Computational Formal Semantics Notes: Part 5

Adrian Brasoveanu

December 3, 2013

Contents
1 Introduction: Basic definitions for parsing 1
2 Recognition, generation and parsing for a simple palindrome grammar 13
3 Intro to parser combinators 15
4 Some basic parser combinators 17
5 Parser combinators 20
6 Simple application of parser combinators: The Palindrome Grammar 23
7 Another example: parsing based on a simple Eng. grammar 25

1 Introduction: Basic definitions for parsing

ghci 1> :l BasicDef

We define parse trees:

ghci 2> :i ParseTree

data ParseTree nonterminal terminal = EmptyTree | Leaf terminal |
Branch nonterminal [ParseTree nonterminal terminal] -- Defined at BasicDef.hs:6:6 instance (Eq nonterminal, Eq terminal, Eq nonterminal, Eq terminal) instance (Show nonterminal, Show terminal) -- Defined at BasicDef.hs:8:10

Here's an example:

ghci 3> let snowwhite = Branch "S" [Branch "NP" [Leaf "Snow White"], Branch "VP" [Branch "TV" [Leaf "loved"], Branch "NP" [Leaf "the dwarfs"]]]

ghci 4> :t snowwhite
snowwhite :: ParseTree [Char] [Char]

Let's examine the value constructors more closely and also how the type parameters of the `ParseTree` type work:

```ghci
ghci 5> snowwhite
[S::[NP::"Snow White"] [VP::[TV::"loved"] [NP::"the dwarfs"]]]
```

We also have a convenience function that retrieves the label / category of the root node of a parse tree, if that label exists:

```ghci
ghci 12> :t nodeLabel
nodeLabel :: ParseTree nonterminal terminal → Maybe nonterminal
```

```ghci
ghci 13> nodeLabel (Branch "NP" [Leaf "Snow White"])
Just "NP"
```

```ghci
ghci 14> nodeLabel (EmptyTree)
Nothing
```
We can index the positions / nodes in a tree so that we can define various relations between them: dominance, c-command, precedes etc.

And we can identify the sub-trees at various positions:
Let’s understand the position function a bit better. This is its definition:

```haskell
ghci 25> subtrees snowwhite
[Just ["S":"NP":"Snow White"] ["VP":"TV":"loved"] ["NP":"the dwarfs"]],
Just ["NP":"Snow White",Just "Snow White",Just ["VP":"TV":"loved"] ["NP":
"the dwarfs"]],Just ["TV":"loved",Just "loved",Just ["NP":"the dwarfs"],
Just "the dwarfs"]
```

```haskell
ghci 26> length $ subtrees snowwhite
8
```

Let’s understand the position function a bit better. This is its definition:

```haskell
type Pos = [Int]
pos :: ParseTree nonterminal terminal → [Pos]
pos EmptyTree = [[]]
pos (Leaf _) = [[]]
pos (Branch _ ts) = [] : [i:p | (i,t) ← zip [0..] ts, p ← pos t]
```

And now let’s looks at a couple of examples of increasing complexity.

```haskell
ghci 27> let tree1 = Leaf "Snow White"
```

```haskell
ghci 28> tree1
"Snow White"
```

Think about how the pos function applied to tree1 gets evaluated, exactly. And do the same for all subsequent examples.

```haskell
ghci 29> pos tree1
[]
```

```haskell
ghci 30> zip [0..] []
[]
```

```haskell
ghci 31> zip [0..] ['a']
[0,'a']
```

```haskell
ghci 32> zip [0..] ['a','b']
[(0,'a'),(1,'b')]
```

```haskell
ghci 33> let tree2 = Branch "NP" [Leaf "Snow White"]
```
ghci 34> tree2
  ["NP":"Snow White"]

ghci 35> pos tree2
  [[],[0]]

ghci 36> let tree3 = Branch "S" [Leaf "Snow White", Leaf "left"]

ghci 37> tree3
  ["S":"Snow White" "left"]

ghci 38> pos tree3
  [[],[0],[1]]

ghci 39> let tree4 = Branch "S" [Leaf "Snow White", Leaf "left", Leaf "early"]

ghci 40> tree4
  ["S":"Snow White" "left" "early"]

ghci 41> pos tree4
  [[],[0],[1],[2]]

ghci 42> nodeLabel tree4
  Just "S"

ghci 43> let tree5 = Branch "S" [Leaf "Snow White", Leaf "left", Branch "AdvP" [Leaf "early", Leaf "and", Leaf "quickly"]]

