CS 583 Final Project Report

Rikki Gibson and Cody Ray Hoeft

1 Overview

The Variational Parser is a tool supporting C preprocessor-like language extensions. The intended use case for this parser is to support an Atom editor plugin enabling users to inspect and modify documents containing the equivalent to #ifdefs by specifying whether dimensions are defined, and if so, whether to choose between a "left" or "right" alternative.

What we're delivering enables users to do two basic actions:

1. Parse a document containing our variational syntax into an abstract syntax tree with line and column numbers that support syntax highlighting.
2. Consume an AST and a set of user selections, producing an AST with the specified dimensions reduced to one of their branches. (the "view" function).

Some of the work done by this parser is already done by the C preprocessor, and the tool bears a vague resemblance to some commonly-used "language service" programs that support editor plugins that maximize the amount of code that can be shared between plugin implementations for various different editors.

2 Types and functions used

`data Segment` is the core of our abstract syntax, providing a case for a simple text node as well as a case for a choice node with a dimension, left branch and right branch.

The parsers `choiceSegment`, `textSegment`, `program`, and several supporting parsers correspond to the shape of our abstract syntax and define

the way that concrete syntax is transformed into abstract syntax (that's the definition of parsing, probably!)

`view` is the main, recursively referenced function for reducing ASTs given a set of user selections, while `getView` serves as an entry point for generating views.

`jsonPrepare` is how we take a just-parsed AST and annotate it with line and column information that does not count the variational concrete syntax when computing line and column positions. This problem and our solution will be discussed further in the design decisions section.

3 Design decisions

1. Our lives were simplified a great deal by the library Megaparsec, which allows users to construct parsers by composing them with other parsers and applying functions that consume and produce parsers, etc. The combinator design pattern, at first a bit of a mind-bender, we eventually found was simple and clear.

2. Using JSON, stdin, and stdout for inter-process communication was a pretty simple choice for us. Aeson proved a good library for encoding and decoding JSON, and our choice to use GHC's generics extension allowed us to minimize the amount of boilerplate translation code in the project.

3. We made the decision to produce different executable targets for the different "functions" we expect the editor to invoke. This seemed to be simpler than taking command line arguments and parsing them to decide which main function to perform.