**ApusApus: A GLL parser generator for Swift[[1]](#footnote-2)**

The purpose of this write-up is to describe the design of a ***G****eneralized* ***L****eft-to-right,* ***L****eftmost* derivation parser that has the following properties:

* Fastest possible processing for those parts of the syntax that are LL1
* Easiest to read source code, matching the input syntax
* Supports left-recursion
* Effectively unlimited ‘lookahead’
* Parses Unicode UTF-8 encoded files

The input syntax will accept EBNF notation in 2 flavors:

(...) [...] {...} <...>  
(...) (...)? (...)\* (...)+

Production rules come in normal and silent form:

color = “restyled” | “green” | “blue” .  
skip : ‘[\s]+’ .

The silent form can be used to define whitespace and comment, which are often not considered in parsers.

String literals are enclosed in double quotes. Regular expressions are enclosed in single quotes. The regular expression syntax is that used by Swift in the sense that what is enclosed in single quotes ‘...’ in an ApusApus syntax would be written as #”...”# in Swift source code. To represent a single quote in a regular expression it needs to be escaped as \’

The syntax can be annotated with actions that are copied verbatim to the generated parser as in @print(“Hello world!”)@. The @ character itself can be escaped as \@.

Important: exchange EBNF and Swift primacy by SwiftUI-like DSL.

The parser generator reads a *target-language* syntax from a file. That target-language syntax is written in *apus-syntax*. The parser generator produces a parser for the *target-language*. That parser can process files written in the *target-language* syntax.

The *target-language* syntax is parsed with a hand-written LL1 parser. This parser builds an abstract syntax tree (AST). This AST is then used to generate a parser for the *target-language*.

The AST node types directly implement the EBNF elements SEQ, ALT, OPT, REP, NTR, and TRM.

The handwritten *apus-syntax* parser used a fixed list of token types: name, regex, action, literal, |, {, }, (, ) etc.

The handwritten scanner produces tokens from the *target-language* syntax that are a tuple (kind: String, image: String). The kind is one of the above token types.

grammar = { name ( silent | verbose ) "." } .  
silent = ":" ( regex | literal ) .  
verbose = "=" selection .  
selection = sequence { "|" sequence } .  
sequence = ( term )+ .  
term = terminal  
 | "[" selection "]"  
 | "{" selection "}"  
 | "<" selection ">"  
 | "(" selection  
 ( ")"  
 | ")?"  
 | ")\*"  
 | ")+"  
 ) .  
terminal = name | literal | regex | action .  
  
name = '[\_a-zA-Z0-9]+' .  
regex = '(\\\'|[^\']+?)\*\'' .  
action = '@(\\@|[^@]+?)\*@' .  
literal = '\"(\\\"|[^\"]+?)\*\"' .  
  
whitespace : '\s+' .  
singleLine : '//.\*' .  
multiLine : '(?s)(/\\*).\*?(\\*/)' .

1. <https://supermemo.guru/wiki/SuperMemo_Algorithm:_30-year-long_labor> [↑](#footnote-ref-2)