ghci 44> tree5
  ["S":"Snow White" "left" ["AdvP":"early" "and" "quickly"]]
Now let’s examine the definition of the \texttt{subtree} function more closely:

\begin{verbatim}
subtree :: ParseTree nonterminal terminal → Pos → Maybe (ParseTree nonterminal terminal)
subtree t [] = Just t
subtree t@(Branch _ ts) p@(i:is) = if p ∈ (pos t) then (subtree ts !! i) is else Nothing
subtree (Leaf _) (i:is) = Nothing

subtrees :: ParseTree nonterminal terminal → [Maybe (ParseTree nonterminal terminal)]
subtrees t = [subtree t p | p ← pos t]
\end{verbatim}
ghci 54> subtree tree2 []
  Just ["NP":"Snow White"]

ghci 55> subtree tree2 [0]
  Just "Snow White"

ghci 56> subtree tree2 [1]
  Nothing

ghci 57> subtree tree2 [0,0]
  Nothing

ghci 58> tree4
  ["S":"Snow White" "left" "early"]

ghci 59> :t tree4
  tree4 :: ParseTree [Char] [Char]

ghci 60> pos tree4
  [[],[0],[1],[2]]

ghci 61> subtree tree4 []
  Just ["S":"Snow White" "left" "early"]

ghci 62> subtree tree4 [0]
  Just "Snow White"

ghci 63> subtree tree4 [1]
  Just "left"

ghci 64> subtree tree4 [2]
  Just "early"
We also define the usual structural relations over nodes in the tree: dominance, sisterhood, c-command, branching nodes, precedence.
We also define a function that splits the incoming input string into 2 or more substrings. Think of the string as a list, in the simplest case a list of characters, but this could very well be a list of words or a list of syntactic categories. We want to non-deterministically explore all the possible partitions of an input list into sublists because we don’t know ahead of time which grammar rule will apply to the initial sublist, or the first two sublists etc.

Here’s the definition of the function that splits a list into 2 sublists:
split2 :: [a] → [[[a], [a]]]

split2 [] = [[[]]]

split2 (x : xs) = [[[], (x : xs)]] ++ (map (λ(y, zs) → ((x : y), zs)) (split2 xs))

And here's the definition of the function that splits a list into \( n \) sublists:

\[
\text{splitN} :: \text{Int} → [a] → [[[a]]]
\]

\[
\text{splitN} n xs
| n \leq 1 = \text{error} "cannot split"
| n \equiv 2 = [[ys, zs] | (ys, zs) ← \text{split2} xs]
| \text{otherwise} = [ys : rs | (ys, zs) ← \text{split2} xs, rs ← \text{splitN} (n - 1) zs]
\]

And now we define a function that generates all finite strings of length \( n \) defined over an input alphabet:
We use this function to define a generator for the infinite set of all finite strings over an input alphabet:
gener′ :: Int → String → [String]
gener′ n alphabet = gener n alphabet ++ gener′ (n + 1) alphabet

generateAll :: String → [String]
generateAll alphabet = gener′ 0 alphabet

ghci 100> take 100 $ generateAll "abc"
["","a","b","c","aa","ab","ac","ba","bb","bc","ca","cb","cc","aaa","aab","aac",
 "aba","abb","abc","aca","acb","acc","baa","bab","bac","bba","bbb","bbc","bca",
 "bcb","bcc","caa","cab","cac","cba","cbb","cbc","cca","ccb","ccc","aaaa",
 "aaab","aaac","aaba","aabb","aacb","aacc","abaa","abab","abac",
 "abba","abbb","abbc","abca","abcb","abcc","aaca","acab","acac","acba","acb",
 "acbc","accc","accc","baaa","baab","babc","bab","babc","bcac","bacc",
 "bcba","bcbb","bcbc","bcca","bcdb","bccc","cbac","bcab","bcba","bcbb","bcbc",
 "bcda","bcab","bcac","bcba","bcbb","bcbc","bcca","bcdb","bccc","cbac","cbab",
 "cbac","caba","cabb","cabc"]
2 Recognition, generation and parsing for a simple palindrome grammar

We can put the last couple of notions to work by defining a recognizer, generator and parser for a simple grammar for palindromes:
:l Palindromes

recognize :: String → Bool

recognize "aa"  
True

recognize "aba"  
True

recognize "ab"  
False

recognize "abca"  
False

generate :: [String]

take 100 generate

parse :: String → [ParseTree String String]
There are 2 basic insights behind the parser-combinator approach to parsing.

