17. Recursion 2

Building a Calculator, 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: 12

Naive Attempt (without Parentheses)

```cpp
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

Example
Input:
13 + 4 ∗ (15 − 7 ∗ 3) =

Needs to be stored such that evaluation can be performed
Analyzing the Problem

13 + 4 \times (15 - 7 \times 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 C++)

Formal Grammars

- Alphabet: finite set of symbols
- Strings: finite sequences of symbols

A formal grammar defines which strings are valid.

To describe the formal grammar, we use:

*Extended Backus Naur Form (EBNF)*

Expressions

\[-(3-(4-5)) \times (3+4 \times 5)/6\]

What do we need in a grammar?

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

Factor

Term

Expression
The EBNF for Expressions

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

\[
\text{factor} = \text{unsigned\_number} \mid (\text{expression}) \mid -\text{factor}.
\]

A term is
- factor,
- factor * factor, factor / factor,
- factor * factor * factor, factor / factor * factor, ...
- ...

\[
\text{term} = \text{factor} \{ \text{"\*" factor | "/" factor} \}.
\]

Parsing

- Parsing: Check if a string is valid according to the EBNF.
- Parser: A program for parsing.
- Useful: From the EBNF 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
### Rules

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

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

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

### Functions (Parser)

Expression is read from an input stream.

```cpp
// POST: returns true if and only if in_stream = factor ...
// and in this case extracts factor from in_stream
bool factor (std::istream& in_stream);

// POST: returns true if and only if in_stream = term ..., 
// and in this case extracts all factors from in_stream
bool term (std::istream& in_stream);

// POST: returns true if and only if in_stream = expression ..., 
// and in this case extracts all terms from in_stream
bool expression (std::istream& in_stream);
```

### Functions (Parser with Evaluation)

Expression is read from an input stream.

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

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

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

### One Character Lookahead...

... to find the right alternative.

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

...to extract the desired character.

// POST: if expected matches the next lookahead then consume it and return true; return false otherwise
bool consume (std::istream& in_stream, char expected)
{
    if (lookahead(in_stream) == expected){
        in_stream >> expected; // consume one character
        return true;
    }
    return false;
}

Evaluating Factors

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

Evaluating Terms

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

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

Evaluating Expressions

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

expression = term { "+" term | "−" term }.
Recursion!

EBNF — and it works!

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

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

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

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

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

Calculating with Rational Numbers

18. Structs

Rational Numbers, Struct Definition

- Rational numbers \( \mathbb{Q} \) are of the form \( \frac{n}{d} \) with \( n \) and \( d \) in \( \mathbb{Z} \)
- C++ does not provide a built-in type for rational numbers

**Goal**

We build a C++-type for rational numbers ourselves! 😊
### Vision

**How it could (will) look like**

```cpp
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;
std::cout << "Rational number s =? ";
rational s;
std::cin >> s;
```

// computation and output

```cpp
std::cout << "Sum is " << r + s << "\n";
```

### A First Struct

**Invariant:** specifies valid value combinations (informal).

```cpp
struct rational {
    int n; // member variable (numerator)
    int d; // INV: d != 0
};
```

member variable (denominator)

- struct defines a new *type*
- formal range of values: *cartesian product* of the value ranges of existing types
- real range of values: `rational ⊊ int × int`.

### Accessing Member Variables

```cpp
struct rational {
    int n;
    int d; // INV: d != 0
};
rational add (rational a, rational b) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

Meaning: every object of the new type is represented by two objects of type `int` the objects are called `n` and `d`.

### A First Struct: Functionality

A struct defines a new *type*, not a *variable*!

```cpp
struct rational {
    int n;
    int d; // INV: d != 0
};
```

// new type rational

```cpp
struct rational {
    int n; // INV: d != 0
    int d;
};
```

// POST: return value is the sum of `a` and `b`

```cpp
rational add (const rational a, const rational b) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

member access to the int objects of `a`.
// Input
rational r;
std::cout << "Rational number r:\n";
std::cout << " numerator =? ";
std::cin >> r.n;
std::cout << " denominator =? ";
std::cin >> r.d;

// Input s the same way
rational s;
...

// computation
const rational t = add (r, s);

// output
std::cout << "Sum is " << t.n << "/" << t.d << ".\n";

struct Definitions

struct T {
T1 name1;
T2 name2;
... ...
Tn namen;
};

Range of Values of T: T1 × T2 × ... × Tn

Struct Definitions: Examples

struct rational_vector_3 {
  rational x;
  rational y;
  rational z;
};

underlying types can be fundamental or user defined
Struct Definitions: Examples

```c
struct extended_int {
    // represents value if is_positive==true
    // and −value otherwise
    unsigned int value;
    bool is_positive;
};
```

the underlying types can be different

Structs: Accessing Members

- `expr.name_k`: expression of type `T_k`; value is the value of the object designated by `name_k`
- `T`: expression of struct-type
- `name`: name of a member-variable of type `T`
- `.`: member access operator

Structs: Initialization and Assignment

Default Initialization:
```c
rational t;
```

- Member variables of `t` are default-initialized
- for member variables of fundamental types nothing happens (values remain undefined)

Initialization:
```c
rational t = {5, 1};
```

- Member variables of `t` are initialized with the values of the list, according to the declaration order.
Structs: Initialization and Assignment

Assignment:

rational s;
...
rational t = s;

- The values of the member variables of s are assigned to the member variables of t.

Structs: Initialization and Assignment

Initialization:

rational t = add (r, s);

t is initialized with the values of add(r, s)

Structs: Initialization and Assignment

Assignment:

rational t;
t = add (r, s);

- t is default-initialized
- The value of add (r, s) is assigned to t
Comparing Structs?

For each fundamental type (int, double, ...) there are comparison operators == and !=, not so for structs! Why?

- member-wise comparison does not make sense in general...
- ...otherwise we had, for example, \( \frac{2}{3} \neq \frac{4}{6} \)

Structs as Function Arguments

```cpp
void increment(rational dest, const rational src)
{
    dest = add(dest, src); // modifies local copy only
}
```

Call by Value!

```cpp
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment(b, a); // no effect!
std::cout << b.n << "/" << b.d; // 1 / 2
```

Structs as Function Arguments

```cpp
void increment(rational & dest, const rational src)
{
    dest = add(dest, src);
}
```

Call by Reference

```cpp
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment(b, a);
std::cout << b.n << "/" << b.d; // 2 / 2
```

User Defined Operators

Instead of

```cpp
rational t = add(r, s);
```

we would rather like to write

```cpp
rational t = r + s;
```

This can be done with Operator Overloading (→ next week).