15. 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

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";

Analyzing the Problem

"Understanding" expressions requires a lookahead to upcoming symbols!
As Preparation: 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 the future we also 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.
```

Output: 67fd

Example: BSD 16-bit Checksum with a File

```
#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";
}
```

Requires a manual termination of the input at the console

```
3Ctrl-D(Unix) / Ctrl-Z(Windows) at the beginning of a line that is concluded with ENTER
```

Output: 67fd

Example: BSD 16-bit Checksum

Reuse of common functionality?
Correct: with a function. But how?
Example: BSD 16-bit Checksum Generic!
#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!
#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!
#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";
"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 C++).

A formal grammar defines which strings are valid.

Mountains

Alphabet: \{/ , \}\nMountains \( M \subset \{/ , \}\^* \) (valid strings)

\[ m' = / \// / \// / \// \]

Forbidden Mountains

Alphabet: \{/ , \}\nMountains: \( M \subset \{/ , \}\^* \) (valid strings)

\[ m''' = \// \// \\notin M \]

Both sides should have the same height. A mountain cannot fall below its starting height.
Mountains in Backus-Naur-Form (BNF)

\[
\text{mountain} = \text{"/"} | \text{"/" mountain "/"} | \text{mountain mountain.}
\]

Possible Mountains

1 \(\text{\ /}\)
2 \(\text{\ /}\ ⇒ \text{\ /}\)
3 \(\text{\ /}\ ⇒ \text{\ /}\ ⇒ \text{\ /}\)

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)
- Factor * Factor, Factor
- Factor * Factor / Factor, ...
- Term + Term, Term
- Term - Term, ...

The BNF for Expressions

A factor is

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

\[
\text{factor} = \text{unsigned_number}
| \text{"(" expression ")"}
| "-" factor.
\]

A term is

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

We need repetition!
EBNF

Extended Backus Naur Form: extends the BNF by

- option [] and
- optional repetition {}

```plaintext
term = factor { "\ast" 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:

```plaintext
term = factor | factor T.
T = "\ast" term | "+" term.
```

The EBNF for Expressions

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

term = factor { "\ast" 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.
  - Nonterminal symbols on the right hand side become function calls
  - Optional repetitions become while–statements

Functions

```plaintext
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.

```cpp
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 desired character.

```cpp
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;
}

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;
    }
return value;
}
Evaluating Expressions

do\,du\,le\,\,e\,x\,p\,r\,e\,s\,s\,i\,o\,n\,(s\,t\,d\,::\,i\,s\,t\,r\,e\,a\,m&\,i\,s)\n{\n\,d\,o\,u\,l\,t\,e\,\,v\,a\,l\,u\,e\,=\,t\,e\,r\,m(is);\n\,w\,h\,i\,l\,e(\,t\,r\,u\,e)\{\n\,\,i\,f\,(\,c\,o\,n\,s\,u\,m(is,\,'+'))\n\,\,\,v\,a\,l\,u\,e\,+=\,t\,e\,r\,m(is);\n\,\,e\,l\,s\,e\,i\,f\,(\,c\,o\,n\,s\,u\,m(is,\,'−'))\n\,\,\,v\,a\,l\,u\,e\,−=\,t\,e\,r\,m(is)\n\,\,e\,l\,s\,e\n\,\,r\,e\,t\,u\,r\,n\,\,v\,a\,l\,u\,e;\n\}\n}\n
expression = term { "+" term | "−" term }

EBNF — and it works!

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

\[
\begin{align*}
\text{factor} & \quad = \text{unsigned\_number} \\
& \quad | \quad "(" \text{ expression } ")" \\
& \quad | \quad "−" \text{ factor}. \\
\text{term} & \quad = \text{factor} \{ \"\cdot\" \text{ factor} \mid \"/\" \text{ factor} \}. \\
\text{expression} & \quad = \text{term} \{ \"+\" \text{ term} \mid \"−\" \text{ term} \}. \\
\end{align*}
\]

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

BNF — and it does not work!

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

\[
\begin{align*}
\text{factor} & \quad = \text{unsigned\_number} \\
& \quad | \quad "(" \text{ expression } ")" \\
& \quad | \quad "−" \text{ factor}. \\
\text{term} & \quad = \text{factor} \mid \text{factor} \"\cdot\" \text{ term} \mid \text{factor} \"/\" \text{ term}. \\
\text{expression} & \quad = \text{term} \mid \text{term} \"+\" \text{ expression} \mid \text{term} \"−\" \text{ expression}.
\end{align*}
\]

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

Recursion!

Expression

Term

Factor
Analysis: Repetition vs. Recursion

Simplification: sum and difference of numbers

Examples

3, 3 − 5, 3 − 7 − 1

EBNF:

\[
\text{sum} = \text{value} \{ "-" \text{value} \mid "+" \text{value} \}.
\]

BNF:

\[
\text{sum} = \text{value} \mid \text{value} "-" \text{sum} \mid \text{value} "+" \text{sum}.
\]

Both grammars permit the same kind of expressions.

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?

No, it does only determine the validity of expressions, not 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
What has gone wrong?

The BNF

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

\[ \text{sum} = \text{value} \mid \text{value} \, \text{"−"} \, \text{sum} \mid \text{value} \, \text{"+"} \, \text{sum}. \]

naturally leads to

\[ 1 - 2 - 3 = 1 - (2 - 3) \]

A Solution: Left-Recursion

\[ \text{sum} = \text{value} \mid \text{sum} \, \text{"−"} \, \text{value} \mid \text{sum} \, \text{"+"} \, \text{value}. \]

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

This is what it looks like:

\[ \text{sum} = \text{value} \mid \text{value} \, \text{s}. \]
\[ \text{s} = \text{"−"} \, \text{sum} \mid \text{"+"} \, \text{sum}. \]

Cf. calculator_l.cpp