Ling 185A: Transition parsing project

Dylan Bumford

due: Fri, Mar 22

1 Transition parsing

In class we defined three different incremental parsing strategies: bottom-up, top-down, and left-corner. Each strategy defines a small number of possible **transitions**: acceptable ways of updating a parse table by either consuming part of the current **buffer** (the phrase still waiting to be parsed), or by updating the current **stack** (the sequence of categories representing what is known and/or expected, given what has been parsed so far). Whether a transition can apply or not depends on what's sitting on the stack. In this sense, the stack functions like the **states** of an automaton. For each possible stack configuration, the transition rules of the parser determine which new stack configurations you can reach depending on which buffer symbol you consume next. In this project, you'll have a chance to implement transition-based CFG parsers in Haskell. If you keep the analogy above in mind, these should be similar to the parsers we built for FSAs.

2 Warm up

Before we get going in earnest, here are a couple of warm-up exercises. If you spot the pattern here, it should make writing the `parse` function below straightforward.

```
type Next a = a -> a
```

The type `Next a` is a synonym for functions from `a` to `a`. For instance,

```
plus3 :: Next Int
plus3 n = n + 3

addDot :: Next String
addDot s = s ++ "."

negate :: Next Bool
negate b = not b
```

For any type `a`, the type `Next a` is a Monoid.

```
instance Monoid (Next a) where
  idty = \a -> a
  f <> g = \a -> g (f a)
```

This says the trivial function `idty` from `a` to `a` is the identify function: it simply leaves `a` the way it is. To aggregate two `Next a` functions `f <> g`, you create a new `Next a` function that first does `f` and then does `g` with the result.

The first task is to write a function `until` that when given a stopping condition `done :: a -> Bool` and a stepping function `next :: Next a` builds a routine that just keeps applying `next` until the result satisfies `done`. At that point, the routine should return whatever value it has reached. I'll write this one for you.

```
until :: (a -> Bool) -> Next a -> Next a
until done next = \a -> if done a then a else (next <> until done next) a
```

For instance, let's say we wanted to find the least common multiple of 3 and 17. One way to do this is to start at `0` and just continue adding `3` until the result is divisible by `17`.

```
lcm'17'3 = until divBy17 plus3 0 where
  divBy17 n = n > 0 && n `mod` 17 == 0
```

Or let's say we wanted to build a string that has exactly `n` dots in it. We could start at the empty string and just continue adding dots until the length of the result is `n`.

```
nDots n = until enough addDot "" where
  enough s = length s == n
```

Ok, let's make this more interesting. Imagine the `next` function is **partial**: it only knows what to do with certain kinds of inputs. In Haskell, that means if you feed it an `a` that it doesn't know how to increment, it returns `Nothing`; otherwise it returns `Just` the next value. Functions of this shape have the type defined by `TryNext`.

```
type TryNext a = a -> Maybe a
```

It turns out, functions with this shape are also Monoidal. The trivial function is the one that `Just` returns its input, unchanged. To aggregate two partial increments `f <> g`, you try to run them sequentially, passing the result of `f` into `g`, as above. But if `f` fails (returns `Nothing`) because it doesn't know what to do with the input, then the aggregation should also fail (you can't get through `g` if you can't even get through `f` first).

```
instance Monoid (TryNext a) where
  idty = \a -> Just a
  f <> g = undefined
```

Write a recursive function like `until` above that tries to apply a partial `next` function of type `TryNext a` repeatedly until the result meets some stopping condition `done :: a -> Bool`. If at any point the result of applying `next` to the current value is `Nothing`, then the whole routine should return `Nothing`. Since both `Next a` and `TryNext a` are Monoids, you should be able to write this in a way that is nearly identical to `until` (really: try to do this using `(<>)` instead of any explicit `case` matching).

```
tryUntil :: (a -> Bool) -> TryNext a -> TryNext a
tryUntil done next =
  undefined
```

