# An Approach to Context-Free Parsing in Haskell

Exclusively for grade assessment in the 30-60-496 course at the University of Windsor. The ideas in this paper are still being expanded upon.

Eric Brisco

September 1, 2013

# Contents

# 1 Assumptions and terms

This paper uses Haskell as an implementation language and refers to many of its concepts without explanation. Information on Haskell can be found at `haskell.org`.

This paper does not establish proofs, instead leaning on case examples. All conclusions should be taken under this light.

A common convention is used for context free grammars — which will be referred to plainly as "grammars". This is an example.

```
A → aA | b | c
```

A notion of starting non-terminal is irrelevant for this paper. A parser will be created for each production rule; selecting a starting non-terminal is only a decision of which parser to evaluate.

This grammar implies we can substitute the non-terminal `A` for any production on the right. Productions are delimited by the pipe symbol. `A` may become `aA`, `aaA`, `aab`, for example.

Upper case letters are always non-terminal and may be substituted. Lower case letters are always terminals and may not be substituted.

We refer to "`A`", "the grammar", "the language", "the parser of `A`", or "the program of `A`" interchangeably most of the time, though with intended sense given the context.

"Sentences of a language" refers to what others may call "words of a language", but in either case means the set of all finite sequences of terminals which can be produced by substitution of the starting non-terminal using the grammar of the language.

Code references inline with text `will look like this` whereas things not tied to implementation `may look like this`.

When giving code snippets, the prefix `-- >>>` means the following is an expression being evaluated, and the next line is the result of the evaluation after the prefix `--`.

# 2 Listing all sentences of a language

```
A → a
```

The language of `A` has just one sentence: `"a"`. We express the list of all sentences in the language `A` as `["a"]`.

We define the function `word` which takes a list of terminals and returns that list in an outer list.

```
word :: [a] -> [[a]]
word = (:[])
```

With `word` the language `A` can be implemented as a program which lists all of its sentences.

```
sA :: [[Char]]
sA = word "a"
-- >>> sA
-- ["a"]
```

$$\texttt{B} \rightarrow \texttt{a} \mid \texttt{b}$$

This grammar has two productions for `B`. All sentences in the language `B` is the list `["a","b"]`.

We define a function called `//`, whose name purposefully mimics the "`|`" symbol in the grammar definition. The function takes two lists and concatenates them.

```
(//) :: [a] -> [a] -> [a]
(//) = (++)
infixl 3 //
```

For now we use the `++` operator, but this definition will become erroneous and thus redefined later. With `//` the grammar `B` can be implemented.

```
sB :: [[Char]]
sB = word "a" // word "b"
-- >>> sB
-- ["a", "b"]
```

Many productions can be put together with `//`.

$$\texttt{C} \rightarrow \texttt{foo} \mid \texttt{baz} \mid \texttt{qux}$$

```
sC :: [[Char]]
sC = word "foo" // word "baz" // word "qux"
-- >>> sC
-- ["foo","baz","qux"]
```

Multiple production rules are also possible.

$$\begin{aligned} \texttt{D} &\rightarrow \texttt{hello} \mid \texttt{E} \\ \texttt{E} &\rightarrow \texttt{world} \mid \texttt{there} \end{aligned}$$

```
sD = word "hello" // sE
sE = word "world" // word "there"
-- >>> sD
-- ["hello","world","there"]
```

A problem is raised with this following redefinition of `D`, `E` and the inclusion of `F`.

```
D → hey | hello
E → world | there
F → D E
```

That is, `F` is any `D` followed by an `E`. To list all sentences for `F`, every `D` must be paired with every `E`. This is familiar as the Cartesian product: all pairs $(a, b)$ where $a \in A$ and $b \in B$.

To implement Cartesian product we define the function `&`. For now, a list comprehension is used, but as with `//` this definition will become erroneous and thus redefined.

```
(&) :: (Monoid a) => [a] -> [a] -> [a]
a & b = [mappend x y | x <- a, y <- b]
infixl 4 &
```

For generality, `mappend` is used. However, for all of our examples, the type `a` is `[Char]`; effectively, `++` is being used under the hood of `mappend`.

