Informatik - AS19

Exercise 9: Recursion and EBNF

Handout: 11. Nov. 2019 06:00

Due: 18. Nov. 2019 18:00

Task 1: Trains

Open Task

Task

The following EBNF defines a language for the description of train formations for railways:

```
train        = "[" ( open | compositions ) "]".
open         = loco cars.
loco         = "*" | "*" loco.
cars         = "-" | "-" cars.
compositions = composition { composition }.
composition  = "<" open loco ">".
```

Write a program that validates whether a string constitutes a valid train formation according to the given EBNF. The input is a string `S`, and the output is must be either `valid` if `S` is a valid train formation, otherwise `invalid`.

Rules: We consider solutions as incorrect, if they:

1. Do not use recursion.
2. Output `valid` and `invalid` at the same time.

Note: This program can be implemented analogously to the expression parser seen in the lecture. The most significant difference is that the expression parser from the lecture computed the result of an expression while here you should test for validity of an expression.

Map each production rule to a function, and use functions `lookahead` and `consume` from the lecture to implement alternatives. We provide these two functions in the template.

Additionally, we provide a template function `train`. Finish its implementation and add other functions corresponding to the remaining productions.

Input

A character string. Examples:

```
[<*-*><***----------******>]
```

```
[<***>]
```

Output

`valid` if the input constitues a valid train formation, `invalid` otherwise.

Respective results for the input examples:

```
valid
```

```
invalid
```

Task 2: Money

Open Task

Task

In how many ways can you own CHF 1? Despite its somewhat philosophical appearance, the question is a mathematical one. Given some amount of money, in how many ways can you partition it using the available denominations (bank notes and coins)? Today's denominations in CHF are 0.05, 0.10, 0.20, 0.50, 1, 2, 5 (coins), 10, 20, 50, 100, 200, 1000 (banknotes). The amount of CHF 0.20, for example, can be owned in four ways (to get integers, let's switch to centimes): (20), (10,10), (10,5,5), (5,5,5,5). The amount of CHF 0.04 can be owned in no way, while there is exactly one way to own CHF 0.00 (you cannot have 4 centimes in your wallet, but you *can* have no money at all in your wallet).

Task: Solve the problem for a given input amount, by writing the following function (all values to be understood as centimes):

```
// PRE: 0 <= end <= v.size(), and
//      0 < denominations[0] < denominations[1] < .. denominators
//      describes a (potentially empty)
//      sequence of denominations
// POST: return value is the number of ways to partition amount
```

```
//       using denominations from denominations[0], ..., denomina
unsigned int partitions (unsigned int amount,
                         const std::vector<unsigned int> & denomi
                         unsigned int end)
```

Rule: Use recursion to solve this problem, non-recursive solutions are considered incorrect.

To allow you to focus on implementing the recursive function `partitions`, we provide you with the template code that implements all functionality except the said function, which should be implemented in file `partitions.cpp`. The input is the amount in centimes as `unsigned int` and the expected output is the number of ways that amount can be owned as `unsigned int`.

Task 3: prefix to infix notation

Open Task

Task

Prefix notation is a way to write expressions so that an operator appears before its operands. For example, in prefix notation the expression `+ a b` means the sum of a and b, written as `a + b` in the more usual infix notation. Similarly, `* + a b - c / d e` in prefix notation corresponds to `( a + b ) * ( c - d / e )` in infix notation.

An interesting property of prefix notation is that there is no ambiguity as to the order in which operators are applied. In particular, parentheses are never required.

Write a program that read an expression from input and convert it to infix notation, using as few parentheses as possible. The allowed operators are +, -, * and / with their usual precedences and associativity (which are the same as in C++). The constants can be either integer or variables.

Note that it is allowed -- and it must be done when this eliminates parentheses -- to re-order the application of operators when such re-ordering is valid in general. For example, both prefix expressions `+ a - b c` (`a + ( b - c )` in infix notation with explicit parentheses) and `+ a b - c` (`( a + b ) - c`) should be converted to `a + b - c`.

However, the order of constants should not be changed, neither should any form of expression simplification be performed. In particular, `+ a b` should not be converted to `b + a`. Similarly, `+ 0 0` should not be simplified to `0`.

Hint: as the number of different concrete cases is big (see the tests), it is recommended to write functions that treat operators and relative precedences as

uniformly as possible. For this, you may use the following remark: a sub-expression e need to be parenthesized iff it is an expression shaped as $e_1\ Op\ e_2$ with Op being an operator of unsufficient precedence to appear without parentheses at e's location. For example, in the expression `a * ( b + c )`, `b + c` need to be parenthesized because `+` does not have high enough precedence to occur as the right sub-expression of `*`. However, `*` or `/` would have high enough precedence.

Note: you can use functions `lookahead` and `consume` from the lecture to parse the input.

Input

An expression in prefix notation. Such expressions are given as a sequence of operators among `+`, `-`, `*`, `/` and constants, separated by whitespaces. Constants can either be non-negative integers given without leading zeroes, or variable names. Variable names are non-empty sequence of alphabetic characters (`'a'-'z'`, `'A'-'Z'`).

Example:

```
* + a b - c / d e
```

Output

The same expression, in infix notation, with as few parentheses as possible. There should be exactly one whitespace between each operator/parenthesis/constants.

Example:

```
( a + b ) * ( c - d / e)
```