Parser Generators are Hard

Sander Wiig

May 19, 2023

1 Project Description

1.1 Project

I believe it was Mark Twain who once said, "*Challenges make life interesting*"; by that standard, my life has been very interesting.

The project's original goal was to create a metalanguage for defining other languages. The plan was to have an underlying semantics and then to let the user define a syntax and give bindings between that syntax and the underlying semantics. It turns out that this is somewhat difficult to achieve.
Therefore, the project goal was scaled back to just a parser generator that outputs a parser that can parse a CFG grammar and an AST for said grammar.

It turns out this also presents some challenges. An AST is very much just a parse tree but with all the superfluous information, such as parenthesis, semicolons, and other non-relevant information, removed so that all that remains is the information necessary to preserve the semantics and structure of the program. The problem then becomes, What information is relevant? Because of this, the output was changed from an AST to a parse tree.

This is where the project is now. The project is a parser generator that takes a CFG grammar(in a format similar to EBNF) and outputs a parser that can parse a program written in the user-defined grammar and outputs a parse tree for that program.

1.2 The Parser Generator

The parser generator works by first parsing a `.ebnf` file. The `.ebnf` file contains a CFG grammar that is defined in a format inspired by EBNF. The EBNF parser outputs an AST representation of the grammar that is then fed to the code generator. Both the grammar parser and the generated parser make use of the Haskell library Mega parsec. MegaParsec is a monadic parser library for Haskell, it is a fork of the Parsec library[**?**].

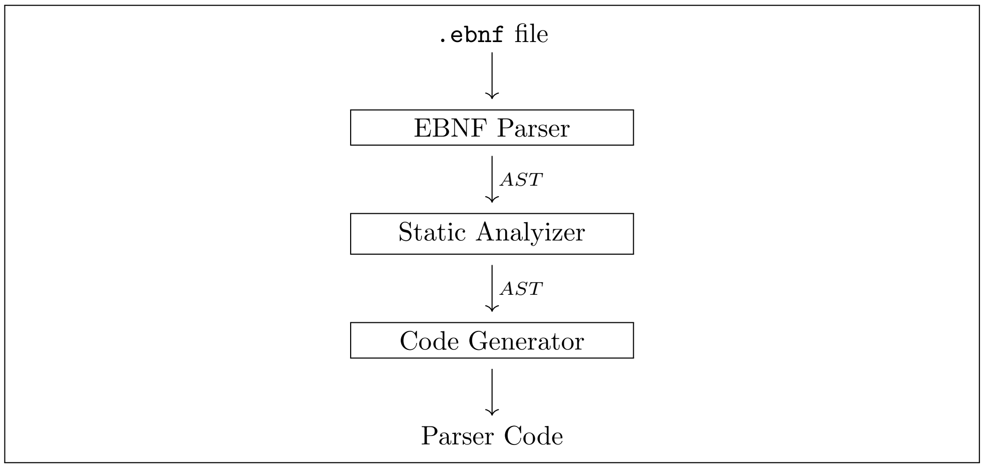

Figure 1: The Parser Generator Process

One early challenge in writing the parser was parsing infix notation. This was resolved by using the `makeExprParser` from parses-combinators[?], a companion library to MegaParsec.
This did not solve the problem since the problem would still occur in the generated parser as well, and it would be too complicated to try to identify infix notation in the user grammar.

1.3 Program internals

The library consists of two sub-libraries,

- **EBNF** - The `EBNF` library is responsible for parsing the user-defined grammar and creating an AST for it.
- **Generator** - The `Generator` takes an AST generated by the `EBNF` library and generates a Megaparsec parser that can parse a user-defined grammar.

There is also a folder containing example grammar (including the EBNF grammar itself) and code for running the tests.

EBNF

The `EBNF` library consists of the following files

- `CheckGrammar.hs` - contains functions for checking the grammar for errors.
- `EBNF.hs` - contains the AST for a grammar.
- `Lexer.hs` - contains helper functions for tokenizing the input.
- `Parser.hs` - contains the parser for the user-defined EBNF grammar.

- `REGEX.hs` - contains the AST and parser for regular expression, not implemented into the program yet.

Generator

The `Generator` library consists of the following files

- `Generator.hs` - contains functions for generating Haskell code from an EBNF AST.
- `GenUtils.hs` - contains helper functions for generating Haskell code, such as generating Haskell headers and imports, and commonly used code.
- `ParseTree.hs` - defines a simple parse tree.