For instance, imagine you have an incrementing function `next n = 3*n - 1` that multiplies an input value by `3` and subtracts `1` so long as the input is not divisible by `23`. If it is, then `next` doesn't know what to do and returns `Nothing`. You'd like to know what the smallest number over `10000` is that you can get to by repeatedly incrementing this way. But if you step on a multiple of `23` while incrementing, then you bust and return nothing.

```
try'3n'1 = tryUntil enough (\n -> if n `mod` 23 == 0 then Nothing else Just (next n)) 1
 where
  enough n = n > 10000
  next n = 3*n - 1
```

Now imagine the mechanism for incrementing is **nondeterministic**: given an input `a` it returns not a single (determinate) next value, but a *list* of possible next values `[a]`. This is encapsulated in the type `Nexts` below.

```
type Nexts a = a -> [a]
```

You will not be surprised to learn that for any type `a`, the type `Nexts a` is a Monoid. A trivial increment returns just one option: the `a` it is given. To combine two nondeterministic increments `f <> g`, you would like to run `f` first, and then run `g` on the result as usual. The kink is that for any input, `f` will produce a *list* of possible outputs, and you must pass *each* potential output into `g`, and then collect all the results. This should feel quite familiar from automata.

```haskell
instance Monoid (Nexts a) where
  idty = undefined
  f <> g = undefined
```

In fact, the `Nexts` type is a Semiring. Given a list `nexts` of ways that an input value might be incremented, you can create a single summary increment by collecting any result you could get by running any of of the individual `next` functions in the list.

```haskell
instance Semiring (Nexts a) where
  summarize nexts = undefined
```

The following tests whether your definitions of `(<>)` and `summarize` validate the Semiring laws. If you've defined things correctly, then all the results in `srTests` should be `True`.

```haskell
srTests = [srTestL n | n <- [1..10]] ++ [srTestL n | n <- [1..10]] where
  k :: Nexts Int
  k =  \n -> [n*1, n*2, n*3]
  t1 = \n -> [n-2, n-1, n]
  t2 = \n -> [n+1, n+2]
  test n = (==) `on` (sort . ($ n))
  srTestL n = test n (summarize [k <> t1, k <> t2]) (k <> summarize [t1, t2])
  srTestR n = test n (summarize [t1 <> k, t2 <> k]) (summarize [t1, t2] <> k)
```

Finally, write a recursive function `untils` that mirrors `until` and `tryUntil` above. Given a stopping condition `done :: a -> Bool` and nondeterministic function `nexts :: a -> [a]`, it should continue applying `nexts` to the intial input `a`, and then again to all the outputs in `nexts a`, and then again to all the outputs of those outputs, etc.. Whenever it reaches a value that satisfies `done`, it simply returns that value as the only output. Again the code here will likely be almost identical to the earlier `until` variants. (This is the power of algebraic abstraction.)

```haskell
untils :: (a -> Bool) -> Nexts a -> Nexts a
untils done nexts =
  undefined
```

Try it out. Here we look for all of the ways to get from `23` to `42` by adding either `5` or `7` repeatedly.

```haskell
inc'5'7 = untils enough nexts [23] where
  enough (n:ns) = n == 42
  nexts ns = let n = head ns in if n > 42 then [] else [n+5:ns, n+7:ns]
```

3 CFG reminders

Here are the components for building simple Context Free Grammars. Production rules are formed from an inventory of syntactic Categories, enumerated in the data type `Cat`. A `Rule` is either terminal, of the form `Cat :- String`, or branching, of the form `Cat :< [Cat]`. A `CFG` is just a list of such production rules.

```haskell
data Cat = S | D | DP | NP | N | VT | VP | P | PP | WP | W | RC | C | AP
  deriving (Eq, Show, Ord)

data Rule = Cat :- String | Cat :< [Cat]
  deriving (Eq, Show, Ord)

type CFG = [Rule]
```

