# Assignment 3 : A Swift Backend for BNFC


### By Armaan Rashid and Filbert Phang


# Part I

## Introduction:

For our project, we have chosen the Haskell package BNFC (Backus-Naur Form Converter) as the foundation for our work. BNFC is a well-known tool for generating parsers in Haskell, C, Java and other languages from files written in pure EBNF (extended Backus-Naur form). Our goal is to add a backend compiling EBNF files for use in Swift's extremely fast Lotsawa parser, which accepts all LR and LL grammars and is based on the Earley parsing algorithm.

## Learning Goals

BNFC operates on EBNF files, which is a DSL for specifying language grammars. It has been actively maintained for almost 20 years and has over that time been extended with backends into many functional and imperative languages like C, C++, Java, OCaml and respective parsers in those languages. We propose adding to this list by creating a Swift backend using the Lotsawa parser. Our implementation uses BNFC to compile the raw EBNF files into a format compatible with Swift and Lotsawa.

Lotsawa is not just another backend for BNFC, however: all the current backends, including the Haskell backend with Happy, rely on recursive descent parsing. Lotsawa, however, is an Earley parser boosted with smaller iterative improvements that, for any totally deterministic grammar, operates in linear time, and works for even ambiguous grammars by returning multiple parses.

In addition to leveraging BNFC, we plan to extend its functionality by implementing a new backend using the Lotsawa parser in Swift. The Lotsawa parser is a powerful parsing library for Swift, which provides efficient and flexible parsing capabilities.

Because of time constraints, we only intend to implement the backend for a subset of EBNF which is normally accepted by BNFC. In particular we are only aiming to support EBNF files which consist of standard '::=' rules and the separator and terminator pragmas.

We also plan to link to the specification learning goal for this course by implementing QuickCheck tests to ensure this cross-compilation works as intended.

# Part II

## Description of Library Interface

For our project, we extended BNFC with the new `BNFC.Backend.Swift` module, which exposes the lone function `makeSwift`. `makeSwift` converts a Context-Free Grammar (CFG) in BNFC to fully-functioning parser code in Swift using the Citron Lexer and Lotsawa Parser.

`BNFC.Backend.Swift` contains 2 submodules: `BNFC.Backend.Swift.Lexer` and `BNFC.Backend.Swift.Parser`, which constructs the Swift code responsible for the lexing and parsing of the target CFG respectively. Since the functions exposed by these submodules are only used in the construction of the Swift parser, these are not re-exported in `BNFC.Backend.Swift`.

## Description of Library Implementation

To implement this backend, we effectively had to translate between the grammar specifications for BNFC and Lotsawa.

BNFC uses a grammar specification called Labelled Backus-Naur Form (LBNF). In LBNF, every grammar is described by a series of statements, each of which have a label, a left-hand side (LHS), and a right-hand side (RHS). An exception to this are macros, which are special statements that compile into the regular syntax. Terminals and nonterminals have string names.

On the other hand, Lotsawa's grammar specification is slightly different. In Lotsawa's specification, a grammar consists of one or more rules, which consist of a LHS and RHS. However, there are no notions of terminals and nonterminals, only categories, which are represented by integers instead of strings. Because of this difference, a mapping must be created from the named terminals and nonterminals in LBNF to integeral categories in Lotsawa's specification.

This mapping is constructed by enumerating over the set of distinct terminals and nonterminals. Since this mapping is required in multiple components of the parser, we decide to store it as a read-only state of `Map` using the `Reader` monad.

Since LBNF offers several special rules like pragmas and macros (which were not supported by Lotsawa's specification), we initially thought that we had to manually expand out these special rules before we could translate them. However, we found out that BNFC already automatically expands the special rules into normal rules when parsing the grammar, so the CFG representation received by `makeSwift` is already ready for translation!

Unfortunately, we did not end up writing QuickCheck tests for the backend as we could not determine what invariants should hold for our project.

## Code Analysis

The implementation was fairly straightforward, with the biggest challenge being understanding how the CFG is represented internally in BNFC. Once we figured out how to extract the rules from the CFG and map them to integers, it was fairly easy to generate the corresponding rule in Lotsawa's specification.

One important thing to note is that this extension merely provides the backend for Swift, but does not actually enable the use of this backend in BNFC yet. However, hooking up the Swift backend to BNFC only requires minor changes:

- Add a new constructor `TargetSwift` to the `Target` data type in `BNFC.Options`, and update the following:
  - Implement `Show`
  - Implement `Maintained`
  - Implement case for `printTargetOption`
  - Implement case for `targetOptions`
  - Implement case for `specificOptions`
- Add `makeSwift` under the case for `TargetSwift` in `maketarget` in `Main`.

Performing the above changes should enable the Swift backend for use with the BNFC command-line executable.