Now `F` can be implemented.

```
sD = word "hey" // word "hello"
sE = word "world" // word "there"
sF = sD & sE
-- >>> sF
-- ["heyworld","heythere","helloworld","hellothere"]
-- >>> (word "a" & word "b") == word "ab"
-- True
```

The precedence of `&` is higher than that of `//` so that we can implement the grammars in a natural way. This is demonstrated in the following examples.

```
-- >>> (word "a" & word "b") // (word "c" & word "d")
-- ["ab","cd"]
--
-- >>>  word "a" & word "b" // word "c" & word "d"
-- ["ab","cd"]
--
-- >>> word "a" // (word "b" & word "c") // word "d"
-- ["a","bc","d"]
--
-- >>> word "a" // word "b" & word "c" // word "d"
-- ["a","bc","d"]
--
-- >>> (word "a" // word "b") & (word "c" // word "d")
-- ["ac","ad","bc","bd"]
```

These two operations, `//` and `&`, are all we need for any grammar. However, there are shortfalls in their implementations, as previously mentioned, which we now discuss and correct.

```
G → gG | g
H → hH | h
```

These languages have right-recursive production rules, and what is significant is that there are infinitely many sentences in such a language. Sentences of `G` are `["g","gg","ggg" ...` and similarly for `H`.

```
I → G | H
```

With the current definition of `++` for `//`, `G // H` behaves undesirably. `++` will put the list of sentences of `H` after the end of the list for `G`. But, `G` has no end, so `++` will never yield any sentences of `H`.

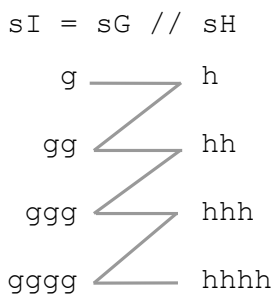
```
J → G H
```

A list comprehension was used to implement `&`, but this behaves undesirably for `G & H`. The list comprehension is stuck on the first sentence of `G` as it is paired with all sentences of `H`, yielding the list `["gh","ghh","ghhh"...`. A reverse order of the generators will be stuck on the first sentence of `H` instead and thus yield the list `["gh","ggh","gggh"...`.

This is unacceptable because we want to list all sentences of `I` and `J`, not just some. A solution for `//` is discussed first. All sentences of `G` cannot be yielded before those of `H`, or vice versa, so instead we interleave the lists together. That is, take one sentence from `G`, then one from `H`, then another from `G`, and so on.

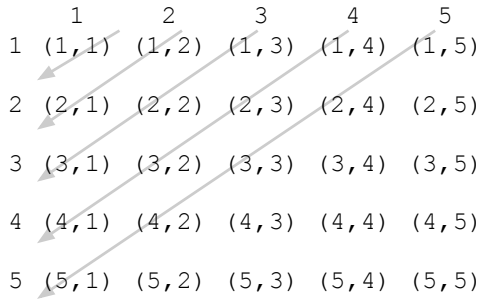
```
interleave :: [a] -> [a] -> [a]
interleave [] yss = yss
interleave (x:xs) yss = x : r
  where
  r = case yss of
    [] -> xs
    (y:ys) -> y : interleave xs ys

(//) = interleave
```

The obvious definitions, as we have established thus far, for `sG`, `sH`, and `sI` are assumed. The `interleave` function is illustrated in the following diagram. As evident, all sentences of `G` and `H` will be listed by `sI`.




A note on `interleave`: we do not pattern match on `yss` purposefully until `x` is yielded. This allows `interleave` to be productive when yss is recursively another `interleave` to infinite depth.

A solution for `&` is discussed next. In the following illustration, sentences of `G` are along the top, where `"g"` has been substituted for 1, `"gg"` for 2, and so on. The same applies to `H` down the left. This represents the Cartesian product of `G` and `H`.




Starting at `(1,1)`, the attached arrow is followed before moving to `(1,2)`, where again the attached arrow is followed before going to `(1,3)`, and so on, moving horizontally across the top. Using this strategy for Cartesian product, all sentences for `G & H` will be yielded. This borrows from Cantor, the mathematician, and his proof that the Rationals are countable.