The first idea is that each basic parser ‘combinator’ defines a binary relation over parse states – much like an accessibility relation over possible worlds in modal logic or like a relation between an input and an output state in dynamic logic.

And in a parallel way, the binary relations over parse states denoted by parser combinators enable us to travel through the space of parse states searching for a parse tree for our input sentence. Just as in modal logic (as opposed to classical logic), our perspective on the space of parse states is local and we travel through them one step at a time along the paths provided to us by the accessibility relations denoted by parser combinators.

The second idea is compositionality of parsers: we can compose basic parsers together to obtain more complex parsers. The result is that we can compositionally assemble a parser for a grammar in a way that closely mirrors the grammar: the grammar gives us the way in which we should compose the parsers togethers.

There are 2 basic ways of combining parsers, as expected given the fact that they basically denote binary accessibility relations over parse states:

- union: we take the union of two binary relations; this is the choice of forming an NP out of a bare N or out of an Adj and an N, for example
- relation composition: we (sequentially) compose two binary relations by taking a step through the parse-state space along the first relation and another one along the second relation; this is how we formalize that we can parse a sentence S if we first parse an NP and then a VP

The grammar provides a compositionally-assembled declarative statement about what counts as a grammatical structure in the language. The corresponding parser provides a compositionally-assembled (in a way that strictly follows the grammar composition) procedural statement about what counts as a sequence of actions / steps to build a grammatical structure for an input sentence in the language.

We define two types that will be essential for all our subsequent definitions:
A parse state is a list of pairs, each pair consisting of the partial parse that has been assembled up to that point and the remaining list of inputs that still need to be parsed.

This type just captures the idea that parsing is really traveling through the space of parse states in a way regulated by the grammar based on which we parse. We define a parse state as a list of \((\text{partialParse}, [\text{remainingInput}])\) pairs because we might have a locally or globally ambiguous sentence with multiple partial (for local ambiguity) or final (for global ambiguity) parses.

A parser is just a function from a list of inputs to a parse state, i.e., to a list of \((\text{partial}−\text{parse}, \text{remaining}−\text{inputs})\) pairs. The type does not reflect the ‘binary accessibility relation’ idea directly; for that, it would have had to be a relation between parse states, i.e., a relation between input \((\text{partial}−\text{parse}, \text{remaining}−\text{inputs})\) pairs and output \((\text{partial}−\text{parse}, \text{remaining}−\text{inputs})\) pairs, where the output \text{partial}−\text{parse} is usually bigger than the input one and the output list of \text{remaining}−\text{inputs} is usually smaller.

Instead of this, we discard the \text{partial}−\text{parse} component of the input pair and simply focus on the list of inputs that are fed into the parser. This is because we take the partial parse and deal with it separately (this is a reflection of the monadic style of programming typical of Haskell). But the output is as expected: it’s a list of \((\text{partial}−\text{parse}, \text{remaining}−\text{inputs})\) pairs.

Despite this apparent mismatch between the intuitive characterization of parsers as ‘accessibility relations over parse states’ and their actual type, the intuitive characterize is still formally correct. But we capture it in a slightly different way because of the ‘monadic plumbing’ we effectively assume under the surface. We don’t have to worry about this, just stick to the intuition and remember that the mismatch between types and intuition is just a minor distraction.

As a first pass, we treat parsing as a problem of segmenting strings, more generally, sequences of of words, and grouping smaller sequences together into larger sequences. But (generative) grammars are not about flat sequences / strings, they are about trees (hierarchical structures) and assembling / composing bigger trees out of smaller trees. The lexicon provides the elementary trees and the grammar rules tell us how to build larger trees from smaller trees.

An example grammar from the Parsing chapter of the Computational Semantics textbook is given below:

LEXICON:

\[
\begin{align*}
\text{NP} & \rightarrow \text{Alice} | \text{Dorothy} \\
\text{VP} & \rightarrow \text{smiled} | \text{laughed} \\
\text{D} & \rightarrow \text{every} | \text{some} | \text{no} \\
\text{N} & \rightarrow \text{dwarf} | \text{wizard}
\end{align*}
\]

PHRASE STRUCTURE RULES:

\[
\begin{align*}
\text{S} & \rightarrow \text{NP VP} \\
\text{NP} & \rightarrow \text{D N}
\end{align*}
\]

