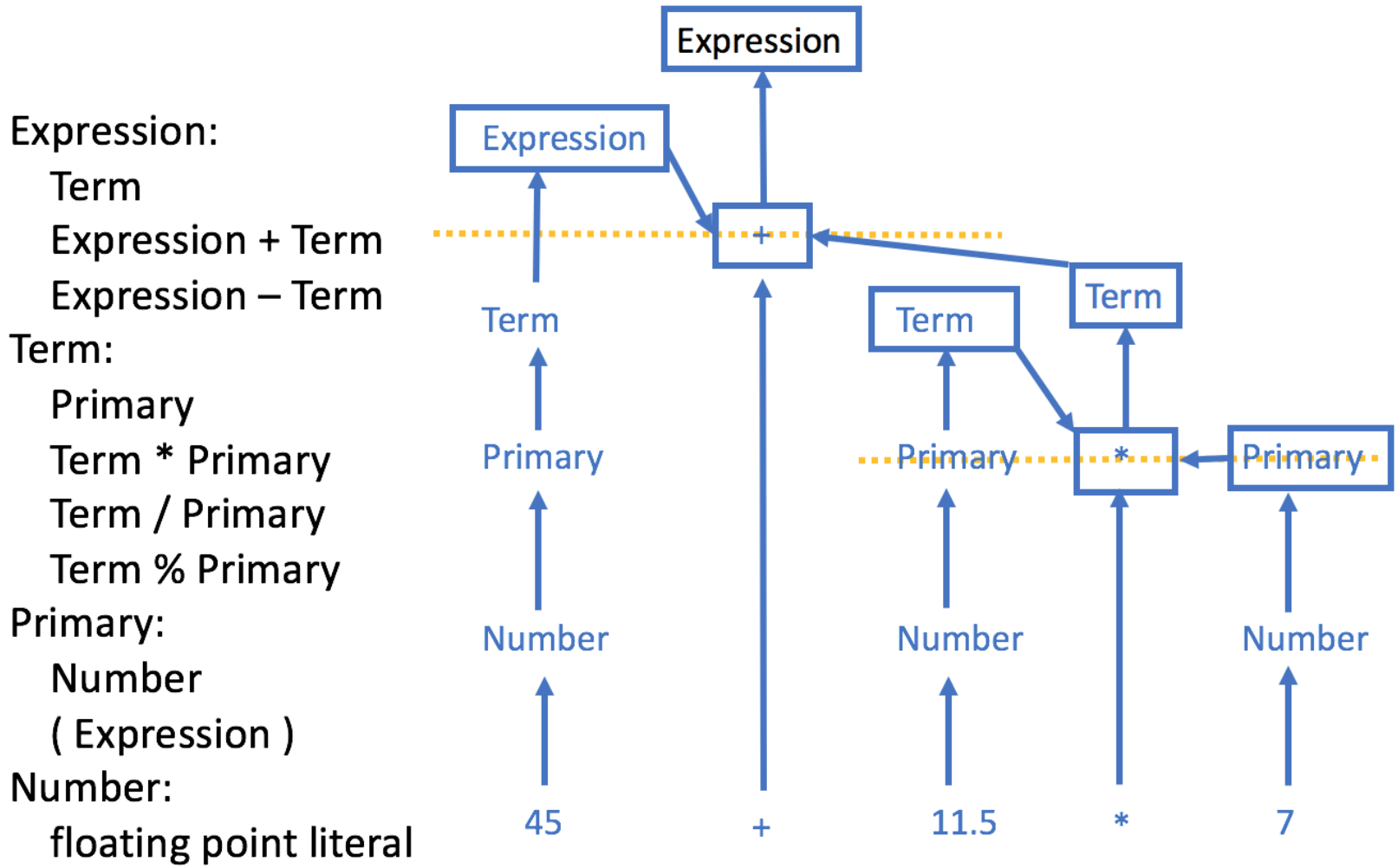


Figure: Evaluating user entry of $45 + 11.5 * 7$ in grammar.