3.1 Transition table setup

The crucial component of a parsing schema is a **configuration**: a pair of a stack of nonterminals and a buffer of words. Categories are deposited to the stack in one of two forms, `Got Cat` or `Pre Cat`, depending on whether we've already been able to form a constituent of that category based on the words we've consumed (`Got`), or we are expecting to see that category in the words that are coming next (`Pre`).

```
data Chunk = Got Cat | Pre Cat
  deriving (Show,Eq,Ord)

got, pre :: Cat -> Chunk
got c = Got c
pre c = Pre c

type Stack = [Chunk]
type Buffer = [String]
type Config = (Stack, Buffer)
```

A parse table is a list of rows, each containing a transition label (specifying what kind of transition led to that row), a production rule (specifying which rule the transition used), and a configuration. Importantly, we are going to add new rows to the *front* of the list. It's just easier this way. So if you are looking at a table on a piece of paper, the **current configuration** is at the bottom of the table, which will be the first row on the list here.

```
data TLabel = Shift     -- used in bottom-up and left-corner
            | Reduce    -- used in bottom-up
            | Predict   -- used in top-down and left-corner
            | Match     -- used in top-down and left-corner
            | Connect   -- used in left-corner
            | None      -- used only in initial rows of configs
           deriving (Show,Eq)

type Table = [(TLabel, Rule, Config)]

-- convenience function to extract the configuration at the bottom of the table
-- (i.e., front of the list)
currentConfig :: Table -> Config
currentConfig ((_, _, config):_) = config
```

Note that I've defined a function `display` that prints out a list of parse tables to your terminal in a readable format mimicking the versions we've seen in the slides and on the page. In print-out, "barred" categories are represented with a star, so for instance, $\bar{S}$ = `S*`.

Finally, we declare a `Transition` to be a relation on `Table`s. That is, a particular transition takes you from a current table to a list of possible next tables. It will do this by adding a row to the table, spelling out a next possible configuration. The reason it returns a *list* of tables is that there may be more than one way to transition, depending on the stack and the rules of the grammar. This of course is what makes parsing difficult, and leads people to make interesting mistakes. Here we take advantage of the `Nexts` type above.

```
type Transition = Nexts Table
--              = Table -> [Table]
```

This type, `Table -> [Table]`, might remind you of all the automaton functions we've written that end in `State -> [State]`, or `(State, w) -> [(State, w)]`. Each table here (specifically the current stack of that table) is like a state in an automaton. It determines "where you are" in the process of parsing, and where you can go next.

4 Parsing

Write a general purpose `parse` function that takes as arguments a stopping function `done :: Table -> Bool`, and a grammar-dependent transition function `steps` that determines how a table can be extended. It should return a grammar-dependent transition function that expands the parse table using `steps` over and over again, stopping only when the table satisfies `done`. Use `untils`!

```
parse :: (Table -> Bool) -> (CFG -> Transition) -> CFG -> Transition
parse done steps cfg = undefined
```

In the next three sections you'll define the particular transition functions that characterize the different parsing strategies.

4.1 Bottom-up parsing

Remember that bottom-up parsing makes use of two transitions:

> Starting configuration: $(\varepsilon, x_1 \dots x_n)$
> where $x_1 \dots x_n$ is the input
>
> SHIFT step: $(\Phi, x_i x_{i+1} \dots x_n) \Rightarrow (\Phi A, x_{i+1} \dots x_n)$
> where there is a rule $A \rightarrow x_i$ in the grammar
>
> REDUCE step: $(\Phi B_1 \dots B_m, x_i \dots x_n) \Rightarrow (\Phi A, x_i \dots x_n)$
> where there is a rule $A \rightarrow B_1 \dots B_m$ in the grammar
>
> Goal configuration: (A, ε)
> where A is one of the grammar's start symbols

Let's implement these definitions. First, imagine the first word on the buffer is `w` and we've found a production rule `c :- w`. Define a function `shiftConfig` that takes this category `c` and the current `Config` and returns the `Config` that should result after application of SHIFT.

```
shiftConfig :: Cat -> Config -> Config
shiftConfig c (stack, buffer) = undefined
```

