

17. Recursion 2

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

Motivation: Calculator

Input: 3 + 5

Output: 8

- binary Operators +, -, *, / and numbers

Motivation: Calculator

Input: 3 / 5

Output: 0.6

- binary Operators +, -, *, / and numbers
- floating point arithmetic

Motivation: Calculator

Input: $3 + 5 * 20$

Output: 103

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

Motivation: Calculator

Input: $(3 + 5) * 20$

Output: 160

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

Motivation: Calculator

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)

```
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";
```

Seems to work...

```
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 1 * 2 * 3 * 4 =
Result 24
```


Oops, Multiplication first...

```
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

Input:

13 + ...

Analyzing the Problem

Input:

$$13 + 4 * \dots$$

Analyzing the Problem

Input:

$$13 + 4 * (15 - ...$$


Analyzing the Problem

Input:

$$13 + 4 * (15 - 7 * ...$$

Analyzing the Problem

Input:

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


Needs to be stored such that
evaluation can be performed

Analyzing the Problem

Result:

$$13 + 4*(15 - 21)$$

Analyzing the Problem

Result:

$$13 + 4 * (-6)$$

Analyzing the Problem

Result:

$$13 + (-24)$$

Analyzing the Problem

Result:

-11

Analyzing the Problem

Expression:

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

This

Analyzing the Problem

Expression:

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

This lecture

Analyzing the Problem

Expression:

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

This lecture is

Analyzing the Problem

Expression:

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

This lecture is pretty

Analyzing the Problem

Expression:

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

This lecture is pretty much

Analyzing the Problem

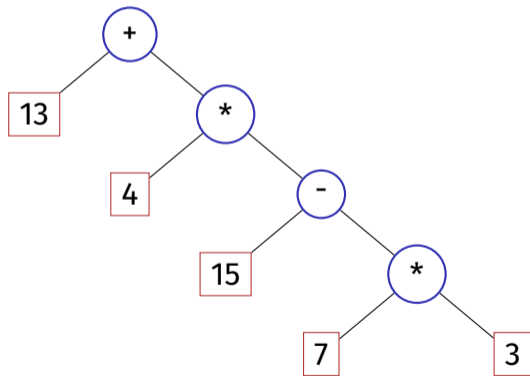
Expression:

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

This lecture is pretty much recursive.

Analyzing the Problem

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



Analyzing the Problem

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

Analyzing the Problem

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

“Understanding an expression requires lookahead to upcoming symbols!”

Analyzing the Problem

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

“Understanding an expression requires lookahead to upcoming symbols!
We will store symbols elegantly using recursion.

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

Formal Grammars

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

A formal grammar defines which strings are valid.

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)

What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?

Niklaus Wirth
Federal Institute of Technology (ETH), Zürich, and
Xerox Palo Alto Research Center

Key Words and Phrases: syntactic description
language, extended BNF
CR Categories: 4.20

The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietary circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasympols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.
4. It avoids the use of an explicit symbol for the empty string (such as (empty) or ϵ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax      = {production}.
production = identifier "=" expression " ".
expression  = term {"|" term}.
term        = factor {factor}.
factor      = identifier | literal | "(" expression ")" |
              "[" expression "]" | "{" expression "}".
literal     = " " " " character {character} " " " " .
```

Repetition is denoted by curly brackets, i.e. {a} stands for ϵ | a | aa | aaa | Optionality is expressed by square brackets, i.e. [a] stands for ϵ | a. Parentheses merely serve for grouping, e.g. (a|b|c stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

Number

An integer is a **sequence of digits**. A **sequence of digits** is

Number

An integer is a sequence of digits. A sequence of digits is

- a digit 2

Number

An integer is a **sequence of digits**. A **sequence of digits** is

■ a **digit** or **2**

■ a **digit** followed by a **sequence of digits**

2 **0** **1** **9**

Number

An integer is a **sequence of digits**. A **sequence of digits** is

■ a **digit** or 2

■ a **digit** followed by a **sequence of digits**

2 0 1 9

`number = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit | digit digits .`

Number

An integer is a **sequence of digits**. A **sequence of digits** is

■ a **digit** or **2**

■ a **digit** followed by a **sequence of digits**



number = digits .