Grammar example 4 – subset of english

- We can write a simple grammar for a few simple english sentences:

```
Sentence:
  Noun Verb
  Sentence conjunction Sentence
Conjunction:
  and
  or
  but
Noun:
  birds
  fish
  C++
Verb:
  rules
  fly
  swim
```

Grammar example 4 – graphically

Sentence:
- noun verb
- sentence conjunction sentence

Conjunction:
- and
- or
- but

Noun:
- birds
- fish
- C++

Verb:
- rules
- fly
- swim

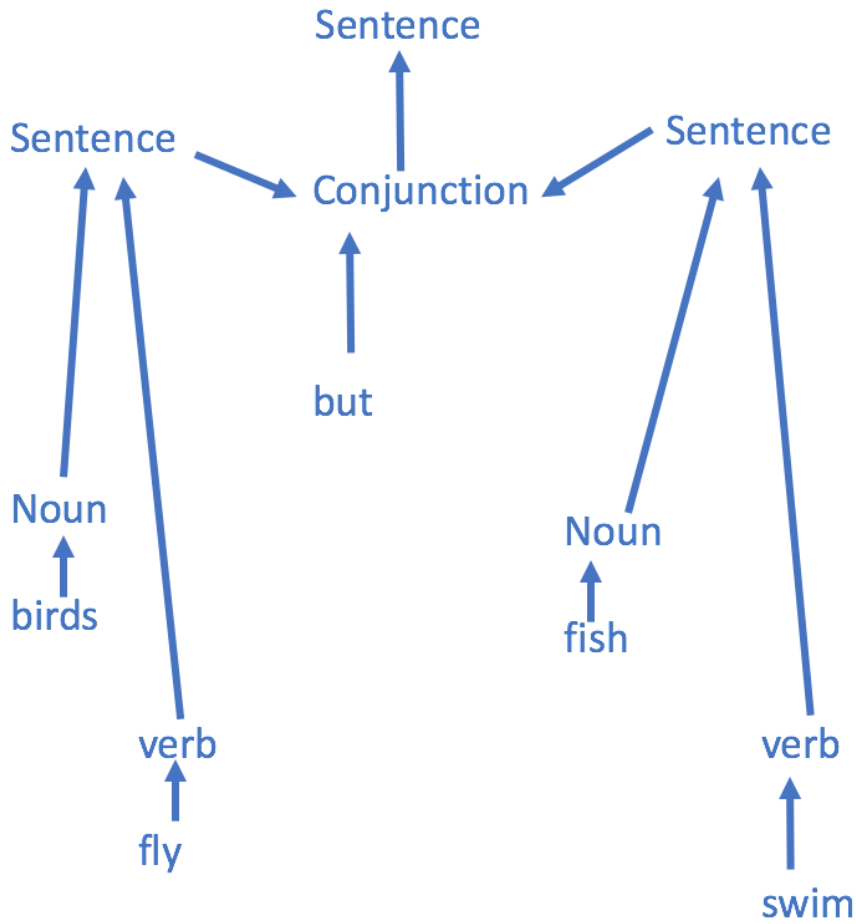


Figure: Parsing the phrase “birds fly but fish swim”

Grammar example 5 – grammar for a list

```
List:
    { sequence }
Sequence:
    Element
    Element, sequence
Element:
    A
    B
```

- Examples of lists:

  ```
  { A }
  { B }
  { A, B }
  { A, A, A, A, B }
  ```

- The following ar not lists (why?):

  ```
  { }
  A
  { A, A, A, A, B
  { A B C }
  { A, A, A, A, B, }
  ```

Turning grammar into code

- Simplest method: write one function for each grammar rule.
- A program that implements a grammar is often called a *parser*
- We need four functions:
  1. `get_token()` to read character and compose tokens (using `cin`)
  2. `expression()` deals with + and -, and calls `term()` and `get_token()`
  3. `term()` deals with *, / and %, and calls `primary()` and `get_token()`
  4. `primary()` deals with numbers and parentheses, and calls `expression()` and `get_token()`
- The function declarations:

```
Token get_token();
double expression();
double term();
double primary();
```

