An Embedded Language for Context-Free Grammars with Multiple Interpretations

Yan Han (yh412)

January 2019

1 Introduction

My project is an embedded language in Haskell for specifying the rules of a context free grammar (CFG) in extended Backus-Naur form (EBNF). I use the tagless final style described by Oleg Kiselyov [1] to allow a single term in the language to be interpreted in multiple ways. Each interpretation produces a specific type from terms in the embedded language. I implement three distinct interpretations:

1. A basic pretty-printer interpretation that produces a string from a term.
2. A parser interpretation using monadic parser combinators [2] that produces a parser for the grammar specified by a term. A parser is a function that maps strings to zero or more parse trees.
3. A graphical interpretation that produces syntax diagrams for a grammars using the diagrams library [3].

The project uses the Haskell stack. Running `stack ghci` in the root directory will start a repl, which suffices to run all of the code examples in the project.

2 The Language

The code for this section is located in `src/CFG.hs`.

I specify the language using a typeclass parameterized by a representation type.

The representation type can be thought of as a production rule for a nonterminal. Most of the language consists of functions for creating production rules, and for combining production rules to create new ones. These functions correspond to the specification of EBNF.

```haskell
class CFGSYM repr where
  t :: String -> repr   -- Terminal
  n :: String -> repr   -- Nonterminal

  cat :: [repr] -> repr -- Concatenation
  alt :: [repr] -> repr -- Alternation
  opt :: repr -> repr   -- Optional
  rep :: repr -> repr   -- Repetition

  rules :: [(String, repr)] -> CFG repr
```

The `t` and `n` functions create the simplest production rules using terminal strings and nonterminal names, respectively.

The `cat` and `alt` functions create production rules from a list of production rules. `cat` concatenates all of the input rules in series, whereas `alt` creates the disjunction of all of the input rules.

The `opt` and `rep` functions transform a single production rule. `opt` makes that rule optional (can be skipped or applied once), and `rep` allows that rule to be repeated (used any number of times).

The `rules` function is special. It represents the top-level structure of a CFG as a mapping of nonterminal names mapped to production rules. I use a newtype wrapper, `CFG`, to statically disallow this structure from being treated as a production rule:

```haskell
newtype CFG a = CFG {unCFG :: a}
```

To demonstrate the various interpretations of the language, I use a CFG that represents parenthesized arithmetic expressions on integers:

```
expr = integer | "(" , expr , op , expr , ")";
op = "+" | "-" | "*" | "/";
integer = "0" | ["-"] , nonzero , {"0" | nonzero};
nonzero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

This is written using the DSL as follows:

```haskell
arith = rules
  [ ("expr", alt [n "integer", cat [t "(", n "expr", n "op", n "expr", t ")"]])
  , ("op"  , alt [t "+", t "-", t "*", t "/"])
  , ( "integer"
    , alt [t "0", cat [opt (t "-"), n "nonzero", rep (alt [t "0", n "nonzero"])]]
    )
  , ( "nonzero"
    , alt [t "1", t "2", t "3", t "4", t "5", t "6", t "7", t "8", t "9"]
    )
  ]
```

3 The Pretty Printer Interpretation

The code for this section is located in `src/PrettyPrinter.hs`.

The pretty printer interpretation produces the EBNF form of a grammar as a string. This is straightforward, since there is a direct correspondence between our embedded language and the operations allowed by EBNF. For our arithmetic expression language, the output is precisely the first form of the grammar given in the previous section, minus the colors.

```
instance CFGSYM String where
  t str = "\"" ++ str ++ "\""
  n str = str
  opt str = "[" ++ str ++ "]"
  rep str = "{" ++ str ++ "}"
  cat   = intercalate " , "
  alt   = intercalate " | "

