16. Recursion 2

Building a Calculator, Streams, Formal Grammars, Extended Backus Naur Form (EBNF), Parsing Expressions

Motivation: Calculator

Goal: we build a command line calculator

Example

Input: `3 + 5`
Output: `8`
Input: `3 / 5`
Output: `0.6`
Input: `3 + 5 * 20`
Output: `103`
Input: `(3 + 5) * 20`
Output: `160`
Input: `-(3 + 5) + 20`
Output: `12`

- binary Operators +, -, *, / and numbers
- floating point arithmetics
- precedences and associativities like in C++
- parentheses
- unary operator -

Naive Attempt (without Parentheses)

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

```
Input 2 + 3 * 3 =
Result 15
```


Analyzing the Problem

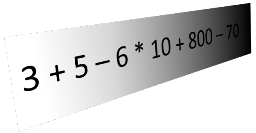
Example

Input:

$$\underbrace{13 + 4 * (15 - 7 * 3)}_{\text{Needs to be stored such that evaluation can be performed}} =$$

"Understanding" expressions requires a lookahead to upcoming symbols!

Preparational Parenthesis: Streams



A program takes inputs from a conceptually infinite input stream.

So far: command line input stream `std::cin`

```
while (std::cin >> op && op != '=') { ... }
```

↑ Consume op from `std::cin`, reading position advances.

In future we want to be able to read from files.

Example: BSD 16-bit Checksum

```
#include <iostream>

int main () {
  char c;
  int checksum = 0;
  while (std::cin >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Input: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Requires a manual termination of the input at the console [7]

Output: 67fd

[7] Ctrl-D(Unix) / Ctrl-Z(Windows) at the beginning of a line that is concluded with ENTER

Example: BSD 16-bit Checksum with a File

output: 67fd

```
#include <iostream>
#include <fstream>

int main () {
  std::ifstream fileStream ("loremispum.txt");
  char c;
  int checksum = 0;
  while (fileStream >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

returns `false` when file end is reached.

Example: BSD 16-bit Checksum

Reuse of common functionality?

Correct: with a function. But how?

Example: BSD 16-bit Checksum Generic!

Reference required: we modify the stream.

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is)
{
  char c;
  int checksum = 0;
  while (is >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  return checksum;
}
```


Equal Rights for All!

input: Lorem Yps with Gimmick
output: checksums differ

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream("loremipsum.txt");

  if (checksum (fileStream) == checksum (std::cin))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```


Why does that work?

- `std::cin` is a variable of type `std::istream`. It represents an input stream.
- Our variable `fileStream` is of type `std::ifstream`. It represents an input stream on a file.
- A `std::ifstream` *is also a* `std::istream`, with more features.
- Therefore `fileStream` can be used wherever a `std::istream` is required.

Again: Equal Rights for All!

input from `stringStream`
output: checksums differ

```
#include <iostream>
#include <fstream>
#include <sstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream ("loremipsum.txt");
  std::stringstream stringStream ("Lorem Yps mit Gimmick");

  if (checksum (fileStream) == checksum (stringStream))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```

Back to Expressions

$$13 + 4 * (15 - 7 * 3)$$

“Understanding an expression requires lookahead to upcoming symbols!

We will store symbols elegantly using recursion.

We need a new formal tool (that is independent of $\mathrm{C}{+}{+}$).

Formal Grammars

- Alphabet: finite set of symbols Σ
- Strings: finite sequences of symbols Σ^*

A formal grammar defines which strings are valid.

Mountains

- Alphabet: $\{/, \backslash\}$
- Mountains $\mathcal{M} \subset \{/, \backslash\}^*$ (valid strings)

$m' = //\backslash\backslash///\backslash\backslash\backslash$

```
          /\
     /\  /  \
    /  \/    \
```


Forbidden Mountains

- Alphabet: $\{/, \backslash\}$
- Mountains: $\mathcal{M} \subset \{/, \backslash\}^*$ (valid strings)

$m''' = /\backslash\backslash//\backslash \notin \mathcal{M}$

```
 /\  /\
   \/
```

Both sides should have the same height. A mountain cannot fall below its starting height.

Berge in Backus-Naur-Form (BNF)

mountain = "/\" | "/" mountain "\" | mountain mountain.

Rules

alternatives

nonterminal

terminal

Possible Mountains

1.
```
/\
```

2.
```
                 /\/\
  /\/\          /    \
 /    \   =>   /      \
```

3.
```
                   /\                 /\
  /\/\      /\    /  \      /\/\  /\  /  \
 /    \    /  \/      \ => /    \/  \/    \
```

It is possible to prove that this BNF describes “our” mountains, which is not completely clear a priori.

Expressions

```
-(3-(4-5))*(3+4*5)/6
```

What do we need in the BNF?

- Number , (Expression)
  -Number, -(Expression)
- Factor $*$ Factor, Factor
  Factor $*$ Factor / Factor , ...
- Term + Term, Term
  Term - Term, ...

Factor

Term

Expression

The BNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

```
factor     = unsigned_number
             | "(" expression ")"
             | "-" factor.
```


The BNF for Expressions

A term is

- factor,
- factor $*$ factor, factor / factor,
- factor $*$ factor $*$ factor, factor / factor $*$ factor, ...
- ...

We need a repetition!