We define the function `carp` which uses this strategy for Cartesian product.

```
carp :: (a -> b -> c) -> [a] -> [b] -> [c]
carp f [] _ = []
carp f _ [] = []
carp f (x:xs) (y:ys) = topRight [x] xs [y] ys
  where
  topRight [] [] _ _ = []
  topRight xL [] yL [] = zipWith f xL yL
      ++ topRight (tail xL) [] yL []
  topRight xL [] yL (y:yR) = zipWith f xL yL
      ++ topRight xL [] (y:yL) yR
```

```
  topRight xL (x:xR) yL [] = zipWith f xL yL
      ++ topRight (tail xL ++ [x]) xR yL []
  topRight xL (x:xR) yL (y:yR) = zipWith f xL yL
      ++ topRight (xL++[x]) xR (y:yL) yR
```

Another strategy using `interleave` can be defined which will also yield the entire Cartesian product. Unlike `carp` however, it will progress through the left list more than the right. We name the function `carpl` due to this left bias.

```
carpl :: (a -> b -> c) -> [a] -> [b] -> [c]
carpl f xss = foldr interleave []
    . fmap (\y -> fmap (flip f $ y) xss
```

The idea of `carpl` is useful later in the development of the parser. `carpl` maps each element of the right list onto its own copy of `xss`, yielding a list of lists. This roughly correlates with the list comprehension used for the first version of `//`, and would behave the same if `++` was used to fold the list of lists instead of `interleave`.

Using the corrected definitions for `&` and `//`, the implementations for `I`, `J`, `G`, and `H` are demonstrated.

```
sG = word "g" // word "g" & inf sG
sH = word "h" // word "h" & inf sH
sI = sG // sH
sJ = sG & sH
-- >>> sI
-- ["g","h","gg","hh" ...
-- >>> sJ
-- ["gh","ghh","ggh","ghhh","gghh","gggh" ...
```

The non-recursive productions are first purposefully. If the productions are reversed, the first element would be infinite, which is not a valid sentence.

The function `inf` relaxes the strict pattern matching in `carp`. The following is the definition.

```
inf :: [a] -> [a]
inf ~(x:xs) = x : inf xs
```

As defined, `carp` checks if the right list is empty, because if so then the Cartesian product is empty. However, in `word "g" & sG`, this means recursing on `sG`, which, in turn, recurses on `sG` again for the same reason, and therefore the program is not productive.

By using `word "g" & inf sG`, the right list is guaranteed as non-empty because `inf` always yields an element — or causes a runtime exception if the list actually empties. The recursive dependency is broken and allows `sG` to be productive.

Left recursive and middle recursive grammars can also be defined. For left recursion, it is imperative that non-recursive rules are first, otherwise the program will recurse before producing output. The following are more example grammars and their corresponding implementations.

```
S → aSa | aa
L → Lx | Ly | z
M → Nm | x
N → Mn | y
```

```
sS = word "aa" // word "a" & inf sS & word "a"
sL = word "z" // inf sL & word "x" // inf sL & word "y"
sM = word "x" // inf sN & word "m"
sN = word "y" // inf sM & word "n"
```

Grammars that describe languages with no sentences we reject as having no useful implementation. These are some some examples.

```
A → aA    (1)
B → Bx    (2)
C → cD    (3)
D → dC    (4)
E → E     (5)
```

1. No production allows the removal of `A`. The corresponding program is still productive and yields an infinite list of infinite lists.
2. No production allows the removal of `B`. The only productive value of the corresponding program is that the list of sentences can be confirmed as not null. An attempt to move past the first element, or to evaluate the first element, will not be productive.
3. With (4), this has the same problem as (1) but is disguised in mutual recursion.
5. There are no terminals.

Our programs will produce duplicate sentences for ambiguous grammars.

```
A → a | AA
```

```
sA = word "a" // inf sA & inf sA
-- >>> sA
-- ["a","aa","aaa","aaa","aaaa","aaaa" ...
```

With what has been developed thus far, for any sensible grammar a program can be implemented which will list all sentences in its language.

Given an input sentence, we can confirm it is in the language by testing if it is in the list of all sentences. To confirm a sentence is not in the language, some maximum search depth must be calculated because the list of sentences may be infinite. While speculatively this is possible, such a question was not explored for this project.

# 3 Listing all syntax trees of a language

The main objective of parsing is to transform a list into a tree. Therefore, we take the next step to list all syntax trees instead of simply sentences.