Now use this operation to define a `Transition` called `shift` that takes an input table and returns all the tables that you could get by SHIFTing. If the buffer of the input table is empty, then the result will also be empty (there are no SHIFTS left to make). Remember that the result here should be a *list of tables*. Each one will have a new row, labeled with `Shift`, prepended to the front of the incoming table.

```
shift :: CFG -> Transition
shift cfg = \table ->
  let conf = currentConfig table in
  case conf of
    (_, v:vs) -> undefined
    _         -> [ ]
```

Do the same for `reduce`. Imagine we've found a rule `c :< ds` where `ds` matches the last several items on the stack, as required by the REDUCE step. Write a function `reduceConfig` that, given all the stuff on the stack preceding `ds` (this is Φ in the definition above), and the matching category `c` updates the configuration.

```
reduceConfig :: [Chunk] -> Cat -> Config -> Config
reduceConfig phi c (stack, buffer) = undefined
```

Now turn this operation into a proper `Transition` called `reduce`. Given a table, it should give back a list of tables each of which is the result of prepending a `Reduce` row and reducing some symbols a the end of the stack. If there are no production rules that apply to any suffix of symbols, then the result should be the empty list (no reductions possible). Remember, `splits` will give you every possible way of splitting the stack into an initial part and a final part. Also keep in mind that the daughters of a production rule `c :< ds` is a list of `Cat`s. But the stack is a list of `Chunk`s. So to compare the daughters to the stack segments, you'll need to convert the daughters to the relevant kinds of `Chunk`s, or convert the `Chunk`s to plain `Cat`s.

```
reduce :: CFG -> Transition
reduce cfg = \table ->
  let conf = currentConfig table in
  undefined
```

Ok, define a stopping predicate `doneBU` in accordance with the **goal configuration** of the definition above.

```
doneBU :: Table -> Bool
doneBU table = undefined
```

And now define a **summary** `Transition` function that combines the updates that you can get by `shift`ing with those you can get by `reduce`ing (you might want to be thinking of Semirings about now).

```
stepBU :: CFG -> Transition
stepBU cfg = undefined
```

Finally, define a bottom-up parser using the `parse` function together with `doneBU` and `stepBU`.

```
parseBU :: CFG -> Transition
parseBU cfg = undefined
```

4.2 Top-down parsing

We'll now repeat the process for the top-down strategy, defined by:

> Starting configuration: $(\overline{A}, x_1 \dots x_n)$
> where A is one of the grammar's start symbols and $x_1 \dots x_n$ is the input
>
> PREDICT step: $(\overline{A}\,\Phi, x_i \dots x_n) \Rightarrow (\overline{B_1} \dots \overline{B_m}\,\Phi, x_i \dots x_n)$
> where there is a rule $A \rightarrow B_1 \dots B_m$ in the grammar
>
> MATCH step: $(\overline{A}\,\Phi, x_i x_{i+1} \dots x_n) \Rightarrow (\Phi, x_{i+1} \dots x_n)$
> where there is a rule $A \rightarrow x_i$ in the grammar
>
> Goal configuration: $(\varepsilon, \varepsilon)$

Start by defining a function to update a `Config` as indicated in the PREDICT step. That is, assume you've found a rule `c :< ds` where `c` matches the head of the stack. So provided with the `ds` of this rule and the current `Config`, return the PREDICT-updated `Config` (keeping in mind, again, that the daughters `ds` are plain `Cat`s, but the stack contains `Chunk`s).

```
predictConfig :: [Cat] -> Config -> Config
predictConfig ds (stack, buffer) = undefined
```

Spin this up into a full `Transition` on tables.

```
predict :: CFG -> Transition
predict cfg = \table ->
  let conf = currentConfig table in
  case conf of
    (s:ss, buf) -> undefined
    _           -> [ ]
```