EBNF

Extended Backus Naur Form: extends the BNF by

- option [] und
- optional repetition {}

```
term = factor { "*" factor | "/" factor }.
```

Remark: the EBNF is not more powerful than the BNF. But it allows a more compact representation. The construct from above can be written as follows:

```
term = factor | factor T.
T = "*" term | "+" term.
```


The EBNF for Expressions

```
factor     = unsigned_number
             | "(" expression ")"
             | "−" factor.

term       = factor { "*" factor | "/" factor }.

expression = term { "+" term | "−" term }.
```


Parsing

- **Parsing:** Check if a string is valid according to the (E)BNF.
- **Parser:** A program for parsing.
- **Useful:** From the (E)BNF we can (nearly) automatically generate a parser:
  - Rules become functions
  - Alternatives and options become `if`–statements.
  - Nonterminial symbols on the right hand side become function calls
  - Optional repetitions become `while`–statements

Functions (Parser with Evaluation)

Expression is read from an input stream.

```
// POST: extracts a factor from is
//       and returns its value
double factor (std::istream& is);

// POST: extracts a term from is
//       and returns its value
double term (std::istream& is);

// POST: extracts an expression from is
//       and returns its value
double expression (std::istream& is);
```

One Character Lookahead. . .

. . . to find the right alternative.

```
// POST: leading whitespace characters are extracted
//       from is, and the first non-whitespace character
//       is returned (0 if there is no such character)
char lookahead (std::istream& is)
{
  if (is.eof())
    return 0;
  is >> std::ws;          // skip whitespaces
  if (is.eof())
    return 0;             // end of stream
  return is.peek();     // next character in is
}
```


Cherry-Picking

. . . to extract the wanted character.

```
// POST: if ch matches the next lookahead then consume it
//       and return true. return false otherwise
bool consume (std::istream& is, char ch)
{
    if (lookahead(is) == ch){
        is >> ch;
        return true;
    }
    return false;
}
```


Evaluating Factors

```
double factor (std::istream& is)
{
  double v;
  if (consume(is, '(')){
    v = expression (is);
    consume(is, ')');
  } else if (consume(is, '-'))
    v = -factor (is);
  else
    is >> v;
  return v;
}
```

```
factor = "(" expression ")"
       | "-" factor
       | unsigned_number.
```


Evaluating Terms

```
double term (std::istream& is)
{
  double value = factor (is);
  while(true){
    if (consume(is, '*'))
      value *= factor (is);
    else if (consume(is, '/'))
      value /= factor(is)
    else
      return value;
  }
}
```

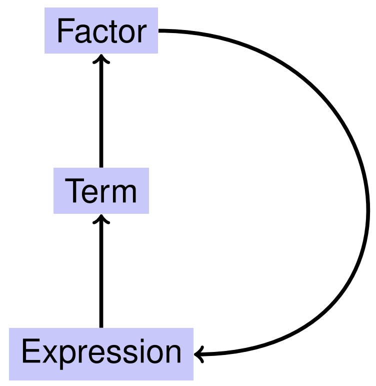
```
term = factor { "*" factor | "/" factor }
```

Evaluating Expressions

```
double expression (std::istream& is)
{
  double value = term(is);
  while(true){
    if (consume(is, '+'))
      value += term (is);
    else if (consume(is, '-'))
      value -= term(is)
    else
      return value;
  }
}
```

expression = term { "+" term | "−" term }

Recursion!



EBNF — and it works!

EBNF (`calculator.cpp`, Evaluation from left to right):

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.

term       = factor { "*" factor | "/" factor }.

expression = term { "+" term | "-" term }.
```

```
std::stringstream input ("1-2-3");
std::cout << expression (input) << "\n"; // -4
```


BNF — and it does not work!

BNF (`calculator_r.cpp`, Evaluation from right to left):

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.

term       = factor | factor "*" term | factor "/" term.

expression = term | term "+" expression | term "-" expression.
```

```
std::stringstream input ("1-2-3");
std::cout << expression (input) << "\n"; // 2
```

Analysis: Repetition vs. Recursion

Simplification: sum and difference of numbers

Beispiele

$$3, \quad 3-5, \quad 3-7-1$$

EBNF:

```
sum = value {"−" value | "+" value}.
```

BNF:

```
sum = value | value "−" sum | value "+" sum.
```

Both grammars permit the same kind of expressions.

`value`

```
double value (std::istream& is){
    double val;
    is >> val;
    return val;
}
```


EBNF Variant

```
// sum = value {"−" value | "+" value}.
double sum(std::istream& is) {
    double v = value(is);
    while(true){
        if (consume(is, '−'))
            v −= value(is);
        else if (consume(is, '+'))
            v += value(is);
        else
            return v;
    }
}
```


We test: EBNF Variant

- input: 1-2
  output: -1 ✓
- input: 1-2-3
  output: -4 ✓

BNF Variant

```
// sum = value | value "-" sum | value "+" sum.
double sum(std::istream& is){
    double v = value(is);
    if (consume(is, '-'))
        return v - sum(is);
    else if(consume(is, '+'))
        return v + sum(is);
    return v;
}
```


We test: BNF Variant

- input: 1-2
  output: -1 ✓
- input: 1-2-3
  output: 2

We Test

Is the BNF wrong ?

```
sum = value
      | value "-" sum
      | value "+" sum.
```

- No, it does only determine the validity of expressions, not over their values!
- The evaluation we have put on top naively.

Getting to the Bottom of Things

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
     v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

3	3
2 - "3"	-1
1 - "2 - 3"	2
"1 - 2 - 3"	2

What has gone wrong?

The BNF

- does officially not talk about values
- but it still suggests the wrong kind of evaluation order.

```
sum = value | value "-" sum | value "+" sum.
```

naturally leads to

$1-2-3=1-(2-3)$

A Solution: Left-Recursion

```
sum = value | sum "-" value | sum "+" value.
```

Implementation pattern from before does not work any more.
Left-recursion must be resolved to right-recursion.

This is what it looks like:

```
sum = value | value s.
s = "-" sum | "+" sum.
```

Cf. `calculator_l.cpp`