Techniques and libraries

Most of the project is built on megaparsec. The project makes heavy use of `Control.Applicative` to implement the Parsers.
An attempt was also made to implement parse trees using free monoids from the library free[**?**].
This was abandoned because I couldn't find out how they worked, and I didn't see the advantage over a normal tree.
The library `pretty-simple` was used to pretty print ASTs and other data structures[**?**].

The techniques and libraries used seem to have been fitting, I can not think of any other techniques that would have been useful.

2 Does it work?

2.1 "There are 56 consecutive parentheses"

Unfortunately, the parser generator does not quite work as intended...
While it can parse a grammar and generate a parser for it, the parser it generates is only functional for the smaller grammars(ex1, ex2, etc). When given the final test, the EBNF grammar itself ends up generating some *interesting* code....:

```
return (Symbol "letter" [((((((((((((((((((((((((((((((((((((((((((((((((((
    (Literal $ symbol "A") <|> (Literal $ symbol "B")) <|>
    (Literal $ symbol "C")) <|> (Literal $ symbol "D")) <|>
    (Literal $ symbol "E")) <|> (Literal $ symbol "F")) <|>
    (Literal $ symbol "G")) <|> (Literal $ symbol "H")) <|>
    (Literal $ symbol "I")) <|> (Literal $ symbol "J")) <|>
    (Literal $ symbol "K")) <|> (Literal $ symbol "L")) <|>
    (Literal $ symbol "M")) <|> (Literal $ symbol "N")) <|>
    (Literal $ symbol "O")) <|> (Literal $ symbol "P")) <|>
    (Literal $ symbol "Q")) <|> (Literal $ symbol "R")) <|>
    (Literal $ symbol "S")) <|> (Literal $ symbol "T")) <|>
    (Literal $ symbol "U")) <|> (Literal $ symbol "V")) <|>
    (Literal $ symbol "W")) <|> (Literal $ symbol "X")) <|>
    (Literal $ symbol "Y")) <|> (Literal $ symbol "Z")) <|>
    (Literal $ symbol "a")) <|> (Literal $ symbol "b")) <|>
    (Literal $ symbol "c")) <|> (Literal $ symbol "d")) <|>
    (Literal $ symbol "e")) <|> (Literal $ symbol "f")) <|>
    (Literal $ symbol "g")) <|> (Literal $ symbol "h")) <|>
    (Literal $ symbol "i")) <|> (Literal $ symbol "j")) <|>
    (Literal $ symbol "k")) <|> (Literal $ symbol "l")) <|>
    (Literal $ symbol "m")) <|> (Literal $ symbol "n")) <|>
    (Literal $ symbol "o")) <|> (Literal $ symbol "p")) <|>
    (Literal $ symbol "q")) <|> (Literal $ symbol "r")) <|>
    (Literal $ symbol "s")) <|> (Literal $ symbol "t")) <|>
    (Literal $ symbol "u")) <|> (Literal $ symbol "v")) <|>
    (Literal $ symbol "w")) <|> (Literal $ symbol "x")) <|>
    (Literal $ symbol "y")) <|> (Literal $ symbol "z")])
```

This example above is the parser for a letter character. Since the ebnf parser parses this as a sequence of `OR`s and because the parser generator defines the parser for `OR` as

```
        (e1) <|> (e2)
```

we end up with 56 parentheses.

2.2 How to make it work

Most problems would be solved by performing a static analysis of the grammar to resolve possible problems. Right now very little static analysis is performed on the grammar, ideally, the grammar would be checked to make sure that it was free from ambiguities, this could be done by attempting to build a PDA from the grammar.

3 How to Use it

3.1 Generating a Parser

To generate a grammar one can use the `generate` function in `GenParser.hs`. `generate` takes a grammar `string` and returns the parser in string form.

See the `test` function in `test/GeneratorTest.hs` for an example of how to use the different functions.

3.2 Using the generated parser

After using `generate` and writing the string to a file you can use the parsers in the file as needed.

References

[1] Gosnell, D. pretty-simple: pretty printer for data types with a 'show' instance. https://hackage.haskell.org/package/pretty-simple.

[2] Karpov, M., and Washburn, A. parser-combinators: Lightweight package providing commonly useful parser combinators. https://hackage.haskell.org/package/parser-combinators-1.3.0/docs/Control-Monad-Combinators-Expr.html.

[3] Kmett, E. A. free: Monads for free. https://hackage.haskell.org/package/free.

[4] Megaparsec contributors, Martini, P., and Leijen, D. megaparsec: Monadic parser combinators. https://hackage.haskell.org/package/megaparsec.