

# Lexing & Parsing

CS153: Compilers

## Notes

- Homework due Friday before class.
  - See Gregory or me if you need help!
  - Don't wait until last minute.
- Reading:
  - Relevant chapters on Lexing & Parsing in Appel
  - MLLex and MLYacc documentation
  - “Monadic Parsing in Haskell” by G.Hutton and E.Meijer.

## Parsing

- Two pieces conceptually:
  - Recognizing syntactically valid phrases.
  - Extracting semantic content from the syntax.
    - E.g., What is the subject of the sentence?
    - E.g., What is the verb phrase?
    - E.g., Is the syntax ambiguous? If so, which meaning do we take?
      - “Fruit flies like a banana”
      - “2 \* 3 + 4”
      - “x ^ f y”
- In practice, solve both problems at the same time.

## Specifying Syntax

We use grammars to specify the syntax of a language.

exp → int | var | exp '+' exp | exp '\*' exp |  
       'let' var '=' exp 'in' exp 'end'

int → '-'?digit+

var → alpha(alpha|digit)\*

digit → '0' | '1' | '2' | '3' | '4' | ... | '9'

alpha → [a-zA-Z]

## Naïve Matching

To see if a sentence is legal, start with the first non-terminal), and keep expanding non-terminals until you can match against the sentence.

$N \rightarrow 'a' \mid ((' N ')$  “((a))”  
 $N \rightarrow ((' N ')$   
 $\rightarrow ((' (' N ') ')$   
 $\rightarrow ((' (' 'a' ') ')) = “((a))”$

## Alternatively

Start with the sentence, and replace phrases with corresponding non-terminals, repeating until you derive the start non-terminal.

$N \rightarrow 'a' \mid ((' N ')$  “((a))”  
 $(((' 'a' ') ')) \rightarrow ((' (' N ') ')$   
 $\rightarrow ((' N ')$   
 $\rightarrow N$

## Highly Non-Deterministic

- For real grammars, automating this non-deterministic search is non-trivial.
  - As we'll see, naïve implementations must do a lot of back-tracking in the search.
- Ideally, given a grammar, we would like an efficient, deterministic algorithm to see if a string matches it.
  - There is a very general cubic time algorithm.
  - Only linguists use it ☺.
  - (In part, we don't because recognition is only half the problem.)
- Certain classes of grammars have much more efficient implementations.
  - Essentially linear time with constant state (DFAs).
  - Or linear time with stack-like state (Pushdown Automata).

## Tools in your Toolbox

- Manual parsing (say, recursive descent).
  - Tedious, error prone, hard to maintain.
  - But fast & good error messages.
- Parsing combinators
  - Encode grammars as (higher-order) functions.
    - Basically, functions that generate recursive-descent parsers.
    - Makes it easy to write & maintain grammars.
  - But can do a lot of back-tracking, and requires a limited form of grammar (e.g., no left-recursion.)
- Lex and Yacc
  - Domain-Specific-Languages that generate very efficient, table-driven parsers for general classes of grammars.
  - Learn about the theory in 121
  - Need to know a bit here to understand how to effectively use these tools.