Parsing: the beauty of LR

One of the important steps of any **compiler** is the parsing phase. After the lexical analyzer tokenizes an input string, those tokens are checked (parsed) during syntactic analysis, which verifies the tokens are arranged in meaningful orders to create a valid expression. Lexers rely on regular expressions for tokenization, and parsers rely on grammars for syntax checking. The goal of the parser is to output a parse tree for the input, more specifically, an abstract syntax tree (AST); which then gets analyzed further (semantic analysis and so on, up to code generation of the target language).

With context-free languages, and grammars, **ambiguity is our enemy**. Ambiguity results in different parse trees for inputs, and therefore, different ASTs and thus potentially different interpretations. What that means is, depending on the parse, our programs could yield different results... clearly not good!

To address this, we have some strategies: **LL parsers** attempt to address the pitfalls of ambiguous grammars, which are: precedence, associativity, left recursion and left factorization. These are addressed by heuristic rewriting-rules that transform an ambiguous grammar into one that isn't, for example:

ambiguous:
$\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E} | \mathrm{E} + \mathrm{E} | (\mathrm{E}) | \mathrm{int}$

not-ambiguous:
$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T} | \mathrm{T}$
$\mathrm{T} \rightarrow \mathrm{T} + \mathrm{F} | \mathrm{F}$
$\mathrm{F} \rightarrow (\mathrm{E}) | \mathrm{int}$

What that means is, when developing a new language, you have to make sure your grammar is not-ambiguous, by identifying ambiguous regions and fixing them. This is not easy to do. In fact, this cannot always be done! Some grammars are inherently ambiguous. Thus, the kinds of languages we can express in this way is constraining and limiting.

Another approach, via **LR parsers**, alleviates some of these frustrations. While grammars for these are still less powerful (expressive) than pure CFLs, they are less cumbersome to work with than those for LL parsers, and more powerful. Making grammars LR-compliant is easier to do, and there is a simple way to check that they are LR-compliant (the DK test), which makes them easier to fix as well.

The first "L" in these two techniques (LL vs LR) refer to "left-to-right" and the second L/R refer to left/right-most derivations. In an LR parser, the trick is to scan left-to-right to identify **valid handles**, and to apply right-most derivations (in reverse) in what are called left-most **reductions**. In processing an input string, we look for **reductions** $\twoheadrightarrow$ (which are rightmost derivations, $\mathrm{T} \rightarrow \mathrm{h}$, in reverse) in which the handle (the prefix xh of the string xhy) uniquely determines the next rule that must be applied:

$$u_i = \overbrace{x_1 \cdots x_j}^{x} \overbrace{h_1 \cdots h_k}^{h} \overbrace{y_1 \cdots y_l}^{y} \rightarrowtail \overbrace{x_1 \cdots x_j}^{x} \overbrace{T}^{T} \overbrace{y_1 \cdots y_l}^{y} = u_{i+1}.$$

Long story short, when every valid string for a grammar has a forced handle, it defines the class of **deterministic context-free grammars**, which of course, corresponds to the **deterministic context-free languages**, and **deterministic pushdown automata (DPDA)**. Thus, LR parsers have a beautiful and very simple mathematical basis: we simply avoid nondeterminism and voila, ambiguity gone and parsing easy. Defining these languages is as we would normally expect, like DFA vs NFA, exactly one transition is allowed; however, we do allow ϵ transitions, so it gets slightly messy to account for ϵ jumps, but the intuition is the same.

And that is why LR parsers are a beautiful thing!

Of course do realize that, while NFAs and DFAs are equivalent in power, (N)PDAs and DPDAs, are not! DPDAs are less powerful, that is, they generate a smaller class of languages than NPDAs, i.e., a subset of the CFLs.

See post @112 for some videos of the nuts-n-bolts of LR parsing.