

CSC-363 Lecture 02C

Introduction to Parsing

Colin McKinney

30 January 2026

Studio Follow-up

- ▶ CharStream vs. TokenStream: CharStream holds the characters of the *source code* and has methods to give us characters from that source. We use it to build a TokenStream, which is a sequence of tokens.
- ▶ TokenStream is just a list, not a linked list.
- ▶ Enum allows us to do checks like `if tok.TokenType == TokenType.PLUS` rather than `if tok.lexeme == '+'`.
- ▶ Parsing being confusing: understandable! We're getting into it today and beyond. For studio, the idea was that just because source code tokenizes correctly, doesn't mean it's a valid program.
- ▶ Forbidden chars checking: this will sink in as you get into the code. Choose a forbidden char and trace through the code to think about where exactly you'd catch it (there are three answers).
- ▶ How parts fit together: like forbidden chars, do a trace of `test0.ac`. Where does the source go, etc.?

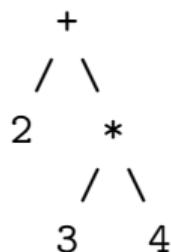
Motivating Examples

Example 1

Consider $2 + 3 * 4$.

Why isn't this $(2 + 3) * 4$? Where does that "rule" come from or live?

AST:

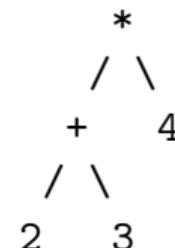


Example 2

Consider $(2 + 3) * 4$.

Why isn't this $2 + 3 * 4$?

AST:



Rules: grammar of the language, and implemented in the parser's logic.

Input (ac)

8 9 +

Tokenizer output

INTLIT(8)

INTLIT(9)

PLUS

EOF

The tokenizer is satisfied. The parser is not.

Where parsing fails

A binary operator needs two subexpressions:

+
/ \
? ?

But in infix order we see:

8 9 +
expr expr operator (too late)

There is no grammar rule that turns
expr expr into a single expr.

What Problem Does the Parser Solve?

- ▶ Tokens arrive in TokenStream, hence, in a linear order.
- ▶ ASTs are not linear, they are hierarchical.
- ▶ The parser must decide:
 - ▶ When a given token becomes a node
 - ▶ Where/how that node attaches to the tree

Operator precedence rules and associativity are rules about the tree shape, not about evaluation.

Hence, parsing needs to take a linear stream of tokens and turn it into a tree, while respecting the precedence and grouping rules of the language.

What Should Parsing Do? What Shouldn't It?

- ▶ As before, take a linear TokenStream and produce a tree that respects the grammar of the language (operator precedence, grouping rules).
- ▶ It doesn't produce numbers or dc instructions. The dc instructions will be produced in the next phase (code generation), which takes the AST as input.
- ▶ It doesn't check overall structural validity: for example, forgetting to declare an integer but then using it somewhere. We could do this before parsing or after.
- ▶ ac instructions, from subsets of the TokenStream, each form statements. Each statement will be parsed to a AST.
- ▶ A program's AST is then a collection of ASTs for each statement within the program. We could think of the root as being the program as a whole, and then each statement as being the first layer of nodes. Or we could think of the program as being a list of trees in order.

What's an AST?

An AST is precisely that, a tree. The leaf nodes of trees in ac will be:

- ▶ Integer literals
- ▶ Variable references
- ▶ Single token instructions like intdcl and print (technically also root of their trees)

The non-leaf nodes will be binary operations with two children (left and right). What binary operations do we have?

- ▶ Arithmetic: + - * / ^
- ▶ Assignment: =

Does anything seem missing?

Parsing Methods

There are several methods for accomplishing the parsing task:

- ▶ Operator-stack methods (we'll use this for ac)
- ▶ Top-down parsers: LL and recursive descent.
- ▶ Bottom-up: LR/shift-reduce

Discussing top-down and bottom-up parsers and theory will require us to talk about **context-free grammars**, which are one step up from regular languages in the Chomsky hierarchy.

The process of building a parser can be done manually (we'll do this for ac) or using parser-generators (we'll use ANTLR for microC. ANTLR derives from work at Purdue!)