The basic idea behind the compositionality of parser combinators is that we should assemble parsers based on a lexicon and a grammar just as trees are generated compositionally based on a lexicon and a grammar. We do not segment sequences / strings and assemble larger sequence / strings out of smaller ones. Instead, we assemble larger parsers out of smaller parsers. In particular, we will have basic parsers for the lexicon and parser combinators corresponding to the phrase structure rules.
4 Some basic parser combinators

`succeed`: takes a particular parse and irrespective of the input, declares the input successfully parsed as the parse it took as an argument.

\[
\text{succeed} :: \text{parse} \rightarrow \text{Parser input parse} \\
succeed \text{ finalParse remainingInputs} = [(\text{finalParse}, \text{remainingInputs})]
\]

```ghci
ghci 118> :t succeed
succeed :: parse \to Parser input parse
```

```ghci
ghci 119> :t succeed "PARSING DONE"
succeed "PARSING DONE" :: Parser input [Char]
```

```ghci
ghci 120> :t succeed "PARSING DONE" "some input here"
succeed "PARSING DONE" "some input here" :: ParseState [Char] Char
```

```ghci
ghci 121> succeed "PARSING DONE" "some input here"
[("PARSING DONE","some input here")]
```

`failp`: this is the parser that always fails for any input that it takes as argument. The empty set of parse states \([\ ]\) is returned for every input:

\[
\text{failp} :: \text{Parser input parse} \\
\text{failp remainingInputs} = [\]
\]

```ghci
ghci 122> :t failp
failp :: Parser input parse
```

```ghci
ghci 123> :t failp "some input here"
failp "some input here" :: ParseState parse Char
```

```ghci
ghci 124> failp "some input here"
[\]
```

We can parse characters / symbols (terminals) by having parsers specialized for each of them:

\[
\text{symbol} :: \text{Eq terminal} \Rightarrow \text{terminal} \rightarrow \text{Parser terminal terminal} \\
\text{symbol} \ t [\ ] = [\] \\
\text{symbol} \ t (x:xs) \mid t \equiv x = [(t,xs)] \\
\mid \text{otherwise} = [\]
\]
We can parse a sequence of characters / symbols at a time by specifying a parser for lists of terminals rather than simply terminals:

\[
\text{token :: Eq terminal} \Rightarrow [\text{terminal}] \rightarrow \text{Parser terminal [terminal]}
\]
\[
\text{token ts xs} \mid \text{ts \equiv take n xs} = [(\text{ts}, \text{drop n xs})] \\
\text{otherwise} = []
\]
\[
\text{where } n = \text{length ts}
\]
Finally, we can also define ‘looser’ parsers that parse an input if it satisfies a predicate (this is what the function \texttt{reads} in the \texttt{Prelude} actually is).

\[
\begin{align*}
\text{\texttt{satisfy}} & : (\text{input} \rightarrow \text{Bool}) \rightarrow \text{Parser input input} \\
\text{\texttt{satisfy}}\ p\ [\ ] & = [\ ] \\
\text{\texttt{satisfy}}\ p\ (i:\ \text{is}) & \mid p\ i = [\ (i,\ \text{is})] \\
& \mid \text{otherwise} = [\ ] \\
\text{\texttt{digit}} & : \text{Parser Char Char} \\
\text{\texttt{digit}} & = \text{\texttt{satisfy}}\ \text{isDigit} \\
\end{align*}
\]

\texttt{ghci 142}:

\texttt{digit "1a"} \\
\[
[\ (\ '1',\ "a") ]
\]
Finally, we define a parser ‘modifier’ / ‘restrictor’: just.
just is a function from parsers to parsers that restricts a parser to final inputs, i.e., we can parse those inputs only if the remaining list of inputs is empty

\[
\text{just} :: \text{Parser input parse} \rightarrow \text{Parser input parse}
\]
\[
\text{just } p = \text{filter } (\text{null } \circ \text{snd}) \circ p
\]

5 Parser combinators

just can actually be thought of as a unary parser combinator: it takes a parser as an argument and returns another parser.

We turn now to binary (or higher arity) parser combinators, i.e., functions that take multiple parsers and assemble them into one single parser. While the basic parser combinators in the previous section basically correspond to lexical rules in a generative grammar, these higher-arity parser combinators correspond to the phrase structure rules.