Do the same thing for the `match` step. Here it's easy to say how the `Config` should be updated. You will have had to match the had of the stack with the head of the buffer, and the result should be a `Config` where both of these heads are removed, since they've MATCHed.

```
matchConfig :: Config -> Config
matchConfig (stack, buffer) = undefined

match :: CFG -> Transition
match cfg = \table ->
  let conf = currentConfig table in
  case conf of
    (s:ss, v:vs) -> undefined
    _            -> [ ]
```

Define `doneTD` to match tables whose current configuration meets the **goal** state for top-down parsing, and then `stepTD` and `parseTD` just as for bottom-up in the previous section.

```
doneTD :: Table -> Bool
doneTD table = undefined

stepTD :: CFG -> Transition
stepTD cfg = undefined

parseTD :: CFG -> Transition
parseTD cfg = undefined
```

4.3 Left-corner parsing

Do it all again for the left-corner schema, as defined by:

> Starting configuration: $(\overline{A}, x_1 \dots x_n)$
> where A is one of the grammar's start symbols and $x_1 \dots x_n$ is the input
>
> SHIFT step: $(\Phi, x_i x_{i+1} \dots x_n) \Rightarrow (A\Phi, x_{i+1} \dots x_n)$
> where there is a rule $A \rightarrow x_i$ in the grammar
>
> MATCH step: $(\overline{A}\,\Phi, x_i x_{i+1} \dots x_n) \Rightarrow (\Phi, x_{i+1} \dots x_n)$
> where there is a rule $A \rightarrow x_i$ in the grammar
>
> LC-PREDICT step: $(B_1\Phi, x_i \dots x_n) \Rightarrow (\overline{B_2} \dots \overline{B_m}\, A\Phi, x_i \dots x_n)$
> where there is a rule $A \rightarrow B_1 \dots B_m$ in the grammar
>
> LC-CONNECT step: $(B_1\, \overline{A}\, \Phi, x_i \dots x_n) \Rightarrow (\overline{B_2} \dots \overline{B_m}\, \Phi, x_i \dots x_n)$
> where there is a rule $A \rightarrow B_1 \dots B_m$ in the grammar
>
> Goal configuration: $(\varepsilon, \varepsilon)$

First, notice that the MATCH step is truly exactly the same as in top-down parsing. So we can just reuse it here. The SHIFT step is also nearly identical to the bottom-up SHIFT. The only difference is that the new nonterminal is added to the *front* of the stack, rather than the back. So writing the `lcShiftConfig` and `lcShift` functions should be very similar.

```
lcShiftConfig :: Cat -> Config -> Config
lcShiftConfig c (st, buf) = undefined

lcShift :: CFG -> Transition
lcShift cfg = \table ->
  let conf = currentConfig table in
  case conf of
    (_, v:vs) -> undefined
    _         -> [ ]
```

For the LC-PREDICT step, imagine you've found a rule `c :< ds` whose **left corner** matches the first symbol of the stack. Provided with this root `c` and daughters `ds`, update the `Config` according to the definition. And then define a `Transition` to update a table with every LC-PREDICTion possible.

```
lcPredictConfig :: Cat -> [Cat] -> Config -> Config
lcPredictConfig c ds (stack, buffer) = undefined

lcPredict :: CFG -> Transition
lcPredict cfg = \table ->
  let conf = currentConfig table in
  case conf of
    (s:ss, buf) -> undefined
    _           -> []
```

Same deal for the LC-CONNECT step.

```
lcConnectConfig :: Cat -> [Cat] -> Config -> Config
lcConnectConfig c ds (s1:s2:stack, buffer) = undefined

lcConnect :: CFG -> Transition
lcConnect cfg = \table ->
  let conf = currentConfig table in
    case conf of
      (s1:s2:ss, buf) -> undefined
      _               -> []
```

And now define the usual stopping, stepping, and parsing functions, as above.

```
doneLC :: Table -> Bool
doneLC table = undefined

stepLC :: CFG -> Transition
stepLC cfg = undefined

parseLC :: CFG -> Transition
parseLC cfg = undefined
```