Few changes need to be made to accomplish this. Currently we are flattening the tree structure with `++ :: [a] -> [a] -> [a]` (the implementation of `mappend` for lists). However, note that `carp` is more general than this: `carp :: (a -> b -> c) -> [a] -> [b] -> [c]`.

$$\texttt{A} \rightarrow \texttt{aA} \mid \texttt{b}$$

We can define a corresponding data type for this grammar.

```
data A = A0 Char A | A1 Char
  deriving Show
```

`A0` is the notation we use for the first production of `A`. Naturally, `A1`, `A2`, and so on follow. A data type may only refer to types, and therefore we must use the type of the terminal `'a'` of `Char`. The tree generator program has the responsibility to ensure the value of `Char` is correct.

Note the type of `A0 :: Char -> A -> A`. It follows then that `carp A0 :: [Char] -> [A] -> [A]`. Also note the type `A1 :: Char -> A`. The following is a tree generator for `A`.

```
tA = [A1 'b'] // carp A0 "a" (inf tA)
-- >>> tA
-- [A1 'b', A0 'a' (A1 'b'), A0 'a' (A0 'a' (A1 'b')) ...
```

This definition is syntactically unpleasing, so we make use of Haskell's Applicative type class to define a cleaner interface. Applicative will also deal with longer production rules cleanly, such as $\texttt{A} \rightarrow \texttt{aBcDeF}$.

To define an Applicative interface, a type is required for tree generators. We name this type `Trees`.

```
newtype Trees a = Trees [a]
  deriving Show
```

We define Functor by delegating to `fmap` on the contained list.

```
instance Functor Trees where
  fmap f (Trees xs) = Trees $ fmap f xs

instance Applicative Trees where
  pure a = Trees [a]
  (Trees xs) <*> (Trees ys) = Trees $ carp ($) xs ys
```

We define `<*>` for `Trees` to mean `carp`. The type of `(<*>) :: f (a -> b) -> f a -> f b` clearly implies `($) :: (a -> b) -> a -> b` can be used to combine the pairs.

Alternative is also defined for `Trees`. This offers the operator `<|>` in place of `//`.

```
instance Alternative Trees where
  empty = Trees []
  (Trees xs) <|> (Trees ys) = Trees $ interleave xs ys
```

The new equivalent for `word` is called `leaf`, and is defined simply as `pure`. Also, because `Trees` hides away the underlying list, `inf` can no longer operate on it. Therefore, `tinf` is introduced and simply calls down to `inf`.

```
leaf :: a -> Trees a
leaf = pure

tinf :: Trees a -> Trees a
tinf (Trees xs) = Trees $ inf xs
```

The tree generator for `A` can now be defined with cleaner syntax. Note that `<$>` is an infix alias for `fmap`.

```
tA = A1 <$> leaf 'b' <|> A0 <$> leaf 'a' <*> tinf tA
--- >>> :type tA
--- tA :: Trees [A]
--- >>> tA
--- Trees [A1 'b', A0 'a' (A1 'b'), A0 'a' (A0 'a' (A1 'b')) ...
```

As expected, the results for `tA` are the same as before. We give one more example of a tree generator in the following.

$$\texttt{B} \rightarrow \texttt{Bx} \mid \texttt{By} \mid \epsilon$$

```
data B = B0 B Char | B1 B Char | B2
  deriving Show
tB =   leaf B2
  <|> B0 <$> tinf tB <*> leaf 'x'
  <|> B1 <$> tinf tB <*> leaf 'y'
-- >>> tB
-- Trees [B2, B1 B2 'y', B0 B2 'y', B1 (B1 B2 'y') 'y' ...
```

With what has been defined thus far, all syntax trees and all sentences can be generated for a language. If a maximum search depth can be calculated, then this is a working parser, capable of taking any input sentence and returning the corresponding syntax tree, or nothing if the sentence is not in the language.

Therefore, since the parser is speculatively complete, we now address a new criteria: performance. The tree generator is absurdly impractical because it spends incredible amounts of time generating invalid trees for a given input sentence. For instance, with the last tree generator for `B`, even if the input is `"yxyxyx"`, the generator will return numerous trees for `"x..."` and `"xy..."` and `"xyx..."` and so on.

The parser should take the input sentence into account. If the first terminal is `"x"`, then it can ignore trees that start with `"y"`. This logic can be applied to the next terminal in input unto the end. By doing this, the number of trees the program considers is dramatically reduced, and thus will be more practical for parsing.

