MAU11602
Lecture 1
2026-01-22

Language for arithmetic

Recap: We have a language for primitive arithmetic, but it's kind of awful.
- Complicated grammar
- Difficult to parse
- Technically has referential transparency, but not in a nice way.

By this I mean we can replace any subexpression with its value without changing the value of the larger expression, but not every substring which is an expression is a subexpression.

Good news: There are at least three different ways to fix all these problems!

Bad news: No one cares enough to put in the effort.

Mandatory parentheses

With infix notation we need to allow parentheses, but what if we require them?
So instead of `1 - 2 * 3 + 4` we write `((1 - (2 * 3)) + 4)`. Instead of `(1 - 2) * (3 + 4)` we write `((1 - 2) * (3 + 4))`.
The grammar is now much simpler, with

```
expr ::= "(" expr ws oper ws expr ")" | int
oper ::= "*" | "+" | "-"
```

instead of

```
expr0 ::= expr0 ws addop ws expr1 | expr1
expr1 ::= expr1 ws mulop ws expr2 | expr2
expr2 ::= "(" expr0 ")" | int
addop ::= "+" | "-"
mulop ::= "*"
```

It's also easier to see where substitution is allowed. We can substitute `7` for `3 + 4` in `((1 - 2) * (3 + 4)` but not in `((1 - (2 * 3)) + 4)`.
Any substring which is a valid expression is a subexpression.

Parsing with mandatory parentheses

Every expression is either an integer or an operator between two subexpressions, all delimited by parentheses.
There may be other operators inside the subexpressions, but those are within further parentheses.
Keep a running count of (’s minus)’s. This number is always positive inside and 0 at the end. To know whether an operator is the main operator or belongs to a subexpression, just check whether it’s 1 or greater.

```
( ( 1 - ( 2 * 3 ) ) + 4 )
                    ^
1 2 2 2 3 3 3 3 2 1 1 1 0
```

so the operator is `+` and the left and right subexpressions are `( 1 - ( 2 * 3 ) )` and `4`.
This also shows the the grammar is unambiguous.
The only disadvantage of mandatory parentheses is that you have write a few more parentheses.
But if you don’t like parentheses I have good news for you!

Prefix notation

Question: How would you read `(1 - 2) * (3 + 4)` out loud?
You could say "open parentheses 1 minus 2 close parentheses times open parentheses 3 plus 4 close parentheses",
or, better, "the product of the difference of 1 and 2 and the sum of 3 and 4".
Why not just write `* - 1 2 + 3 4`?
Prefix notation was developed by Jan Lukasiewicz in 1924.
It's often called Polish notation.
No parentheses are needed. `(( 1 - (2 * 3)) + 4)` is written as `+ - 1 * 2 3 4`.
More generally, all five ways of parenthesising `1 - 2 * 3 + 4` have distinct representations in prefix notation.
The grammar is also simple:

```
expr ::= oper ws expr ws expr | int
oper ::= "*" | "+" | "-"
```

Parsing is easy as well. We use a running count again. We start it at 1, increment it when we see an operator and decrement it when we see an integer.

Parsing prefix notation

Parsing is easy as well. We use a running count again. We start it at 1, increment it when we see an operator and decrement it when we see an integer.
The count is zero at the end of the expression. If the expression is not an integer then the first subexpression ends when the count first hits 1.

```
  * - 1 2 + 3 4       + - 1 * 2 3 4
        ^                     ^
1 2 3 2 1 2 1 0     1 2 3 2 3 2 1 0
```

`* - 1 2 + 3 4` is the product of `- 1 2` and `+ 3 4`, while `+ - 1 * 2 3 4` is the sum of `- 1 * 2 3` and `4`.
This has both the advantages of mandatory parentheses:
- Any substring which is an expression is a subexpression, so substitution is easy to understand.
- The grammar is simple and easy to parse mechanically.

In addition:
- No parentheses are needed.
- It can be read naturally.

Postfix notation

We could also write the operator after the operands, like `1 2 3 * - 4 +` or `1 2 - 3 4 + *`.

Note that the operators are written in the order we perform the operations in!

The grammar is also simple:

```
expr ::= expr ws expr ws oper | int
oper ::= "*" | "+" | "-"
```

To parse, keep a running count of numbers minus operators.

Initially it's 0. Thereafter it's always positive. At the end it's 1.

The expression is an integer or ends in an operator. If so, the first subexpression ends the last time the count hits 1 before the final operator.

```
  1 2 3 * - 4 +       1 2 - 3 4 + *
          ^                   ^
0 1 2 3 2 1 2 1     0 1 2 1 2 3 2 1
```

We don't even need to parse in order to evaluate though!

Stacks

A stack is a data structure. The stack is either empty, or consists of a head and a tail, which is also a stack. Pushing an item onto a stack creates a new stack with that item as head and the old stack as tail. Popping an item off a non-empty stack gives you the head of that stack and a new stack, the tail of the old stack.

A stack based evaluator for postfix expressions starts with an empty stack of integers. We read integers and operators from input, one at a time. If we read an integer we push it onto the stack. If we read an operator we pop two integers off the stack, apply the operator, and then push the result back onto the stack. At the end of input there should be only one item on the stack, the value of the expression.

```
1  2  3  *  -  4  +        1  2  -  3  4  +  *

1  2  3  6 -5  4 -1        1  2 -1  3  4  7 -7
   1  2  1    -5              1    -1  3 -1
      1                                -1
```

Stacks and prefix notation

We can also implement a prefix operator with a stack of integers and operators.
Again, start with an empty stack and read integers or operators one at a time.
If we read an operator, push it onto the stack.
If we read an integer, pop the top item from the stack.
If it was an operator, push it back, then push the integer we read.
If it was an integer, pop then next item. It should be an operator. Apply it to the number we popped and the one we read, then act as if we had just read that number.
At the end of input there will be one item on the stack, the value of the expression.

```
+  -  1  *  2  3     4        *  -  1  2  +  3  4

+  -  1  *  2  6 -5 -1        *  -  1 -1  +  3  7 -7
   +  -  1  *  1  +              *  -  * -1  + -1
      +  -  1  -                    *     * -1  *
         +  -  +                                *
            +
```

Lists and stacks

A list is a data structure. The list is either empty, or consists of a head and a tail, which is also a list. Pushing an item onto a list creates a new list with that item as head and the old list as tail. Popping an item off a non-empty list gives you the head of that list and a new list, the tail of the old list.

This should look familiar.

Lists are stacks.

For some reason people use different terminology though. The head of a stack is usually called the top. The tail doesn't have a name. Pushing onto a list is usually called consing. Popping off a list doesn't have a name but for bizarre historical reasons the operation is described in terms of two other operations, called car and cdr.

It's important to recognise stupidity and adapt to it, but not submit to it.

You should know that stacks and lists are the same thing, but also know that most people don't know that, and use terminology they understand.

You should understand why prefix, postfix and fully parenthesised infix are all better than infix with precedence and associativity rules, but also accept that you have to communicate with other people, who don't use the better alternatives.