MAU11602
Lecture 2
2026-01-22

Whitespace

Whitespace is spaces, tabs, new lines, etc.
Most formal languages allow whitespace, e.g. spaces, tabs, new lines, etc.
Normally it should be forbidden in certain contexts, e.g. inside an integer.
Normally it's also mandatory in some contexts, e.g. between adjacent integers.
In other contexts it's optional.
In some languages, e.g. Python and Haskell, Wisp, the type of whitespace in an expression can affect its value.
It's possible to use whitespace to indicate the order of operations.

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

This is an accessibility nightmare!

Infix and prefix in real computer languages

Most computer languages have a fixed grammar for arithmetic expressions. Some, e.g. SML, Haskell, Racket have a default but allow you to modify it.
Most use infix with precedence and associativity rules, so some parentheses are necessary but not fully parenthesised.
Some, mostly Lisps like Racket, use prefix.
(Most) Lisps have mandatory parentheses, even though prefix languages shouldn't need them, i.e. `(* (- 1 2) (+ 3 4))` and `(+ (- 1 (* 2 3)) 4)`.
Why?
- Gives easier access to parse tree
- Allows parsing even when the arity of operators is unknown or variable, e.g. `(+ 1 2 3)` is a valid Lips expression, with value 6.

Wisp is a Lisp which uses significant whitespace instead of parentheses. The following is `(* (- 1 2) (+ 3 4))` in Wisp:

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

Postfix in real computer languages

Some languages, e.g. Forth, PostScript, use postfix notation, without parentheses.
The Post in PostScript actually refers to postfix.
The infix expression `(1 - 2) * (3 + 4)` corresponds to the expression
`1 2 - 3 4 + *` in Forth and the expression `1 2 sub 3 4 add mul` in PostScript.
Evaluating postfix seems easier than prefix or especially infix, but we'll see later there is a problem.

Implementation details

Parsing is usually done in two stages. The first, called lexing, divides the input up into a list of tokens. Often we have the parser strip off whitespace as well.
Tokens have a type and value. There are finitely many types, but may be infinitely many values.
In this case the tokens would be integers, operators and possibly parentheses.
Depending on the grammar we might need to distinguish additive and multiplicative operators.
The parser can generate a tree and hand it to a separate evaluator, or it can evaluate as it goes, or something in between.
Writing parsers by hand is tedious and error-prone. We can use a parser generator, which reads a formal grammar and produces a parser for that language.
The language I've used for grammars is the one used by the most widely used parser generators.
There are even lexer generators. They tend to use regular expressions to describe tokens.

Booleans, relations

We'd like to introduce relations, like $<$, $\leq$,$=$, etc. into our language, so we can write things like `2 + 2 = 4`, which should evaluate to `true`.
First we need to modify the grammar(s) .
For postfix and prefix this is easy. Just change the definition of `oper` to
`oper ::= "*" | "+" | "-" | "<" | "<=" | "=" | ">" | ">="`
There's a problem though. Meaningless expressions like the postfix `1 2 < 3 4 = +` become possible.
This should step to `true 3 4 = +`, because $1 < 2$ is true, then `true false +`, since $3 = 4$ is false, but then `true false +` can't be evaluated.
Different languages deal with this in different ways.
- Treat `true` and `false` as `0` and `1`, or vice versa.
- Add a special error value.
- Modify the grammar to have two types of expression, and make this a syntax error.
- Keep the same grammar but introduce a notion of types. Check the types, either before or during evaluation.

Adding relations to the grammar(s)

We could change the postfix grammar to

```
expr ::= bexp | iexp
iexp ::= iexp ws iexp ws oper | int
bexp ::= iexp ws iexp ws rel
oper ::= "*" | "+" | "-"
 rel ::= "<" | "<=" | "=" | ">" | ">="
```

Now `1 2 < 3 4 = +` can't be parsed.

Prefix is similar. Infix is worse.

This works, but

- As we add more types the grammar becomes awful.
- We need the change the grammar every time we define a function.
- We can only have finitely many types.
- Are type errors really syntax errors?

Better to keep the grammar simple and introduce types, and type rules.