Turning grammar into code

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

```
1 double expression(){
2   double left = term(); // read and evaluate a term
3   Token t = get_token();
4   switch ( t.kind ){
5     case '+': return left + expression();
6     case '-': return left - expression()l
7     default: return left;
8   }
9 }
```

- Unfortunately this isn't quite right
- It works for many cases, but consider $1 - 2 - 3$

Problem with `expression()`

- $1 - 2 - 3$ evaluates as follows:
- 1 gets read as a term, then 2-3 gets read as an expression (-1), finally we evaluate 1 - (-1), which give 2
- We expected 4, so this isn't so good
- Problem is we implemented a different grammar:

```
Expression:
    Term
    Term + expression
    Term - expression
```

- Need to look for a + or - after each term and keep doing that until there are no more + or -

A working verion of `expression()`

```
1  double expression(){
2    double left = term(); // read and evaluate a term
3    Token t = get_token();
4    while ( true ){
5      switch ( t.kind ){
6      case '+':
7        left += term();
8        t = get_token();
9        break;
10     case '-':
11       left -= term();
12       t = get_token();
13       break;
14     default:
15       return left;
16     }
17   }
18 }
```

First attempt at implementing `term()`

```
Term:
  Primary
  Term * Primary
  Term / Primary
  Term % Primary
```

```
double term(){
  double left = primary();
  Token t = get_token();
  while ( true ){
    switch ( t.kind ){
    case '*': left *= primary();
              t = get_token();    break;
    case '/': left /= primary();
              t = get_token();    break;
    case '%': left %= primary();
              t = get_token();    break;
    default:  return left;
    }
  }
}
```

Note of first attempt at `term()`

- It doesn't compile! % only exists for integers – for now lets get rid of that case, and come back to it later
- If user enters 2/0 we get a divide by zero error
- We should check for divide by zero – updated code:

```
1  double term(){
2    double left = primary();
3    Token t = get_token();
4    while ( true ){
5      switch ( t.kind ){
6      case '*':
7        left *= primary();
8        t = get_token();      break;
9      case '/':{
10       double d = primary();
11       if ( d==0 ) error("term(): divide by zero");
12       left /= d;
13       t = get_token();      break;
14     default:
15       return left;
16     }
17   }
```

First attempt at implementing `primary()`

```
Primary:
  Number
  ( expression )
```

```
1  double primary(){
2    Token t = get_token();
3    switch ( t.kind ){
4      case '(': {
5        double d = expression();
6        t = get_token();
7        if ( t.kind != ')' ) error("primary() expected
             )");
8        return d;
9      }
10     case '8':
11       return t.value;
12     default:
13     error("primary expected");
14   }
15 }
```

First attempt at implementing `main()`

```
1  int main()
2  try {
3    while (cin) {
4      cout << expression << endl;
5    }
6    return 0;
7  }
8  catch ( exception & e ){
9    cerr << e.what() << endl;
10   return 1;
11 }
12 catch (...) {
13   cerr << "exception" << endl;
14   return 2;
15 }
```

Try out the first calculator `calculator00.cpp`

- It doesn't work the way we wanted:
- enter 2, newline -> no response
- enter newline -> no response
- enter 3 newline -> no response
- enter 4 newline -> it answers 2
- Now the screen looks like:

```
2

3
4
2
```

Try out the first calculator `calculator00.cpp`

- enter 5+6+7 -> response is 5
- Now the screen looks like:

```
2


3
4
2
5+6+7
5
```

- Puzzling! Also, we should update the code to distinguish between input and output.
- We add = before all output, then try entering: 1 2 3 4 + 5 6 + 7 8 + 9 10 11 12, and get:

```
=1
=4
=6
=8
=10
```

- program only outputs every third token?
- program seems to be eating input (turns out to be in expression)