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

digits = digit | digit digits .

alternative

terminal symbol

non-terminal symbol

Number (non-recursive)

An integer is a sequence of digits. A sequence of digits is

■ a digit, or

2

■ a digit followed by an arbitrary number of digits

2 0 1 9

number = digits .

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

digits = digit { digit } .

Number (non-recursive)

An integer is a sequence of digits. A sequence of digits is

■ a digit, or

2

■ a digit followed by an arbitrary number of digits

2 0 1 9

number = digits .

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

digits = digit { digit } .

optional repetition

Formal Grammars: Viewpoints

Production: generate valid strings

$$L = 'F' \mid L '+' L '+' .$$

(Lindenmayer) generates

'F', 'F+F+', 'F+F++F+F++', 'F+F++F+F+++F+F++F+F++++' (etc.)

Other applications: generation of test programs

Formal Grammars: Viewpoints

Production: generate valid strings

$$L = 'F' \mid L '+' L '+' .$$

(Lindenmayer) generates

'F', 'F+F+', 'F+F++F+F++', 'F+F++F+F+++F+F++F+F++++' (etc.)

Other applications: generation of test programs

Analysis (Parsing): Check *if and how* a string can be generated from a grammar

Use: Compiler, Interpreter, computer algebra systems (e.g. Mathematica, Matlab), Pretty-Printer, regular expressions (e.g. text search, grep, sed) etc.

Expressions

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

What do we need in a grammar?

Expressions

$$-(\underline{3} - (\underline{4} - \underline{5})) * (\underline{3} + \underline{4} * \underline{5}) / \underline{6}$$

What do we need in a grammar?

- Number

Expressions

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

What do we need in a grammar?

- Number, (?)

Expressions

$$\underline{-} (3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number, (?)
-Number, -(?)

Expressions

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- ? * ?, ? / ?, ...

Expressions

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- ? * ?, ? / ?, ...
- ? - ?, ? + ?, ...

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- ? * ?, ? / ?, ...
- ? - ?, ? + ?, ...

Factor

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- Factor * Factor,
Factor / Factor, ...
- ? - ?, ? + ?, ...

Factor

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- Factor * Factor,
Factor / Factor, ...
- ? - ?, ? + ?, ...

Factor

Term

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- Factor * Factor, Factor
Factor / Factor, ...
- ? - ?, ? + ?, ...

Factor

Term

Expressions

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- Factor * Factor, Factor
Factor / Factor, ...
- Term + Term,
Term - Term, ...

Factor

Term

Expressions

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- Factor * Factor, Factor
Factor / Factor, ...
- Term + Term,
Term - Term, ...

Factor

Term

Expression

Expressions

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

What do we need in a grammar?

- Number, (?)
-Number, -(?)
- Factor * Factor, Factor
Factor / Factor, ...
- Term + Term, **Term**
Term - Term, ...

Factor

Term

Expression

Expressions

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

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,

The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses

The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses

The EBNF for Expressions

A factor is

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

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

The EBNF for Expressions

A factor is

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

factor = number
| "(" expression ")"
| "-" factor.

non-terminal symbol

terminal symbol

alternative

The EBNF for Expressions

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

Implication: a factor starts with

- a digit, or
- with “(”, or
- with “-”.

The EBNF for Expressions

A term is

- factor,

The EBNF for Expressions

A term is

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

The EBNF for Expressions

A term is

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

The EBNF for Expressions

A term is

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

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

The EBNF for Expressions

A term is

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

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

optional repetition

The EBNF for Expressions

digit = '0' | '1' | '2' | ... | '9'.

number = digit {digit}.

factor = number
| "(" expression ")"
| "-" factor.

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

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

Parsing

- **Parsing:** Check if a string is valid according to the EBNF.

Parsing

- **Parsing:** Check if a string is valid according to the EBNF.
- **Parser:** A program for parsing.

Parsing

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

digit = '0' | '1' | '2' | ... | '9'.

number = digit {digit}.

factor = number
| "(" expression ")"
| "-" factor.

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

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

Expression is read from an input stream.

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

Expression is read from an input stream.