4.4 Putting it to work

Try out your functions! Here's a little grammar. Feel free to modify it as you like. (Note that if you add any **left-recursive** rules of the form `X :< [X, ...]`, your top-down parsing functions are likely to not terminate; think about why.)

```
eng :: CFG
eng = [ S  :< [DP, VP]    ,
        S  :< [DP, AP]    , -- AP is a VP with a modifier (to avoid left recursion)
        S  :< [WP, S ]    , -- preposed while-phrases
        WP :< [W , S]     , -- making a while-phrase
        VP :< [VT, DP]    ,
        DP :< [D , NP]    ,
        NP :< [N]         ,
        NP :< [N, PP]     ,
        NP :< [N, RC]     , -- relative clause modification of an N
        PP :< [P , DP]    ,
        AP :< [VP, PP]    ,
        RC :< [C, VP]     , -- subject-gap rel clauses require "that"
        RC :< [DP, VT]    , -- object-gap rel clauses without "that"
        DP :- "Mary"      ,
        DP :- "John"      ,
        DP :- "I"         ,
        VT :- "wrote"     ,
        VT :- "know"      ,
        VT :- "watched"   , -- "watched" is transitive
        VP :- "watched"   , -- "watched" is also intransitive
        VT :- "read"      ,
        VP :- "cried"     ,
        VP :- "awoke"     ,
        VP :- "stinks"    ,
        D  :- "the"       ,
        D  :- "this"      ,
        D  :- "every"     ,
        N  :- "book"      ,
        N  :- "baby"      ,
        N  :- "student"   ,
        P  :- "with"      ,
        C  :- "that"      ,
        W  :- "while"     ]
```

To test a sentence, you'll need an initial `Table`. This table should contain a single row, and the only thing that matters is the `Config` (the initial label and initial rule are completely ignored, so I've just filled them with dummy values). You'll need to make sure to choose the correct configuration for the parsing schema you're testing.

```
startBU, startTD, startLC :: [String] -> Table
startBU sentence = [ (None, S :< [], undefined) ]
startTD sentence = [ (None, S :< [], undefined) ]
startLC sentence = [ (None, S :< [], undefined) ]
```