  rules = CFG . showRules   where
    showRules []            = ""
    showRules ((n, s) : rs) = n ++ " = " ++ s ++ ";\n" ++ showRules rs
```

To print the pretty-printed form of the arithmetic expression language, run `putStr arithStr` in GHCi.

4 The Syntax Diagram Interpretation

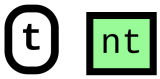
The code for this section is located in `src/SyntaxDiagram.hs`.

The syntax diagram interpretation is a fancy version of the pretty printer interpretation. It uses the diagrams library [3] to produce diagrams that depict the possible "paths" one can take to produce strings satisfying a given production rule. The functions in this interpretation build objects of type `Diagram B`, where `B` is a type synonym for the SVG backend of the diagrams library.

The diagrams library provides many ways to create and combine diagrams. The implementations of each function are messy and dependent on library-specific functions and idioms, so I provide high-level explanations of what they do rather than presenting their code.

4.1 The interpretation

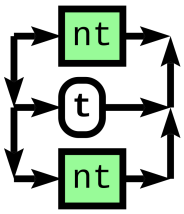
`nt` and `t` take nonterminal names and terminal strings, respectively, and draw them enclosed in stylized rectangles. These are the basic building blocks that the other functions combine into complex diagrams.



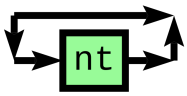
`cat` takes a list of diagrams and produces a combined diagram in which there is a sequential path through them.

`alt` takes a list of diagrams and draws a parallel stack of paths through all of them, only one of which can be taken.

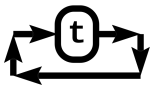
This function imposed an interesting design constraint on the DSL. `alt` (and for consistency, `cat`) has the signature `[repr] -> repr` but could have had the signature `repr -> repr -> repr`, which is more minimal and avoids the semantics of the empty list, but makes it impossible to tell where a chain of alternations ends. I chose the former over the latter because having access to the entire list at once greatly simplifies the drawing of diagrams in the vertical "stack" style.



`opt` takes a single diagram and draws two branches: one going through it, and one that goes past it.



`rep` takes a single diagram and draws a path through it that loops back around, representing optional repetition.

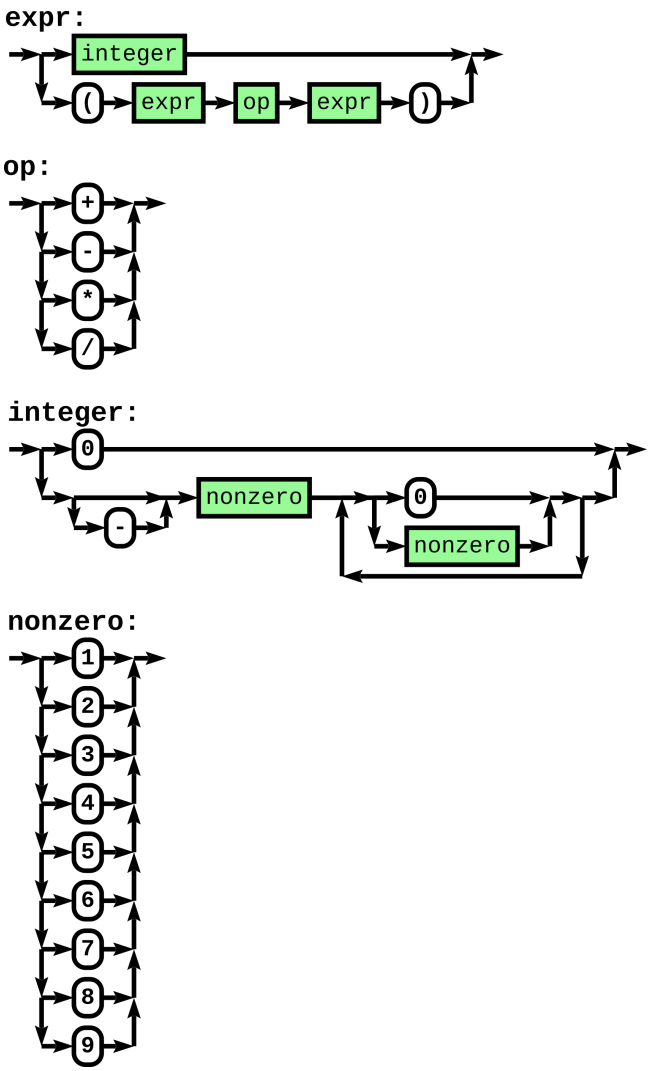


4.2 The example

Below is the syntax diagram for the arithmetic DSL, showing off the `rules` function as well as some more complex, nested production rules. To reproduce it, run the following lines in sequence in GHCi:

