15. 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: 160
Input: `-(3 + 5) + 20`
Output: 12

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

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";
```

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

Analyzing the Problem

Example

Input:

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

Needs to be stored such that evaluation can be performed

Analyzing the Problem

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

"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 $C++$).

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

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

What do we need in a grammar?

- Number , (Expression)
  −Number, −(Expression) → Factor
- Factor ∗ Factor, Factor
  Factor / Factor , ... → Term
- Term + Term, Term
  Term − Term, ... → Expression

The EBNF for Expressions

A factor is

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

non-terminal symbol

$$\begin{aligned}
\text{factor} \quad &= \text{unsigned\_number} \\
&| \text{ "(" expression ")"} \\
&| \text{ "}-\text{" factor}.
\end{aligned}$$

alternative

terminal symbol

The EBNF for Expressions

A term is

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

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

optional repetition

The EBNF for Expressions

$$\begin{aligned}
\text{factor} \quad &= \text{unsigned\_number} \\
&| \text{ "(" expression ")"} \\
&| \text{ "}-\text{" factor}.
\end{aligned}$$

$$\text{term} \quad = \text{factor } \{ \text{ "}*\text{" factor } | \text{ "/" factor } \}.$$

$$\text{expression} = \text{term } \{ \text{ "+" term } | \text{ "}-\text{" term } \}.$$

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.
  - Nonterminial symbols on the right hand side become function calls
  - Optional repetitions become `while`–statements

Rules (Parser)

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 is = factor ...
//       and in this case extracts factor from is
bool factor (std::istream& is);

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

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

Functions (Parser with Evaluation)

Expression is read from an input stream.

```cpp
// 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
// 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())        // eof: end of file (checks if stream is finished)
        return 0;
    is >> std::ws;       // skip all whitespaces
    if (is.eof())
        return 0;        // end of stream
    return is.peek();    // next character in is
}
```

Cherry-Picking

. . . to extract the desired character.

```cpp
// 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

```cpp
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;
}
```

$$factor = \text{"("} \; expression \; \text{")"}$$
$$\qquad | \; \text{"-"} \; factor$$
$$\qquad | \; unsigned\_number.$$

Evaluating Terms

```cpp
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;
    }
}
```

$$term = factor \; \{ \; \text{"*"} \; factor \; | \; \text{"/"} \; factor \; \}.$$

Evaluating Expressions

```cpp
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;
    }
}
```

$$expression = term \; \{ \; \text{"+"} \; term \; | \text{"-"} \; term \; \}.$$

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 }.
```

```cpp
std::stringstream input ("1−2−3");
std::cout << expression (input) << "\n"; // −4
```
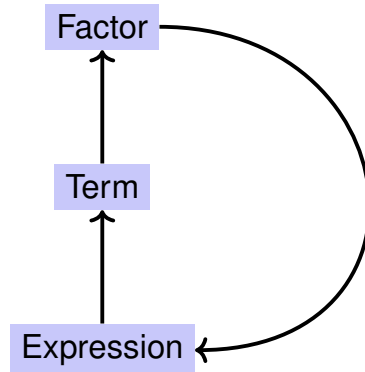
16. Structs

Rational Numbers, Struct Definition

Calculating with Rational Numbers

- Rational numbers ($\mathbb{Q}$) are of the form $\dfrac{n}{d}$ 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
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;
std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

// computation and output
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 (**n**umerator)

member variable (**d**enominator)

- `struct` defines a new *type*
- formal range of values: *cartesian product* of the value ranges of existing types
- real range of values: $\texttt{rational} \subsetneq \texttt{int} \times \texttt{int}$.

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;
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

A First Struct: Functionality

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

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

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

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

member access to the `int` objects of `a`.

Input

```cpp
// Input r
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;
...
```

Vision comes within Reach ...

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

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

Struct Definitions

name of the new type (identifier)

names of the underlying types

```
struct T {
  T₁ name₁;
  T₂ name₂;
  ⋮    ⋮
  Tₙ nameₙ;
};
```

names of the *member variables*
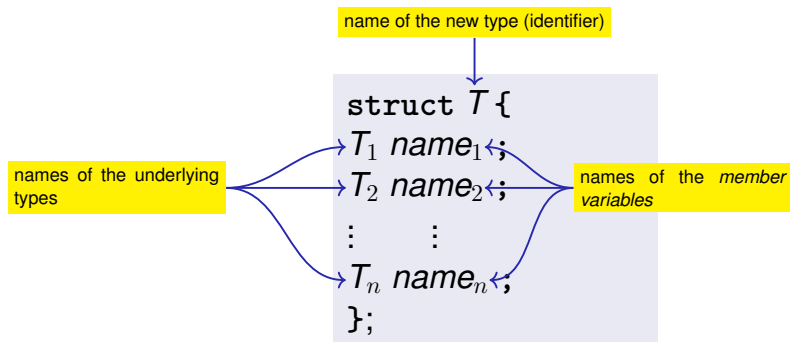
Range of Values of T: $T_1 \times T_2 \times ... \times T_n$

Struct Defintions: Examples

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

underlying types can be fundamental or user defined

Struct Definitions: Examples

```
struct extended_int {
  // represents value if is_positive==true
  // and −value otherwise
  unsigned int value;
  bool is_positive;
};
```
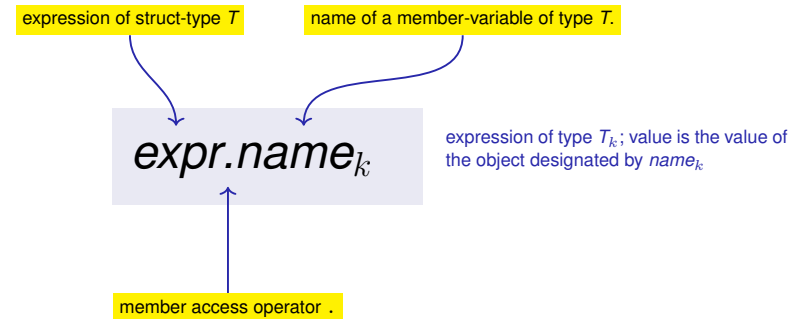
the underlying types can be different

Structs: Accessing Members

expression of struct-type T

name of a member-variable of type T.

$$expr.name_k$$

expression of type T_k; value is the value of the object designated by $name_k$

member access operator .

Structs: Initialization and Assignment

Default Initialization:

```
rational t;
```

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

Structs: Initialization and Assignment

Initialization:

```
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

```
t.n         .n
    = add (r, s)      ;
t.d         .d
```

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`

Structs: Initialization and Assignment

```
rational s;  ←── member variables are uninitialized


rational t = {1,5};  ←── member-wise initialization:
                          t.n = 1, t.d = 5

rational u = t;  ←── member-wise copy


t = u;  ←── member-wise copy


rational v = add (u,t);  ←── member-wise copy
```

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

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

Call by Value !

```
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

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

Call by Reference

```
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

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

we would rather like to write

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

This can be done with *Operator Overloading ($\rightarrow$ next week)*.