# 4 Culling syntax trees based on an input sentence

We describe what `leaf` should return based on various inputs. We name this new iteration on `leaf` as `tok`.

```
tok 'a' "xyz" == []
tok 'a' "ayz" == [("yz", 'a')]
tok 'a' ""    == []
tok 'a' "aaa" == [("aa", 'a')]
tok 'a' "a"   == [("", 'a')]
```

From this, the type of `tok` is `tok :: (Eq a) => a -> [a] -> [([a],a)]`. The constraint of `Eq` is required because `tok` will have to use the equality operator to compare the terminal it is trying to match with the terminal coming from input.

`tok` takes a terminal and returns a parser as the type `[s] -> [([a],a)]`. A parser takes a list of terminals and returns a list of 2-tuples. Each element in the list is a valid parse of the input list. The left value in the tuple is what input was left over, and the right value is a label to remember what has been parsed. Specifically for `tok`, the parser looks at the head of input and checks if the terminal is what it is expecting. If so, it removes the head from input and returns the tail. Else, it returns an empty list. This is the first implementation for `tok`.

```
tok :: (Eq a) => a -> [a] -> [([a],a)]
tok t [] = []
tok t (x:xs) = if t == x then [(xs,t)] else []
```

Next we describe the necessary behaviour of `<|>`. Note that we will be using pseudo code until a Parser type is defined along with instances for Functor, Applicative, and Alternative.

```
let ab = tok 'a' <|> tok 'b'
let aa = tok 'a' <|> tok 'a'
ab "ab" == [("b",'a')]
ab "ba" == [("a",'b')]
ab "aa" == [("a",'a'),("a",'a')]
ab "z"  == []
```

The desired behaviour of `<*>` is described by the following.

```
let aorb   = tok 'a' <|> tok 'b'
let athenb = (,) <$> tok 'a' <*> tok 'b'
let asbs   = (,) <$> aorb <*> aorb
athenb "ba"  == []
athenb "a"   == []
athenb "abc" == [("c",('a','b'))]
asbs "aa" == [("",('a','a'))]
asbs "ba" == [("",('b','a'))]
```

Note that `<*>` still involves Cartesian product. The parse must be resumed after every result given by the previous parser, creating a list of lists of results. For example, the following is a list of results given by some previous parser. The labels are omitted because they are not important for this example; the list of results reduces from the 2-tuple to simply a the left value, which is the remaining input. Ellipses are used to indicate arbitrary remaining elements in the lists.

```
["abc ...", "xyz ...", "mnp ...", ...  ]
```

To resume from these results, the next parser `p` is applied to each remaining input.

```
[p "abc ...", p "xyz ...", p "mnp ...", ...  ]
```

`p` returns its own list of results. Therefore, there is now a list of lists.

```
[["jkl ...", "mno ...", ...], ["pqr" ...], ...]
```

This is problematic type-wise in `pA <*> pB <*> pC`, because if `pA <*> pB` returns a list of lists as shown, then `pC` cannot follow — unless a list of lists of lists is used, but this descends into an infinite type problem. The answer is to flatten the list of lists back into just a list. We borrow the core idea from `carpl`: `foldr interleave [] :: [[a]] -> [a]`. Indeed, we could use `++` instead of `interleave`, because the only practical result list from a parser will

be finite. The decision to use `interleave` is merely because it was used for the sentence and tree generators.

Using `carpl` as an analogy, it is evident that effectively Cartesian product is still being performed by `<*>`.

`fmap` and a type for parsers is defined by the following.

```
data Result a b = Result { label :: b, rest :: [a] }

newtype Parser a b = Parser ([a] -> [Result a b])

instance Functor (Parser a) where
  fmap f (Parser p) = Parser $ (fmap . fmap) f . p
```

The mapping function is applied to each label in the list of `Result` returned by the parser. This depends on Functor for `Result`, which we define in the following along with an instance for `Show`.

```
instance Functor (Result a) where
  fmap f res = res { label = f $ label res }

instance (Show a, Show b) => Show (Result a b) where
  show (Result lbl rst) = '{' : show lbl ++ ","
      ++ show rst ++ "}"
```

The `tok` function changes slightly.

```
tok :: (Eq a) => a -> Parser a a
tok t = Parser p
  where
  p [] = []
  p (x:xs) = if t == x then [Result t xs] else []
```

