

15. Recursion 2

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

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
```

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 -

489

490

Analyzing the Problem

Example

Input:

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

Needs to be stored such that evaluation can be performed

491

492

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)

Short Communications
Programming Languages

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, 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.

Communications
of
the ACM

November 1977
Volume 20
Number 11

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasymbols 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 a|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.

Received January 1977; revised February 1977

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

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

non-terminal symbol (pointing to "(")

terminal symbol (pointing to "-")

alternative (pointing to the vertical bar "|")

497

The EBNF for Expressions

A term is

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

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

optional repetition (pointing to the curly braces "{ }")

498

The EBNF for Expressions

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

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

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

499

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

500

Rules

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

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

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

501

Functions

(Parser)

Expression is read from an input stream.

```
// 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);
```

502

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);
```

503

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

504

Cherry-Picking

... to extract the desired 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;
}
```

505

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.
```

506

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 }.
```

507

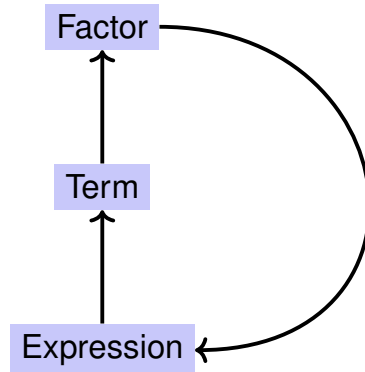
Evaluating Expressions

```
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 }.
```

508

Recursion!



16. Structs

Rational Numbers, Struct Definition

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
```

509

510

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! 😊

511

512

Vision

How it could (will) look like

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

Invariant: specifies valid value combinations (informal).

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

513

514

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;
}
```

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

515

A First Struct: Functionality

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

```
// 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`.

516

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 ...

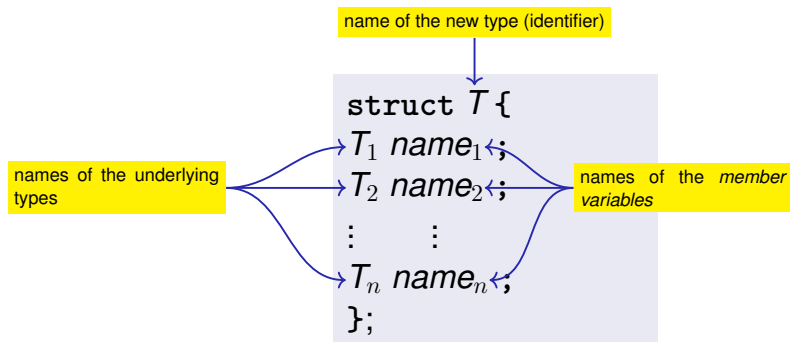
```
// computation
const rational t = add (r, s);

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

517

518

Struct Definitions



Range of Values of T : $T_1 \times T_2 \times \dots \times T_n$

Struct Definitions: Examples

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

underlying types can be fundamental or **user defined**

519

520

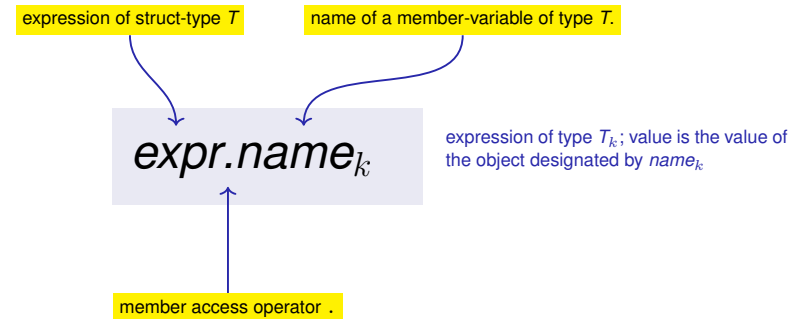
Struct Definitions: Examples

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

the underlying types can be **different**

521

Structs: Accessing Members



522

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)

523

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.

524

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`.

525

Structs: Initialization and Assignment

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

Initialization:

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

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

526

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`

527

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
```

528

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
```

529

530

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
```

531

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* (→ next week).

532