Parse Trees

Beefed-up calculator language

run → *stmt run* | *stmt*
stmt → *ares* `STOP`
ares → `VAR ASN` *bres* | *bres*
bres → *bres* `BOP` *res* | *res*
res → *res* `COMP` *exp* | *exp*
exp → *exp* `OPA` *term* | *term*
term → *term* `OPM` *factor* | *factor*
factor → `NUM` | `VAR` | `LP` *bres* `RP`

Download today's tarball and run `make` to get a parse tree for some string in this language.

We notice that the parse tree is large and unwieldy with many unnecessary nodes.

Abstract Syntax Tree

Consider the program `x := (5 + 3) * 2; x - 7;`.
What should the AST for this look like?

AST Properties

Remember, *ASTs are not about the syntax!*
They *remove* syntactic details from the program, leaving only the semantics.

Typically, we show ordering (e.g. of *ares*'s in the previous example) by nesting: the last child of a statement is the next statement, or null.

Are ASTs language independent?

Static type checking

Consider the string `(7 > 2) + 3;`. This is an error.
But where should this error be identified?

Each node in the AST has a type, possibly "void".

Static type checking with variables

What about the string `x = 6 > 3; x * 12;`?

We have to know the *type* of the variable x.
Otherwise, there is no way to detect this error at compile-time.

Only *statically-typed languages* allow this sort of checking.
Remember, in this class *errors are a good thing!*

Unit outcomes

You should know:

- What an AST is, and why we need them.
- The relationship between language, parse tree, and AST.
- How static type-checking works, at a basic level.

You should be able to:

- Draw a parse tree for a given string, given the grammar.
- Determine the AST from the parse tree. Note that there is some flexibility here!