We also define `resume`, which implements the core of what was described as the behaviour for `<*>`. This is defined separately from `<*>` because it will be reused later.

```
resume :: (b -> c -> d) -> [Result a b] ->
          Parser a c -> [Result a d]
resume f left p = foldr interleave [] . fmap right $ left
  where
  right (Result lbl rst) = (fmap . fmap) (f lbl) $ parse p rst
```

Now, Applicative for `Parser` is easy to define.

```
instance Applicative (Parser a) where
  pure a = Parser $ pure . Result a
  a <*> b = Parser $ \xs -> resume ($) (parse a xs) b
```

The function `parse` is straight-forward, simply taking a `Parser` and returning the contained parsing function.

```
parse :: Parser a b -> [a] -> [Result a b]
parse (Parser p) = p
```

Finally, the Alternative instance for `Parser`.

```
instance Alternative (Parser a) where
  empty = Parser (const [])
  (Parser a) <|> (Parser b) = Parser $ liftA2 interleave a b
```

Note that `liftA2 interleave a b` is equivalent to `xs -> interleave (a xs) (b xs)`.

Using the defined interface, a parser for the following grammar is implemented.

$$\texttt{A} \rightarrow \texttt{aA}$$

```
data A = A0 Char A | A1 Char
  deriving Show

pA = A1 <$> tok 'a' <|> A0 <$> tok 'a' <*> pA
-- >>> :type pA
-- pA :: Parser Char A
-- >>> :type parse pA
-- parse pA :: [Char] -> [Result Char A]
-- >>> parse pA ""
-- []
-- >>> parse pA "xyz"
-- []
-- >>> parse pA "aa"
-- [{A1 'a', "a"},{A0 'a' (A1 'a'), ""}]
-- >> parse pA "aazaa"
-- [{A1 'a', "azaa"},{A0 'a' (A1 'a'), "zaa"}]
```

As expected, `pA` constructs all syntax trees from the start of input to the end, or up to the first unrecognized terminal. A trace is given by the following for `parse pA "aa"` `A1` and `A0` refer to the parsers which recognize the respective production rules for `A`. Instead of `Result`, a tuple is used for brevity. Unimportant details are omitted.

```
pA "aa" == ?
```

1. `<|>` gives `"aa"` to `A1`.
2. `A1` recognizes one `a` and returns `[("a",A1 'a')]`.

3. `<|>` returns `[("a",A1 'a') ...` and gives `"aa"` to `A0`.

4. `A0` recognizes one `a`, returning `[("a", A0 'a')]`, which is passed recursively back as input to `pA`.

```
pA "aa" == [("a",A1 'a'), ... ?
pA "a" == ?
```

5. Having recursed, `<|>` gives `"a"` to `A1`.

6. `A1` recognizes one `a` and returns `[("a", A1 'a')]`.

7. `<|>` returns `[("", A1 'a') ...` and gives `"a"` to `A0`.

8. `A0` recognizes one `a`, returning `[("", A0 'a')]`, which is passed recursively back as input to `pA`.

```
pA "a" == [("", A1 'a'), ... ?
pA "" == ?
```

9. Having recursed, `<|>` gives `""` to `A1`.

10. `A1` fails and returns `[]`.

11. `<|>` returns `[ ...` and gives `""` to `A0`.

12. `A0` fails and returns `[]`.

13. `<|>` returns `[]`.

```
pA "" == []
```

14. (8) gets `[]` from the recursive call. Cartesian product with an empty list is empty, so returns `[]` overall.

```
pA "a" == [("", A1 'a')]
```

15. (4) gets `[("", A1 'a')]`, which becomes `[("", A0 'a' (A1 'a'))]`.

16. (3) gets `[("", A0 'a' (A1 'a'))]` and returns `... ("", A0 'a' (A1 'a'))]`.

```
pA "aa" == [("a",A1 'a'), ("", A0 'a' (A1 'a'))]
```

This interface can also define parsers for middle recursive and ambiguous grammars. However, left recursion poses a problem which we address next.

# 5 Self left recursion

In the strategy we devised for parsing, there is a small but dangerous caveat: in order for the parser to terminate, every production must consume at least one terminal from input before recursing. The input is assumed as finite, and so by repeated removal of terminals, it must empty. Observe the problem with this left recursive parser.

```
A → Aa | b
```

```
data A = A0 A Char | A1 Char
  deriving Show

pA = A1 <$> tok 'b' <|> A0 <$> pA <*> tok 'a'
```

There is no condition under which this parser will terminate. The `A1` parser will fail under `parse pA "x"`, for instance, but `<|>` will only fail if the `A0` parser fails too. For `A0` to fail, it is necessary for `pA` to fail, thus starting the loop over again, because `pA` only fails if `<|>` fails, and so on.