Look again at `expression()`

```
double expression(){
  double left = term(); // read and evaluate a term
  Token t = get_token();
  while ( true ){
    switch ( t.kind ){
    case '+':
      left += term();
      t = get_token();
      break;
    case '-':
      left -= term();
      t = get_token();
      break;
    default:
      // we are about to return, but
      // we didn't use the token?
      return left;
    }
  }
}
```

`Token_stream()`

- What do we do with token we aren't using?
- Need a way to put it back on stream
- Rather than dealing with character stream, we introduce a `Token_stream` type
- We give it member functions `get()` to get a token, and `putback( t )`

Token_stream()

- Assuming we have implemented:

```
Token_stream ts; // define this somewhere before
    functions using it
// expression becomes:
double expression(){
  double left = term();
  Token t = ts.get();
  while ( true ){
    switch ( t.kind ) {
    case '+':
      left += term();
      t = ts.get();
      break;
    case '-':
      left -= term();
      t = ts.get();
      break;
    default:
      ts.putback( t ); // put back unused token
      return left;
    }
  }
```

Update `term()` to use `Token_stream()`

```
double term(){
  double left = primary();
  Token t = ts.get();
  while ( true ){
    switch ( t.kind ){
    case '*':
      left *= primary();
      t = ts.get();  break;
    case '/':{
      double d = primary();
      if ( d==0 ) error("term(): divide by zero");
      left /= d;
      t = ts.get();  break;
    default:
      ts.putback( t );
      return left;
    }
  }
}
```

- For `primary()`, just change `get_token()` to `ts.get()` – it uses every token

Notes on this version of calculator

- Try it out – it basically works, but down't print last answer
- Need a way to say we are done input (lets use ';')
- Need a way to quit the program (lets use 'q')
- That changes the line in main() from:

```
while (cin) cout <<"="<<expression()<<endl;
```

- To:

```
double val=0;
while (cin){
  Token t = ts.get();
  if ( t.kind == 'q' ) break;
  if ( t.kind == ';' ){ // print now
    cout << "= "<< val << endl;
  } else {
    ts.putback( t );
  }
  val = expression();
}
```

- Now calculator is useable
- Need to implement `Token_stream` type

Implementing `Token_stream()`

- Recall in the `Token` type we had the keyword `public:`
- For `Token_stream` we also need a `private:` keyword, that is used to separate the user interface from implementation details:

```
class Token_stream {
  public:
  // user interface
  // accesssible by Token_stream users

  private:
  // implementation details
  // not directly accesible to Token_stream users
};
```

Public and private parts

- Public interface should only contain what is needed by the user
- Private interface contains things needed to make the public interface work. It hides data and functions to initialize object
- The public part of `Token_stream`:

```
class Token_stream {
  public:
  Token_stream(); // make a Token_stream that
      reads from cin
  Token get();     // get a Token
  void putback( Token t ); // put token back

  private:
  // implementation details
};
```

- We used the name `putback` instead of `put` since istream has the same naming – makes it more consistent with std library

Full implementation of `Token_stream`

```
class Token_stream {
public:
  Token_stream();
  Token get();
  void putback( Token t );
private:
  bool full; // is there a Token in buffer?
  Token buffer; // here is where we keep a Token
};
// Constructor of Token_stream
Token_stream::Token_stream() : full( false ),
    buffer(0) { }
// Putback method:
void Token_stream::putback( Token t ){
  if ( full ) error("putback() into a full buffer");
  buffer = t; // copy to buffer
  full = true;
}
```

- Still have to define `get()` method, which is hardest

Defining `Token_stream::get()`

```
Token Token_stream::get(){ // get method:
  if ( full ) { // check if we return buffered value
    full = false;
    return buffer;  }
  char ch;  cin >> ch; // get character to test
  switch ( ch ){
    case ';':  case 'q': // for print and quit
    case '(': case ')': case '+': case '-':
    case '*': case '/': case '%':
      return Token( ch ); // character represents
          itself
    case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9': {
      cin.putback( ch ); // put back digit
      double val;          // so we can read a double
      cin >> val;
      return Token('8', val); }// '8' for number
    default:
    error("Bad token");
  }
}
```

