MAU11602
Lecture 0
2025-01-21

Module information

MAU11602, Computation Theory and Logic
http://www.maths.tcd.ie/~stalker/11602/
John Stalker (stalker@maths.tcd.ie)
http://www.maths.tcd.ie/~stalker
30% (approximately) weekly assignments
70% two hour exam

Topics

- (Formal) languages
- Computability
- Logic

We need formal languages to describe computations, formulate precise statements to prove or disprove.
We need computability to understand language processing, provability.
We need logic to reason about languages, computations.
It's turtles all the way down!
In some sense these are all the same subject anyway!
For example, types correspond to logical propositions, and expressions of a given type correspond to proofs of the corresponding proposition (Curry-Howard).

A simple language for integer arithmetic

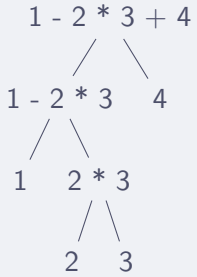
We start simple, with integers, no variables, and three operators: `+`, `*` and `-`
There's a lot missing, but we'll add more later.
Question: What is the value of the expression `1 - 2 * 3 + 4`?
Answer: `-1`, but why?
`2 * 3` reduces to `6` so `1 - 2 * 3 + 4` reduces to `1 - 6 + 4`.
`1 - 6` reduces to `-5` so `1 - 6 + 4` reduces to `-5 + 4`.
`-5 + 4` reduces to `-1`.
`-1` is a value, so no further reduction is possible.
We're using referential transparency: If a subexpression evaluates to a value then replacing that subexpression with that value in a larger expression doesn't change its value.
As we add more to our language we want to preserve referential transparency.
But wait! `1 - 2` reduces to `-1`, so does `1 - 2 * 3 + 4` reduce to `-1 * 3 + 4`?

Precedence and associativity

In evaluating `1 - 2 * 3 + 4` we need to perform the multiplication first, then the subtraction, then the addition. Any other order will give the wrong answer.
The reason we do multiplication first is precedence. The reason we do the subtraction before the addition is associativity.
Multiplication has higher precedence than addition or subtraction, and operators of the same precedence are left associative, i.e. we apply them from left to right.
What does wrong mean? It means not the one everyone has agreed to call the right answer.
What about referential transparency? Is that okay?
Yes, but we need to arrange that `2 * 3` is a subexpression of the expression `1 - 2 * 3 + 4`, but `1 - 2` is not.
But what if we actually want to perform the subtraction first?
Then we should write `(1 - 2) * 3 + 4` instead of `1 - 2 * 3 + 4`, but our language doesn't have parentheses, so we need to add them.

Trees and subexpressions

We can represent the structure of subexpressions of `1 - 2 * 3 + 4` using a tree.

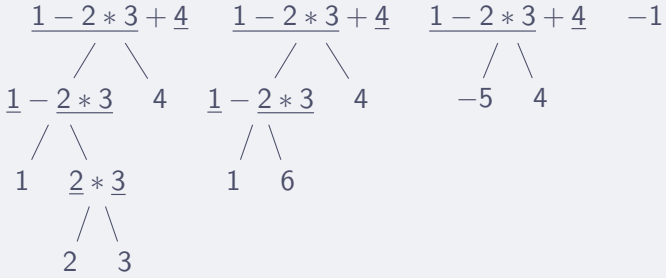


The terminology for trees is a weird mixture of botany (root, branches, leaves, etc.), geneology (parent, children, siblings, etc.), and completely random stuff (nodes, traversal, etc.).

The full expression `1 - 2 * 3 + 4` is at the root–which is at the top of the tree!–and the nodes show the subexpressions. The expression `1 - 2` is not at any node and so is not a subexpression.