We create a new strategy for left recursion by observing the production `Aa` for `A`. With an `A`, then `Aa` can be used to append an `'a'` to it, creating a new `A`. With the first `A` from `A → b`, `Aa` can be applied again, and so on. With a first `A`, `Aa` can be repeated until it no longer matches the input. We define a new function `</>` to capture this idea, and for a point of reference, redefine `pA`.

```
(</>) :: Parser a b -> Parser a  (b -> b) -> Parser a b
a </> b = Parser $ p . parser a
  where
  p [] = []
  p left = interleave left $ p $ resume (flip ($)) left b
infix 2 </>

pA = A1 <$> tok 'b' </> flip A0 <$> tok 'a'
```

The parser for the A grammar is broken into two distinct parts. All the non-left recursive productions go to the left of `</>` and all the left recursive productions go to the right. The `</>` function gives the left recursive parser the value of `A` that they need, so they are only implemented to parse what comes afterwards.

Note that changing the definition of `A0 A Char` to `A0 Char A` would then not require `flip A0`. This is an aesthetic tradeoff only; the types of the labels has nothing to do with how the underlying parsing works.

A closer look is taken on at how `</>` works for `pA`. First, `</>` calls the parser to the left — in this case it is `A1 <$> tok 'b'`. This returns all productions which start from the beginning of input. The result is either `[]` or just a singleton list with an `A1` in it. Next `p` is evaluated as defined in `</>`. If there are no results, then the parser is done.

Otherwise, `interleave` the results — this gets the results from `A1` into the results of `</>` — with a recursive call to `p`. `p` is given the results resumed with the left recursive parser — in this case `flip A0 <$> tok 'a'`. `p` will continue to recurse, repeatedly applying the left recursive parser until it returns no results, which must happen if the input is finite and the left recursive parser is consuming terminals. Fortunately, the syntax tree comes out correctly, because the last application of the left recursive parser will be on top.

# 6 Mutual left recursion

Given `</>`,parsers for self left recursive grammars can be implemented. There is still one more nuance: mutual left recursion. Consider this grammar and implementation.

$$\texttt{A} \rightarrow \texttt{Ba} \mid \texttt{x}$$
$$\texttt{B} \rightarrow \texttt{Ab} \mid \texttt{y}$$

```
data A = A0 B Char | A1 Char
data B = B0 A Char | B1 Char

pA = A1 <$> tok 'x' </> flip A0 <$> tok 'a'
pB = B1 <$> tok 'y' </> flip B0 <$> tok 'b'
```

This will not compile due to a type error, and is nonsensical anyways. A first attempt to fix `pA` might look like the following.

```
pA = A1 <$> tok 'x' <|> (pB </> flip A0 <$> tok 'a')
```

But this still fails type check and is still nonsensical. The `A0` parser wants a `B`, not an `A`, but even given a `B`, the `A0` parser returns an `A`, and that cannot be merged into a list of `B` — refer again to the `</>` definition — and even if it could, the list would be contaminated with `A` values which, when `</>` loops on the `A0` parser, would be invalid to follow with another `A`, according to the grammar.

We defined `</>` under the assumption of a self cycle such as $\texttt{A} \rightarrow \texttt{A} \rightarrow \texttt{A} \ldots$ which is to effectually say, self left recursion only. With the mutual left recursion between `A` and `B`, the cycle is actually $\texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{A} \rightarrow \texttt{B} \ldots$, or $\texttt{B} \rightarrow \texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{A} \ldots$ depending on whether `pA` or `pB` is chosen for the starting non-terminal. Another function is required to express these cycles which we call `<.`.

```
(<.) :: Parser a (b -> c) -> Parser a (d -> b) -> Parser a (d -> c)
a <. b = Parser $ \xs -> resume (flip (.)) (parse b xs) a
infixr 9 <.
```

This is almost identical to `<*>`. The only difference is the types correspond with function composition `(.) :: (b -> c) -> (a -> b) -> a -> c` rather than function application `($) :: (a -> b) -> a -> b`.

