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 more than 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-Leo parser (written by Jeffrey Kegler and ported into Swift by Dave Abrahams). Kegler has shown in particular that for any totally deterministic LL(k) or LR(k) grammar, operates in **linear time**, and works for even ambiguous grammars by returning multiple parses.

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`. Similar to the other `make` functions currently in the BNFC backend, `makeSwift` converts a Context-Free Grammar (CFG) in BNFC to fully-functioning parser code in Swift using, in our case, the Citron Lexer and Lotsawa Parser. Our implementation is not fully complete (details in the Limitations section) but the heart of the pipeline is there.

`BNFC.Backend.Swift` contains 2 submodules: `BNFC.Backend.Swift.CFtoCitronLexer` and `BNFC.Backend.Swift.CFtoLotsawaParser`, which construct 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's no explicit differentiation between terminals and nonterminals: everything is simply a category, which is represented by a fixed-width integer instead of a custom datatype (the `Cat` type used in the BNFC backend). Because of this difference, a mapping must be created from the named terminals and nonterminals in LBNF to integral 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!

The implementation was fairly straightforward, as our main task was syntax translation: translating BNFC's abstract syntax into our own abstract syntax (represented as `LotsawaRule` and `LotsawaGrammar` in our code) and then reverse engineering that back into concrete Swift syntax. The biggest challenge was 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. Extensive use of reader monads ensured that once we had established the enumeration of categories, we were using it consistently throughout the rest of the code. This was especially important while generating Swift code, an incredibly finicky task since we had to manually generate strings which must be syntactically correct Swift. In particular, this is more difficult in Swift since, like Python, it relies mostly on whitespace to parse its code, unlike `;`-based imperative languages like C++ and Rust.

Code Analysis

To mitigate the challenges of dealing with issues in code generation, we made use of the power of Haskell types and typeclasses to make this process more modular and consistent. In particular for the generation of individual rules in Lotsawa, we implemented types `LotsawaGrammar` and `LotsawaRule` which simulate the Swift types of `Grammar` and `Rule` in the Lotsawa API; similarly to generate the lexer rules in Citron for the `token` and `comment` pragmas, we used our own `CitronLexerRule` type to model these rules internally. These Haskell types internally hold the same data their Swift counterparts do, and we utilized the `Show` typeclass for them to implement their conversion into correct Swift code.

Unfortunately since there's no current obvious way to directly call Swift code from Haskell besides going directly through the C FFI, we weren't able to write the tests we'd like to, which would be to directly compare the parses created by the standard BNFC backend in Haskell's Happy parser and those created by our new Lotsawa backend. Even more broadly, there's no immediately obvious way to generate 'Arbitrary' (in terms of QuickCheck) instances of LBNF grammars to test on, since such instances would have to obey many conditions (i.e. only including the defined categories in its rules) in order to be meaningful test data; most would be unlike realistic grammars and wouldn't provide very meaningful testing. Nonetheless, we did implement an `Arbitrary` instance for the CF type to test a very specific property of the backend that doesn't depend on the grammar being reasonable.

In particular we tested the enumeration of BNFC categories, the most crucial process in our backend. We test that the enumeration is total, which itself ensures our Lotsawa grammar is well-formed. That is to say, we want to ensure that for a well-formed `CFG` (in the sense of all rules in the grammar only using defined categories)

1. Our enumeration scheme for turning the categories and terminals into integers actually enumerates everything used in the ruleset. Our implementation assigns a sentinel value of `-1` to categories that aren't in our enumeration, so our tests check that no `-1` occurs in the enumerated `LotsawaRules` we generate. We also test that
2. The enumeration is consistent with the original rules such that, having converted BNFC rules into the integer form for Lotsawa, converting back through the same maps recovers the original ruleset.

Of course using QuickCheck in this manner meant that we had to create an `Arbitrary` instance for BNFC CFGs, bringing up the same problem from earlier of the challenge in generating arbitrary well-formed grammars. We dealt with this challenge by generating arbitrary instances of categories, terminals and rules first and then forming the CFG as a wrapper around those rules, so that it would always be well-formed.

For testing the generated Swift parser itself, we used those examples of grammars and the texts they're meant to parse from BNFC's own testing suite, and ensured that Lotsawa outputs at least one parse that is equal, translating the integers back into categories, to the parse created by the Happy backend. ("At least one" since, for ambiguous grammars, Lotsawa will return multiple parses if they exist.)

Limitations

Our backend currently only supports LBNF grammars WITHOUT `layout` pragmas (which refer to whitespace tokens), ignores `position` directives in `position token` pragmas, and ignores `internal` pragmas as well. That said, Lotsawa can parse very complex whitespace syntax (i.e., Kegler's shown it can parse Haskell2010 itself) and there's no reason an improved version of this backend couldn't translate `layout` into suitable Lotsawa rules.

Another important thing to note is that this extension merely provides most of the backend logic for Swift, but does not actually enable the use of this backend in BNFC yet as we haven't written it inside the BNFC codebase (rather, we've written it as a module that externally imports BNFC for the sake of this project). 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`. As far as we can tell, a Swift backend would not require any more special options, as the parser backend accepts a superset of the grammars which BNFC accepts, so the `SharedOptions` type could be left alone.

It's possible though that since the Swift ecosystem is rapidly expanding, in particular to Windows, options may be necessary to handle certain configuration or versioning issues such that the outputted Swift packages actually build. As of right now these efforts are still nascent, so our Swift backend only officially works on macOS (any version >= 10.15) as per usual, with Linux support conditional on whether a user's Linux distribution works with at least Swift 5.7.

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

Lastly, a crucial and necessary improvement for our project would be to provide an executable test suite in Swift, so that users can verify that the generated parser indeed works on their target grammar. The generation of a test suite is done in the other BNFC backends such as the CPP and Haskell backends. Ideally we would generate the files more modularly (i.e. one for the Lexer, another for the Parser, one for the executable) to enable easier testing and integration with the BNFC's own testing module in its codebase. Right now, though we have tested that our backend produces Swift code that does build, this is not strong enough for actual integration into the codebase; comparison of the results of Lotsawa's parses with the other backends would be necessary to ensure the backend is at least as reliable as the other BNFC backends.