17. 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
```

- 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";
```

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


Analyzing the Problem

Example

Input:

$$\underbrace{13 + 4 * (15 - 7 * 3)}_{\text{Needs to be stored such that evaluation can be performed}} =$$

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++).

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 proprietory 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:

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 a | ϵ. 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.

Received January 1977; revised February 1977

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

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.

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

non-terminal symbol

terminal symbol

alternative

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

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

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

expression = term { "+" term |"−" 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

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);
```


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())
        return 0;           // end of stream
    return is.peek();       // next character in is
}
```


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;
}
```


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.

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;
    }
}
```

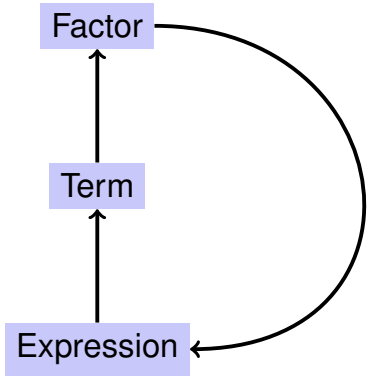
term = factor { "*" factor | "/" factor }.

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 }.

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 }.
```

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