`a <. b` will parse `b`, then parse `a`. A natural result of this kind of composition is that multiple parsers can be chained together. `a <. b <. c` is parse `c`, then `b`, then `a`. If one parser in the chain fails, it all fails. For preference, `.>` is defined as `<.` simply with arguments reversed, as it is arguably clearer in context with the usage.

We now implement `pA` and `pB` properly. The definitions need to share common parts, so the implementation becomes somewhat verbose.

```
leftA :: Parser Char (B -> A)
leftA = flip A0 <$> tok 'a'

rightA :: Parser Char A
rightA = A1 <$> tok 'x'

leftB :: Parser Char (A -> B)
leftB = flip B0 <$> tok 'b'

rightB :: Parser Char B
rightB = B1 <$> tok 'y'

pA = rightA <|> rightB <**> leftA </> leftB .> leftA
pB = rightB <|> rightA <**> leftB </> leftA .> leftB
```

The left recursive parsers are defined as `leftA`, `leftB`, and the non-left recursive parses as `rightA`, `rightB`. Note the type of `leftB .> leftA :: Parser Char (A -> A)`. This is exactly the self cycle `</>` is expecting, and this is done by `leftB` taking the `A`, and `leftA` returning an `A`, with a `B` being passed from `leftB` to `leftA` in the middle (`A` $\rightarrow$ `B` $\rightarrow$ `A`). The same applies to `leftA .> leftB` but cycling on `B` $\rightarrow$ `A` $\rightarrow$ `B`.

The last notable part is `rightB <**> leftA` and the corresponding `rightA <**> leftB`. `<**>` is simply `<*>` with reversed arguments, so it is an aesthetic choice. Consider just the parser for `A`. The left recursive cycle is `A` $\rightarrow$ `B` $\rightarrow$ `A` but, fully, the sequence is `B` $\rightarrow$ `A` $\rightarrow$ `B` $\rightarrow$ `A` .... Particularly, the sentence `"ya"` is in the language, which is a `B` followed by an `A`. The `rightB <**> leftA` production accounts for this, and `rightA <**> leftB` does the same for the `B` parser for the same reasoning.

We have now defined a general context free parser for sensible grammars thereof. For more examples, and additional useful basic parsers other than `tok`, review the full source code associated with this paper.

# 7 Conclusions and extras

Over the course of this paper we have defined three useful programs. We made a generator for all sentences in a language, a generator for all syntax trees in a language, and finally a parser for context free grammars which built on top of those ideas.

Because rigour was not taken in proving the generality of the parser, the reader is cautioned with depending on this. Also, worst case complexities are not proven and have not been investigated beyond a few test grammars — notably poor performers are middle recursion such as `S` $\rightarrow$ `aSa | aa` and ambiguous grammars. However, the objective of this project was simply to think about context free parsing from scratch, and to that, great success was had.

There is some dissatisfaction with simplicity for the programmer if the grammar involves complex mutual left recursion. Because every cycle must be calculated and expressly stated, there may be a greater potential for mistakes, and thus the parser will not recognize some sentences. For an alternative definition, dynamic typing could work, but there was not enough project time to implement this. Nonetheless, a sketch can be given.

```
    A1 <$> tok 'x' *|* B1 <$> tok 'y'
*/* flip A1 <$> tok 'a' *|* flip B1 <$> tok 'b
```

By using dynamic types, we can store `A` and `B` together. The operators `*|*` and `*/*` correspond to `<|>` and `</>` respectively but lift the parsers they are given to a dynamic world. Using this, `.>` or `<.` become unnecessary. `*/*` repeatedly applies both left recursive rules to every `A` and `B` currently collected, just as `</>` does for a single type. However, the dynamic parser will only work when applied to what it is expecting. That is, `A1` only wants a `B`, and so when `*/*` applies it unanimously to an `A`, it will return `Nothing`, indicating to `*/*` that it should move on.

With this mechanism, every `A` that is fed into `B1` will return a `B`, which on the next run will be recognized by `A0`, and vice versa. The cycling between `B` and `A` will happen naturally as an emergent property at runtime. This scales well in simplicity to complex mutual left recursion where there are many cycles, though at the cost of some extra overhead. Naturally, `*/*` can terminate once it makes a full pass and no results were generated.