theory questions

fredrik robertsen

2025-09-18

task 1) mdc

i answered the next section first (task 2), but this is essentially the same process. using an auxiliary function, we can interpret the tokens sequentially from left-to-right by adding numbers to the stack (which is passed down the recursion as an argument to the auxiliary function, so that we can access the elements on the stack at arbitrary recursion depth) and popping some upon discovering a non-number. once we have popped some numbers we can simply perform the operation that the token corresponds to and push the result onto the stack.

task 2) postfix conversion

the high-level overview of the algorithm is simply that you have a stack that stores the current numbers you've encountered when reading the tokens left-to-right. then you consume the top two elements of the stack when you find a binary operator and push the resulting expression. doing so recursively creates a tree of the infix operations.

this is handy, because postfix notation has no ambiguity when it comes to operator precedence, and as such can be useful for calculators with a simple lex/parse/interpret implementation. going the other way, from an infix expression to a postfix one is more challenging, due to the operator precedences.

this algorithm is essentially the shunting-yard algorithm, one you'll quickly meet if you do any amount of parsing, such as if you implement a calculator of your own. https://en.wikipedia.org/wiki/Shunting_yard_algorithm

task 3) theory

a)

we can define a formal grammar that specifies the accepted *lexemes* in our mdc implementation:

```
let G = (V, S, R, v), where
    V = {e},  % e is the empty string
    S = {n | n in N} union {+, -, *, /, p, d, c, i},
    R = {(e, s) | s in S},
    v = e
```

b)

using extended backus-naur form, we can express the grammar of ExpressionTree like so:

```
<operator> ::= plus | minus | multiply | divide
<expression> ::= number
               | <operator>(<expression> <expression>)
```

the program outputs an `<expression>`, which is a nested non-terminal formatted with parentheses to denote the order of operations, i.e. depth of nesting.

c)

the grammar of a) is regular because all we do is of the form `<e> ::= s` (map to symbol), which is one of the three valid rules for a regular grammar, the other two being `<v> ::= empty` (map to empty) and `<v> ::= <a> s` (map symbol to right of variable).

the grammar of b) is context-free because we freely string non-terminals together without requiring any information about the context of the non-terminal we are expanding. i.e. all our rules are of the form

`<v> ::= str, where str is any sequence of variables and symbols`