Program Structure

- Overall program, with some of details ommitted is as follows:

calculator1.cpp

```
#include "std_lib_facilities.h"
class Token { /*...*/ };
class Token_stream { /*...*/ };

Token_stream::Token_stream(): full(false),
    buffer(0){ }
void Token_stream::putback( Token t ){ /*...*/ }
Token Token_stream::get(){ /*...*/ }

Token_stream ts;
double expression();                // declare for use
    in primary()
double primary(){ /*...*/ }         // for numbers and
    parentheses
double term(){ /*...*/ }            // for *, /, and %
double expression(){ /*...*/ }      // for +, -

int main() { /* ... */ }
```

Function call graph

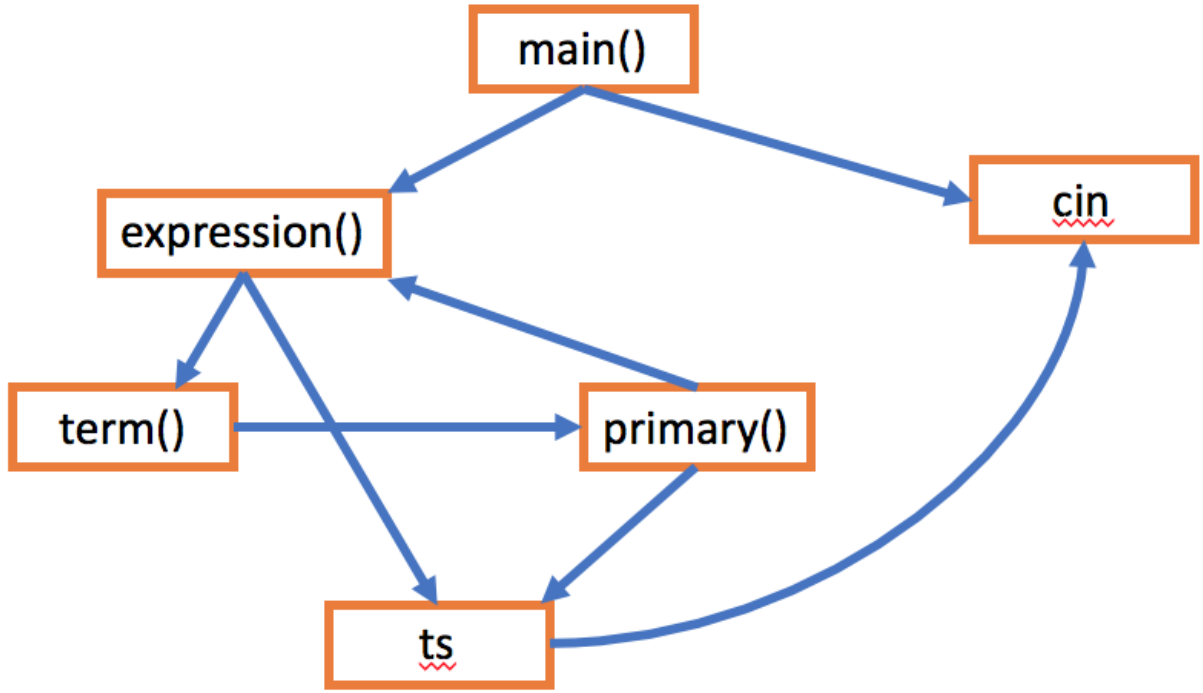


Figure: Representation of function calls.