```
import Diagrams.Backend.SVG.CmdLine
let x = mainWith arithDiagram
:run x ["-w2000", "-h2000", "-odiagram.svg"]
```

This creates a .svg file named diagram.svg in the root directory of the project.



5 The Parser Interpretation

The code for this section is located in `src/Parser.hs`.

The parser interpretation uses a simple implementation of monadic parser combinators to build recursive descent parsers from production rules. Most of the base parser code is adapted from Hutton's tutorial [1].

My approach is rather naive and produces parsers that do not always terminate for certain non-LL(k) languages. I also did not bother to optimize for performance at all. Nonetheless, the approach produces working parsers, and was an enjoyable introduction to the elegance of this approach to parsing.

5.1 Monadic parser combinators

I use the following definition of a parser:

```
newtype Parser a = Parser { parse :: String -> [(a, String)] }
```

The idea is that a parser is a function that consumes some piece of an input string and returns the remaining string, as well as something that represents the parsed piece of the string.

More formally, a parser for type `a` is a function that takes a `String` to be parsed and returns a list of pairs of type `(a, String)` - the result of parsing the string, and the remaining, unparsed string. Returning the empty list represents failure, and returning a list containing more than one element represents ambiguous parsing.

`Parser` is a functor since we can `fmap` a function over a parser by applying it to the result:

```
instance Functor Parser where
    fmap :: (a -> b) -> Parser a -> Parser b
    fmap f p = Parser $ \cs -> [ (f a, res) | (a, res) <- parse p cs ]
```

It is also a monad:

```
instance Monad Parser where
    return :: a -> Parser a
    return v = Parser $ \cs -> [(v, cs)]

    (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = Parser $ \cs -> concat [ parse (f v) res | (v, res) <- parse p cs ]
```

`return` produces a parser that consumes none of the input string and returns a constant result. `bind` takes a parser and a function that produces parsers from its result type. It produces a parser that runs the parser-producing function on

each result of the original parser, applies that parser to the remaining string for each result, and concatenates the lists of results.

Some more useful parsers include `zero`, the parser that always fails, and `plus`, the parser that runs two parsers on something and concatenates the results. These functions make up `Alternative` and `MonadPlus` instances for `Parser`, but I use their names in the code for clarity.

```
zero :: Parser a
zero = Parser $ const []

plus :: Parser a -> Parser a -> Parser a
plus p p' = Parser $ \cs -> parse p cs ++ parse p' cs
```

These functions and basic parsers are the building blocks for more complex ones, which all have short descriptions in the source code.

5.2 The interpretation

We want the functions in this interpretation to accept and produce parsers. To make the parsers as generic as possible, we define a `ParseTree` type that represents the parse tree of a string using an arbitrary CFG.

```
data ParseTree =
  List [ParseTree]
  | N String [ParseTree]
  | T String
```

This represents a parse tree where the leaves (the `T` constructor) are terminal strings and the nodes (the `N` constructor) are nonterminals. The `List` constructor exists because we sometimes need to create parsers that will produce a list of parse trees without knowing what nonterminal the parse trees will be a part of.

`Parser ParseTree` is a candidate for the representation type for this interpretation. However, the nature of nonterminals makes this insufficient. The `n` function builds a parser out of the name of a nonterminal, but this requires access to the complete parser for that nonterminal, which may depend on the `n` call itself. In other words, we need a way for the `n` function to access the result of the `rules` function. I accomplish this by attaching the final set of parsers to the representation type, making it `Reader (Map String (Parser ParseTree)) (Parser ParseTree)`. More concretely, the `rules` function takes its input list of nonterminals mapped to parse trees, builds a `Map` from it, and stores it in the `Reader` monad. Since `rules` is always the outermost function, all of the composite functions will have access to this map via `Reader`. Haskell's laziness makes the cyclic dependencies present in this approach unproblematic.

This solution has downsides because of the additional boilerplate required for the

`Reader` wrapper. Even though the attached `Map` is only used in the `n` function, all of the other functions have to deal with the monadic wrapper. Additionally, it makes the representation type significantly less readable and requires it to be unwrapped before the complete parser can be used. Nonetheless, I believe it is the best solution because it manages to hide most of the additional complexity in the implementation of one interpretation, while the only other approach I thought of involved adding the `Map` to the signature of `n`, adding unnecessary complexity to all interpretations of the language and making it clunky to use.

The implementations of the language functions usually lean on one or two simple parser combinators. I use `do` notation to unwrap the monadic context from `Reader`, and monadic comprehension notation to construct the parser itself, to provide some syntactic distinction between the two layers.

For example, `rep` takes a parser and returns a parser that applies it zero or more times to the input and wraps the result in the `List` constructor of `ParseTree`, to be labeled in the future using the `N` constructor.