In particular, just as we have ‘choice’, i.e., optional / non-deterministic rewrite rules (e.g., \( VP \rightarrow V \ NP \mid V \ NP \ PP \)), we have a parser combinator that takes two parsers and parses an input if at least one of the two parsers parses the input.

\[
\text{infixr } 4 \langle | \rangle
\]
\[
\langle | \rangle :: \text{Parser input parse} \rightarrow \text{Parser input parse} \rightarrow \text{Parser input parse}
\]
\[
\langle p1 \langle | \rangle p2 \rangle xs = p1 \hspace{1em} xs + p2 \hspace{1em} xs
\]

\[
\text{ghci 149} > t (\langle | \rangle)
\]
\[
\langle | \rangle :: \text{Parser input parse} \rightarrow \text{Parser input parse} \rightarrow \text{Parser input parse}
\]
We also want to be able to sequence 2 parsers. For examples, to parse a sentence according to this rule \( S \to NP \ VP \), we first want to parse an \( NP \), then a \( VP \).
A simple example is a parser that parses "a" first, and then "b":

```ghci 161> let tokenAthenB = (token "a") < *> (token "b")

ghci 162> tokenAthenB "abc"
[ ["ab", "c"] ]

ghci 163> tokenAthenB "bac"
[]

ghci 164> tokenAthenB "cab"
[]

ghci 165> let tokenBthenA = (token "b") < *> (token "a")

ghci 166> tokenBthenA "abc"
[]

ghci 167> tokenBthenA "bac"
[ ["ba", "c"] ]
```

Finally, we define an operator that enables us to apply functions to the parses produced by a parser. The reason for this is that the parsers we want to build are parsers that take strings as input and output trees, not simply strings.

```infixl 7 <$>```

```(<$>) :: (input → parse) → Parser s input → Parser s parse
(f <$> p) xs = [(f x, ys) | (x, ys) ← p xs]```

A simple example is a function that parses / identifies digits in strings and then turns them into integers:

```ghci 168> :i ord

ord :: Char → Int -- Defined in 'GHC.Base'
```
6 Simple application of parser combinators: The Palindrome Grammar

We first specialize our general `Parser` type to a parser that outputs trees:

Given this specialized `PARSER` type, we define a `succeed` parser for the empty tree:

We also define a `PARSER` that will build a `Leaf` sub-tree on top of any basic symbol:
symbolT :: Eq input ⇒ input → PARSER input category
symbolT s = (λx → Leaf x) < $ > symbol s

ghci 177> symbol 'a' "abc"
[(('a','bc')]

ghci 178> symbol 'a' "abc"
[(('a','bc')]

Our palindrome parser can now be stated very concisely as follows:

ghci 179> let {palindrome :: PARSER Char String;
   palindrome = epsilonT <|>
   parseAs "A-pair" [symbolT 'a', palindrome, symbolT 'a'] <|>
   parseAs "B-pair" [symbolT 'b', palindrome, symbolT 'b'] <|>
   parseAs "C-pair" [symbolT 'c', palindrome, symbolT 'c'] }

Compare this with our previous definition of the parser

parse :: String → [ParseTree String String]
parse = λxs →
  [EmptyTree | null xs] ++
  [Leaf "a" | xs ≡ "a"] ++
  [Leaf "b" | xs ≡ "b"] ++
  [Leaf "c" | xs ≡ "c"] ++
  [Branch "A-pair" [Leaf "a", t, Leaf "a"] | ["a", ys, "a"] ← splitN 3 xs, t ← parse ys] ++
  [Branch "B-pair" [Leaf "b", t, Leaf "b"] | ["b", ys, "b"] ← splitN 3 xs, t ← parse ys] ++
  [Branch "C-pair" [Leaf "c", t, Leaf "c"] | ["c", ys, "c"] ← splitN 3 xs, t ← parse ys]

ghci 180> palindrome "aa"
[("aa"),("a","a"),(["A-pair" : 'a' 'a'],""))]

ghci 181> palindrome "abba"
[("abba"),("a","bba"),(["A-pair" : 'a' ["B-pair" : 'b' 'b'] 'a'],""))]

ghci 182> palindrome "abccba"
[("abccba"),("a","bcba"),(["A-pair" : 'a' ["B-pair" : 'b' 'b'] 'a'],""))]

ghci 183> palindrome "abcacba"
[("abcacba"),("a","bcacba"),(["A-pair" : 'a' ["B-pair" : 'b' 'b'] 'a'],""))]

24
Another example: parsing based on a simple Eng. grammar

We build a parser for the very simple Eng. fragment we mentioned above:

LEXICON:
NP → Alice | Dorothy
VP → smiled | laughed
D → every | some | no
N → dwarf | wizard