- The program basically works now – try it out and see what you think? It is in `calculator01.cpp`
- Chapter 6 Drill – get buggy calculator working
- Chapter 6 Exercise 3 – add factorial

Outline

Writing a Program

Completing a program
- Cleaning up the program
- Adding variables to the calculator

Introduction

- Once you get program running "reasonably," you are probably only about half-way done.
- Let's try to improve the calculator – in doing so, we will see how to gradually improve code
- Some modifications we will make:
  - Add a prompt '>'
  - Test and fix some pathological cases
  - Add error handling to catch runtime_error, and wait for ~ input to exit after reporting error

Fixing problem with quit command

- Consider this input:

```
1 + 2 q
1 + 2; q
```

- The first one exits, but doesn't print the 3
- The second one prints 3, but exits with "error: primary expected"
- Error is that in main() we should also look for 'q' when we find ;

```
int main()
try {
  while (cin){
    cout <<">";
    Token t = ts.get();
    while ( t.kind == ';' ) t = ts.get(); // eat ;
    if (t.kind == 'q') {
      return 0;
    }
    ts.putback( t );
    cout << "=" << expression() << endl;
  }
  return 0;
}// catch ..
```

Negative numbers

- Consider this input: -1/2;
- It exits with "error: primary expected"
- Modify the grammar to allow using minus in primary:

```
Primary:
    Number
    ( expression )
    - Primary
    + Primary
```

- Resulting code on next slide.

Negative numbers

```
double primary(){
  Token t = ts.get();
  switch ( t.kind ){
    case '(': {
      double d = expression();
      t = ts.get();
      if ( t.kind != ')' ) error(") expected");
      return d;
    }
    case '8': return t.value;
    case '-': return - primary();
    case '+': return + primary();
    default: error("primary expected");
  }
}
```

Remainder %

- We would like to add remainder, but nly works for int
- What to do if user enters float?
- Can use `fmod` function which defines $x\%y = x - y * int(x/y)$, eg.
- $6.7\%3.3 = 6.7 - 3.3 * int(x/y) = 0.1$
- Add to term's switch statement:

```
case '%': {
  double d = primary();
  if ( d==0 ) error("% divide by zero");
  left = fmod( left, d );
  t = ts.get();
  break;
}
```

Cleaning up the code

- Using '8' as a magic constant can be replaced with a `const` definition
- Also, print, quit, prompt, and result characters can be defined:

```
const char number = '8';
const char quit = 'q';
const char print = ';';
const char prompt = '>';
const char result = '=';
/* ... */
int main()
try {
  while (cin){
    cout << prompt;
    Token t = ts.get();
    while ( t.kind == print ) t=ts.get();
    if ( t.kind == quit ) return 0;
    ts.putback(t);
    cout << result << expression() <<endl;
  }
} //... catch
```

Cleaning up main()

- Try to break functions up so that each function only does one logical action
- We can break main up so that it provides scaffolding to start/stop the program and catch errors, but move calculation loop to a new function `calculate()`

```
1  void calculate(){
2    while (cin){
3      cout << prompt;
4      Token t = ts.get();
5      while ( t.kind == print ) t=ts.get();
6      if ( t.kind == quit ) return 0;
7      ts.putback(t);
8      cout << result << expression() <<endl;
9    }
10 }
```

Cleaned up main()

```
int main()
try {
   calculate();
   return 0;
} catch ( runtime_error & e ){
   cerr << e.what() << endl;
   return 1;
} catch (...) {
   cerr<<"exception"<<endl;
   return 2;
}
```

Code layout

- Keep code so it fits on a page
- Long messy code lines can be hard to notice bugs in
- should move case conditions to be one per line, or grouped by type when they are symbols, eg in get():

```
switch (ch){
 case quit:
 case print:
 case '(':  case ')':
 case '+':  case '-':
 case '*':  case '/':
 case '%':
     return Token{ ch };
 case '.':
 case '0': case '1': case '2': case '3': case '4':
 case '5': case '6': case '7': case '8': case '9':
     {
     cin.putback( ch);
     double val;
     cin >> val;
     return Token{ number, val };
 }
```