```
rep parser = do
    p <- parser
    return [ List xs | xs <- many p ]

-- Helper combinator
many p = [ x : xs | x <- p, xs <- many p ] `plus` return []
```

`t` takes a string and returns a parser that consumes precisely that string, wrapping the result in the appropriate `ParseTree` constructor. The `string` combinator is defined earlier in the file.

```
t str = return [ T s | s <- string str ]
```

`alt` takes a list of parsers and returns a parser that runs all of them in parallel on the input string.

```
alt parsers = do
    ps <- sequence parsers
    return $ foldl' plus zero ps
```

`rules` creates a map of all parsers using the aforementioned circular definition. It then returns a modified version of the parser for the first nonterminal in the list, that returns only results that have parsed the entire string.

```
rules lst = CFG $
    let startName = fst . head $ lst
        allParsers :: Map String (Parser ParseTree)
        allParsers = foldr
          (\(name, parser) acc ->
            Map.insert name (runReader parser allParsers) acc
          )
          Map.empty
```

```
          lst
        startParser = fromJust $ Map.lookup startName allParsers
    in  return [ N startName (toList xs) | xs <- complete startParser ]
```

5.3 The example

Run `parse arithParser "string"` to print the list of parse results for a string using our arithmetic expression grammar. Since it is not an ambiguous grammar, the parser will produce either zero or one result. Here are some examples.

```
> parse arithParser "01"
[]
> parse arithParser "(1+2))"
[]
> parse arithParser "(10+23)"
[(expr["(",expr[integer[nonzero["1"],"0"]],op["+"],
    expr[integer[nonzero["2"],nonzero["3"]]],")"],"")]
> parse arithParser "((1+2)*3)"
[(expr["(",expr["(",expr[integer[nonzero["1"]]],op["+"],expr[integer[nonzero["2"]]],")"],
    op["*"],expr[integer[nonzero["3"]]],")"],"")]
```

I also provide a simple example of a parser that does not terminate. It is in the same file as the rest of the parser code.

```
nonTerminating = rules [("x", alt [t "1", cat [n "x", t "2"]])]
nonTerminatingParser :: Parser ParseTree
nonTerminatingParser = getParser nonTerminating
```

Since this language has a left-recursive production rule, it loops forever when the parser is applied, examining deeper and deeper nestings of the same nonterminal. However, because Haskell is lazy, it is possible to short-circuit this in instances where it finds a correct parse tree by only asking for the first element of the result list. For example:

```
> parser nonTerminatingParser "asdf"
<loops forever>
> parse nonTerminatingParser "122"
[(x[x[x["1"],"2"],"2"],"")<loops forever>
> head $ parse nonTerminatingParser "asdf"
<loops forever>
> head $ parse nonTerminatingParser "122"
(x[x[x["1"],"2"],"2"],"")
```

References

[1] O. Kiselyov, "Typed Tagless Final Interpreters," D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and J. Gibbons, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7470, pp. 130–174. [Online]. Available: http://link.springer.com/10.1007/978-3-642-32202-0_3

[2] G. Hutton, "Monadic parser combinators," *Journal of Functional Programming*, 1996, iSBN: 9781605583792.

[3] B. Yorgey, "Diagrams," 2016. [Online]. Available: hackage.haskell.org/package/diagrams