PHRASE STRUCTURE RULES:
S → NP VP
NP → D N

We will now define a parser for this grammar. A first simple attempt is this:

ghci 186> let { pS, pNP, pVP, pD, pN :: Parser String String;
   pS = pNP <*> pVP;
   pNP = symbol "Alice" < | > symbol "Dorothy" < | > (pD <*> pN);
   pVP = symbol "smiled" < | > symbol "laughed";
   pD = symbol "every" < | > symbol "some" < | > symbol "no";
   pN = symbol "dwarf" < | > symbol "wizard" }

ghci 187> pS ["Alice","smiled"]
   [(["Alicesmiled",[]])]

ghci 188> pS ["no","wizard","smiled"]
   [(["nowizardsmiled",[]])]

Once again, we need post-processing for the parser output. But: we need post-processing for sequential compositions of parsers, not just for single parsers. For example, a parser like pNP <*> pVP should output a binary branching node with the results of the parsers pNP and pVP as its two daughter nodes.

In the general case, the rhs of a rule can be any list of terminals and nonterminals, so we need to be able to decompose such lists. The previous version of parser composition was composing parsers of the same kind

\[
(< \ast >) \mapsto \text{Parser input [parse]} \to \text{Parser input [parse]} \to \text{Parser input [parse]}
\]
\[
(p \ast q) \mapsto [(r1 \oplus r2, zs) \mid (r1, ys) \leftarrow p x, (r2, zs) \leftarrow q ys]
\]
We switch to a version of sequential composition of parsers that can accommodate lists of more than 2 categories on the rhs. In particular, the first parser outputs a category of some sort (can be a list etc.), while the second parser outputs a list of such categories (list of lists etc.). That is, the second parser is the composition of all the remaining parsers, whatever they are, that are needed for the rhs of the corresponding grammar rule. The result is a parser of the more general kind, i.e., a parser that outputs a list of categories.

```
infixl 6 <::>
(<::> :: Parser input category → Parser input [category] → Parser input [category]
(p <::> q) xs = [(r:rs,zs) | (r,ys) <- p xs, (rs,zs) <- q ys]
```

We use the parser composition function `<::>` to define a function that collects the results of a list of parsers operating one after the other:

```
collect :: [Parser input category] → Parser input [category]
collect [] = succeed []
collect (p : ps) = p <::> collect ps
```

Instead of defining a sentence parser as `pNP < * > pVP` as we did above, we can now define it as `collect [pNP, pVP]`:

```
ghci 189> let {pS = collect [pNP,pVP];
    pNP = symbol "Alice" <|> symbol "Dorothy";
    pVP = symbol "smiled" <|> symbol "laughed"}

ghci 190> pS ["Alice","smiled"]
     [[("Alice","smiled"),[]]]
```

Finally, we also want to construct a parse tree based on the list of outputs we get after collecting multiple parsers. For that, we use a parser `parseAs`, which builds and labels non-leaf nodes. The parser `parseAs` takes as its first argument a nonterminal (the category label), and as its second argument a list of parsers.

For example, to handle a rule `S → NP VP`, the function `parseAs` takes "S" as its label and assumes that a list of parsers for NPs and VPs has been constructed

```
parseAs :: category → [PARSER input category] → PARSER input category
parseAs label ps = (λxs → Branch label xs) <$> collect ps
```

We have to change the basic parsers also from `symbol` (parsers that output strings) to `symbolT` (parsers that output trees, in particular, leaves). So if we want to parse a sentence, we specify the parser as:

```
ghci 191> let {pS = parseAs "S" [pNP,pVP];
   pNP = symbolT "Alice" <|> symbolT "Dorothy";
   pVP = symbolT "smiled" <|> symbolT "laughed"}

ghci 192> pS ["Alice","smiled"]
     [[("S":"Alice" "smiled"),[]]]
```
The full parser specification is this:

```hs
let { pS, pNP, pVP, pD, pN :: PARSER String String;
    pS = parseAs "S" [pNP, pVP];
    pNP = symbolT "Alice" <| symbolT "Dorothy" <| > parseAs "NP" [pD, pN];
    pVP = symbolT "smiled" <| > symbolT "laughed";
    pD = symbolT "every" <| > symbolT "some" <| > symbolT "no";
    pN = symbolT "man" <| > symbolT "woman" }
```

```hs
ghci 194> pS ["Alice","smiled"]
[(["S":["Alice" "smiled"],[]])]
```

```hs
ghci 195> pS ["no","woman","smiled"]
[(["S":["NP":["no" "woman"] "smiled"],[]])]
```