For instance,

```
s0 = words "Mary read the book with John"
```

```
ghci> display (parseBU eng (startBU s0))

None    , S    -> []      , ([ ], "Mary read the book with John")
Shift   , DP   -> "Mary"  , ([DP], "read the book with John")
Shift   , VT   -> "read"  , ([DP, VT], "the book with John")
Shift   , D    -> "the"   , ([DP, VT, D], "book with John")
Shift   , N    -> "book"  , ([DP, VT, D, N], "with John")
```

```
Shift   , P    -> "with"   , ([DP, VT, D, N, P], "John")
Shift   , DP   -> "John"   , ([DP, VT, D, N, P, DP], "")
Reduce  , PP   -> [P,DP]   , ([DP, VT, D, N, PP], "")
Reduce  , NP   -> [N,PP]   , ([DP, VT, D, NP], "")
Reduce  , DP   -> [D,NP]   , ([DP, VT, DP], "")
Reduce  , VP   -> [VT,DP]  , ([DP, VP], "")
Reduce  , S    -> [DP,VP]  , ([S], "")


None    , S    -> []       , ([ ], "Mary read the book with John")
Shift   , DP   -> "Mary"   , ([DP], "read the book with John")
Shift   , VT   -> "read"   , ([DP, VT], "the book with John")
Shift   , D    -> "the"    , ([DP, VT, D], "book with John")
Shift   , N    -> "book"   , ([DP, VT, D, N], "with John")
Reduce  , NP   -> [N]      , ([DP, VT, D, NP], "with John")
Reduce  , DP   -> [D,NP]   , ([DP, VT, DP], "with John")
Reduce  , VP   -> [VT,DP]  , ([DP, VP], "with John")
Shift   , P    -> "with"   , ([DP, VP, P], "John")
Shift   , DP   -> "John"   , ([DP, VP, P, DP], "")
Reduce  , PP   -> [P,DP]   , ([DP, VP, PP], "")
Reduce  , AP   -> [VP,PP]  , ([DP, AP], "")
Reduce  , S    -> [DP,AP]  , ([S], "")


ghci> display (parseTD eng (startTD s0))

None    , S    -> []       , ([S*], "Mary read the book with John")
Predict , S    -> [DP,VP]  , ([DP*, VP*], "Mary read the book with John")
Match   , DP   -> "Mary"   , ([VP*], "read the book with John")
Predict , VP   -> [VT,DP]  , ([VT*, DP*], "read the book with John")
Match   , VT   -> "read"   , ([DP*], "the book with John")
Predict , DP   -> [D,NP]   , ([D*, NP*], "the book with John")
Match   , D    -> "the"    , ([NP*], "book with John")
Predict , NP   -> [N,PP]   , ([N*, PP*], "book with John")
Match   , N    -> "book"   , ([PP*], "with John")
Predict , PP   -> [P,DP]   , ([P*, DP*], "with John")
Match   , P    -> "with"   , ([DP*], "John")
Match   , DP   -> "John"   , ([ ], "")


None    , S    -> []       , ([S*], "Mary read the book with John")
Predict , S    -> [DP,AP]  , ([DP*, AP*], "Mary read the book with John")
Match   , DP   -> "Mary"   , ([AP*], "read the book with John")
Predict , AP   -> [VP,PP]  , ([VP*, PP*], "read the book with John")
Predict , VP   -> [VT,DP]  , ([VT*, DP*, PP*], "read the book with John")
Match   , VT   -> "read"   , ([DP*, PP*], "the book with John")
Predict , DP   -> [D,NP]   , ([D*, NP*, PP*], "the book with John")
Match   , D    -> "the"    , ([NP*, PP*], "book with John")
Predict , NP   -> [N]      , ([N*, PP*], "book with John")
Match   , N    -> "book"   , ([PP*], "with John")
Predict , PP   -> [P,DP]   , ([P*, DP*], "with John")
Match   , P    -> "with"   , ([DP*], "John")
Match   , DP   -> "John"   , ([ ], "")
```

```
ghci> display (parseLC eng (startLC s0))

None    , S    -> []      , ([S*], "Mary read the book with John")
Shift   , DP   -> "Mary"  , ([DP, S*], "read the book with John")
Connect , S    -> [DP,VP] , ([VP*], "read the book with John")
Shift   , VT   -> "read"  , ([VT, VP*], "the book with John")
Connect , VP   -> [VT,DP] , ([DP*], "the book with John")
Shift   , D    -> "the"   , ([D, DP*], "book with John")
Connect , DP   -> [D,NP]  , ([NP*], "book with John")
Shift   , N    -> "book"  , ([N, NP*], "with John")
Connect , NP   -> [N,PP]  , ([PP*], "with John")
Shift   , P    -> "with"  , ([P, PP*], "John")
Connect , PP   -> [P,DP]  , ([DP*], "John")
Match   , DP   -> "John"  , ([ ], "")


None    , S    -> []      , ([S*], "Mary read the book with John")
Shift   , DP   -> "Mary"  , ([DP, S*], "read the book with John")
Connect , S    -> [DP,AP] , ([AP*], "read the book with John")
Shift   , VT   -> "read"  , ([VT, AP*], "the book with John")
Predict , VP   -> [VT,DP] , ([DP*, VP, AP*], "the book with John")
Shift   , D    -> "the"   , ([D, DP*, VP, AP*], "book with John")
Connect , DP   -> [D,NP]  , ([NP*, VP, AP*], "book with John")
Shift   , N    -> "book"  , ([N, NP*, VP, AP*], "with John")
Connect , NP   -> [N]     , ([VP, AP*], "with John")
Connect , AP   -> [VP,PP] , ([PP*], "with John")
Shift   , P    -> "with"  , ([P, PP*], "John")
Connect , PP   -> [P,DP]  , ([DP*], "John")
Match   , DP   -> "John"  , ([ ], "")
```