Commenting

- When cleaning up the code, look at each part of the program to see if the original comments are:
  1. Still valid
  2. Adequate for a reader
  3. Not so verbose that they distract from the code
- The intent of the code should be documented, eg. we should write out the grammar:

```
/*  simple calculator
    Revision history:  Bjarne Stroustrup Nov. 2013
                       ....
    This program implements a basic expression
    calculator. Input is from cin, and output to
    cout.  The grammar for input is:

Expression:
    Term
    Expression + Term
    Expression - Term
Term:
    Primary
    Term * Primary
```

Recovering from errors

- Instead of exiting on an error, we can try to recover
- Move the try, catch block into the calculate function:

```
1  void calculate(){
2    while (cin ) try {
3      cout << promt;
4      Token t = ts.get();
5      while ( t.kind == print ) t = ts.get();
6      if (t.kind == quit ) return;
7      ts.putback( t );
8      cout << result << expression() << endl;
9    } catch ( exception & e ) {
10     cerr << e.what() << endl;
11     clean_up_mess();
12   }
13 }
```

Recovering from errors

- What do we do for `clean_up_mess()`?
- Want to get rid of any tokens related to the aborted calculation.
- eg. suppose we enter 1++2*3; 4+5;
- Have two options:
  1. Purge all tokens from `Token_stream`
  2. Purge all tokens from current calculations from `Token_stream`
- In the first choice, the 4+5; calculation is discarded, while in second it isn't
- Lets try to implement the second choice.

First attempt at `clean_up_mess()`

```
1 void clean_up_mess(){
2   while (true) {
3     Token t = ts.get();
4     if ( t.kind == print ) return;
5   }
6 }
```

- Unfortunately this doesn't work if there are multiple input errors
- Consider if user enters: 1@z; 1+3;
- The @ gets us into the catch clause for the while loop
- Then clean_up_mess() calls get() and reads the z, that gives another eror
- That puts us into the catch(...) which exits the program

Need a way to clean up mess

- Need a way to clean tokens out of `Token_stream` that couldn't possibly throw an exception
- We will add a new method to `Token_stream` to ignore characters up to and including a character passed to the method:

```
1 class Token_stream{
2   public:
3   Token get();
4   void putback( Token t );
5   void ignore( char c ); // discard characters up to
        including c
6   private:
7   bool full{false};
8   Token buffer;
9 };
```

Toekn_stream::ignore(char c)

```
1
2  void Token_stream::ignore( char c ){
3    if ( full && c == buffer.kind ){ // first check
         buffer
4      full = false;
5      return;
6    }
7    full = false;
8    char ch=0;
9    while ( cin >> ch ){
10     if ( ch == c ) return;
11   }
12 }
```

Updated clean up mess

- Now our cleanup function call becomes:

```
void clean_up_mess(){
  ts.ignore( print );
}
```

- Fixing error catching code can be tricky.
- Quality error handling is one mark of a professional programmer

Variables

- Lets try to add varaibles to our calculator
- Need calculator to keep (name, value) pairs, so we can access the value given the name
- We can define a `Variable` type:

```
class Variable{
  string name;
  double value;
};
```

- We can store `Variables` in a vector, and define a helper function to get the value of a variable:

```
vector< Variable > var_table;
double get_value( string s ){
  for ( const Variable & v : var_table ){
    if ( v.name == s ) return v.value;
  }
  error("get_value undefined name: ",s);
}
```

Variables continued

- Similarly we can define a `set_value()` function to give a variable a new value:

```
1 void set_value( string s, double d ){
2   for ( Variable & v : var_table ){
3     if (v.name == s ) {
4       v.value = d;
5       return;
6     }
7   }
8   error("set_value undefined variable ",s);
9 }
```

- How do we get a new variable into `var_table`?
- What does user enter to define a new variable?
- To distinguish between declaration and assignment, lets make declaration of a new variable `var`:

```
let var = 7.2;
```

Update grammar to handle Variables

- In order for calculator to handle variables, we need to update the grammar.
- The new parts of the grammar are below:

```
Calculation:
    Statement
    Print
    Quit
    Calculation statement
Statement:
    Declaration
    Expression
Declaration:
    let name = expression
```