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

```
// POST: the next character at the stream is returned (but not consumed)
//       if no input is available, 0 is returned
char peek (std::istream& input){
    char ch = input.peek(); // non consuming read, can fail
    if input.good() return ch;
    return 0; // end of stream or error
}

// POST: leading whitespace characters are extracted
//       from input, and the first non-whitespace character is returned (but
//       if an error or end of stream occurs, 0 is returned
char lookahead (std::istream& input) {
    input >> std::ws; // skip whitespaces
    return peek(input);
}
```

Parse numbers

```
bool isDigit(char ch){
    return ch >= '0' && ch <= '9';
}
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int number (std::istream& input){
    char ch = lookahead(input);
    assert(isDigit(ch));
    unsigned int num = 0;
    while(isDigit(ch) && input >> ch){ // read remaining digits
        num = num * 10 + ch - '0';
        ch = peek(input);
    }
    return num;
}
```

number =digit { digit }.

digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.

Cherry-Picking

...to extract the desired character.

```
// PRE: Valid input stream input, expected > 0
// POST: If ch matches the next lookahead then it is consumed and true is returned
//        otherwise no character is consumed and false is returned
bool consume(std::istream& input, char expected) {
    assert(expected > 0);
    if (lookahead(input) == expected) {
        input.get();
        assert(input.good());
        return true;
    }
    return false;
}
```

Evaluating Factors

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

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

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!

Factor

Term

Expression

Recursion!

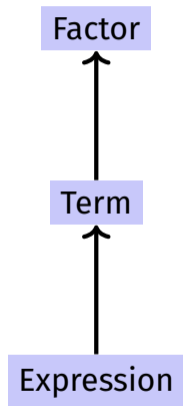
Factor

Term

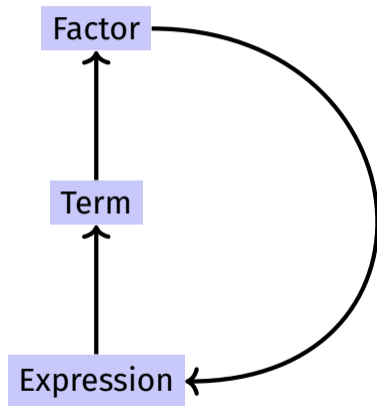
Expression



Recursion!



Recursion!



EBNF — and it works!

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

```
factor      = number  
             | "(" expression ")"  
             | "-" factor.  
  
term        = factor { "*" factor | "/" factor }.  
  
expression = term { "+" term | "-" term }.
```

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

18. Structs

Rational Numbers, Struct Definition

Calculating with Rational Numbers

- 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

Calculating with Rational Numbers

- 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

```
// 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

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

A First Struct

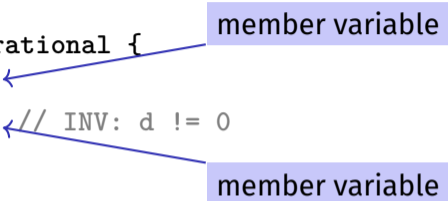
```
struct rational {  
    int n; ← member variable (numerator)  
    int d; ← // INV: d != 0  
};  
           ← member variable (denominator)
```

A First Struct

```
struct rational {  
    int n; ← member variable  
    int d; ← // INV: d != 0  
};
```

member variable

member variable



- struct defines a new type

A First Struct

```
struct rational {  
    int n; ← member variable  
    int d; ← // INV: d != 0  
};
```

member variable

member variable

- struct defines a new type
- formal range of values: *cartesian product* of the value ranges of existing types

A First Struct

```
struct rational {  
    int n; ← member variable  
    int d; ← // INV: d != 0  
};  
           ← member variable
```

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

Accessing Member Variables

```
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;  
}
```

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

Input

```
// 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 ...

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

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

Struct Definitions: Examples

```
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: Initialization and Assignment

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

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

Structs: Initialization and Assignment

```
rational s;
```

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

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t; ← member-wise copy
```

```
t = u;
```

```
rational v = add (u,t);
```

Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u; ← member-wise copy
```

```
rational v = add (u,t);
```

Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u;
```

```
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?

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...

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, $\frac{2}{3} \neq \frac{4}{6}$

User Defined Operators

Instead of

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

we would rather like to write

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

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).