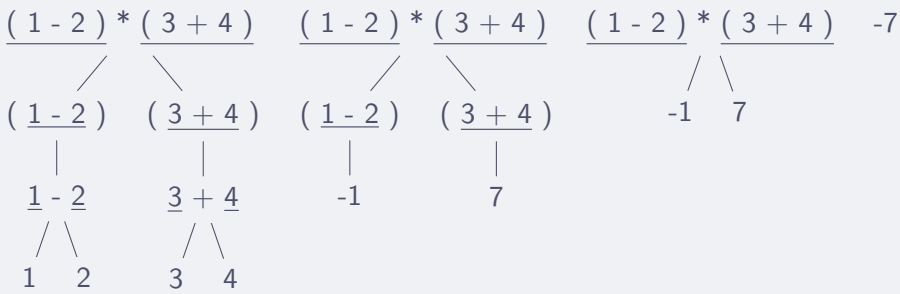
Trees and evaluation

The leaves of the parse tree are values, expressions which can't be reduced further. We work our way upwards from the leaves to the root. An expression can be evaluated in one step once the values of its children are known.



Parentheses

As discussed earlier, parentheses change the order of evaluation.



This time 1 − 2 *is* a subexpression.

A grammar for (infix) arithmetic

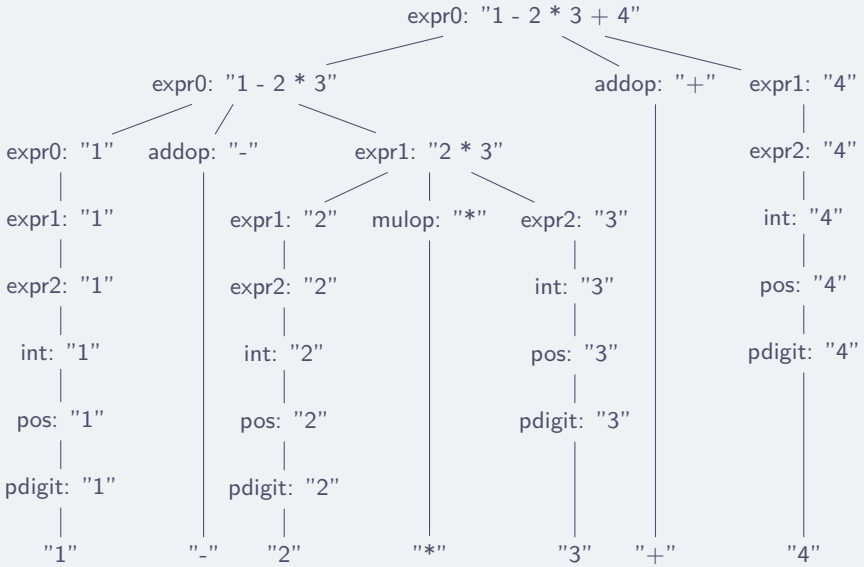
We can express the grammar of our language as follows:

```
 expr0 ::= expr0 ws addop ws expr1 | expr1
 expr1 ::= expr1 ws mulop ws expr2 | expr2
 expr2 ::= "(" expr0 ")" | int
 addop ::= "+" | "-"
 mulop ::= "*"
   int ::= "0" | pos | "-" pos
   pos ::= pdigit | pos digit
 digit ::= "0" | pdigit
pdigit ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
    ws ::= " " | ws " "
```

This is a (mostly) standard way of expressing grammars. The initial line means that an `expr0` is either an `expr0`, then whitespace, then an `addop`, then more whitespace, and finally an `expr1`, or it's a an `expr1`. Two lines later we see that an `expr2` is either a `(` then an `expr0` and then a `)`, or it's an `int`.

Note that these definitions are recursive!

A parse tree


expr0: "1 - 2 * 3 + 4"
expr0: "1 - 2 * 3"
addop: "+"
expr1: "4"
expr0: "1"
addop: "-"
expr1: "2 * 3"
expr2: "4"
expr1: "1"
expr1: "2"
mulop: "*"
expr2: "3"
int: "4"
expr2: "1"
expr2: "2"
int: "3"
pos: "4"
int: "1"
int: "2"
pos: "3"
pdigit: "4"
pos: "1"
pos: "2"
pdigit: "3"
pdigit: "1"
pdigit: "2"
"1"
"-"
"2"
"*"
"3"
"+"
"4"

More about trees

Given an epxression tree we can check, mechanically, that the expression tree conforms to the grammar.
We can construct the simpler expression tree we saw earlier mechanically from the full parse tree.
Once we have that we can evaluate the expression.
Question: How do we construct the parse tree from the grammar and the string?
Question: Can we be sure there's only one possible parse tree?
Question: This seems needlessly complicated. Can we avoid all this work?
These are good questions, not all of which we will answer in this module.