`statement()` and `calculate()` definitions

```
1  double statement(){
2    Token t = ts.get();
3    switch ( t.kind ){
4      case let:
5        return declaration();
6      default:
7        ts.putback( t );
8        return expression();
9    }
10 }
11 void calculate(){ // statement instead of expression
12   while (cin ) try {
13     cout << prompt;
14     Token t = ts.get();
15     while ( t.kind == print ) t = ts.get();
16     if ( t.kind == quit ) return;
17     ts.putback();
18     cout << result << statement() << endl;
19   } catch (...) {
20     /*...*/
21   }
22 }
```

Defining variables

- Two parts for defining a variable with name `var`:
  1. Check whether there is already a varaible called `var`
  2. Add (var, val) to `var_table`
- We can make this two functions:

```
1  bool is_declared( string var ){ // true if var is
       in var_table
2    for ( const Variable & v : var_table ){
3      if ( v.name == var ) return true;
4    }
5    return false;
6  }
7  double define_name( string var, double val ){
8    // add (var, val) to var_table
9    if ( is_declared( var ) ) error(var, "declared
         twice");
10   var_table.push_vack( Variable{var, val} );
11   return val;
12 }
```

The `declaration()` function

- The declaration function becomes:

```
1  double declaration() {
2    // assumes we already saw let keyword
3    // handle name = expression
4    Token t = ts.get();
5    if ( t.kind != name ) error("name expected in
         declaration");
6    string var_name = t.name;
7    Token t2 = ts.get();
8    if ( t2.kind != '=' ){
9      error("= missing in declaration of ",var_name
           );
10   }
11   double d = expression();
12   define_name( var_name, d );
13   return d;
14 }
```

Modifications to Token_stream::get()

```
const char name='a';    // add name, and let as tokens
const char let ='L';
const string declkey = "let";
Token Token_stream::get(){
  if (full){
    full = false;
    return buffer;
  }
  char ch;  cin >> ch;
  switch ( ch ){
    // ... as before
    default:
    if ( isalpha( ch ) ){
      cin.putback( ch );
      string s;
      cin >> s;
      if ( s == declkey ) return Token( let );
      return Token{ name, s};
    }
    error("Bad token");
  }
}
```

Continued modifications

- Still need to modify Token to add a string to hold a variable
- Updated Token:

```
class Token {
  public:
  char kind;
  double value;
  string name;
  Token( char ch ) : kind{ch} { }
  Token( char ch, double val ) : kind{ch},
      value{val} { }
  Token( char ch, string n ): kind{ch}, name{n} { }
};
```

- Above we introduced three "constructors" used for defining new Tokens
- With this change we can try out the calculator
- We find that it works, but recall how we set name, we read a string
- That means we have to be careful to put spaces after our name

Handling no spaces after name

- We can define names to contain only letters and digits (we could also look for underscore – exercise for the student)
- Update the "default" case in Token_stream::get():

```
default:
if ( isalpha( ch ) ){
  string s;
  s += ch;
  while ( cin.get(ch) &&
          ( isalpha(ch) || isdigit(ch) ) ){
    s += ch;
  }
  cin.putback(ch);
  if ( s == decltype ) return Token{let};
  return Token{ name, s };
}
error("Bad token");
```

Predefined names

- Add pi and e as pre-defined variables in main((:

```
int main()
try {
  // pre-defined names
  define_name("pi", acos(-1.) );
  define_name("e", exp(1.) );
  calculate();
  return 0;
} catch ( exception & e ) {
  cerr << e.what() << endl;
  return 1;
} catch (...) {
  cerr << "exception" <<endl;
  return 2;
}
```

- Still need to provede the assignment operator
- And do some code cleanup again
- HW problem is Chapter 7 drill to get `calculator08buggy.cpp` working!

Week4 Done!

- Homework 3 given out last week is due Sept. 29 by 17:00
- Homework 4 given out today is due Oct. 6 by 17:00