Computational Formal Semantics Notes: Part 5

Adrian Brasoveanu*

December 3, 2013

Contents

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
```

Here's an example:

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

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

*Code based on *Computational Semantics with Functional Programming* by Jan van Eijck & Christina Unger, http://www.computational-semantics.eu.

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

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

```
ghci 6> :t EmptyTree
  EmptyTree :: ParseTree nonterminal terminal
```

```
ghci 7> :t Leaf
  Leaf :: terminal → ParseTree nonterminal terminal
```

```
ghci 8> :t Leaf "Snow White"
  Leaf "Snow White" :: ParseTree nonterminal [Char]
```

```
ghci 9> :t Branch
  Branch :: nonterminal → [ParseTree nonterminal terminal] → ParseTree nonterminal terminal
```

```
ghci 10> :t Branch "NP"
  Branch "NP" :: [ParseTree [Char] terminal] → ParseTree [Char] terminal
```

```
ghci 11> :t Branch "NP" [Leaf "Snow White"]
  Branch "NP" [Leaf "Snow White"] :: ParseTree [Char] [Char]
```

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

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

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

```
ghci 14> nodeLabel (EmptyTree)
  Nothing
```

```
ghci 15> nodeLabel (Leaf "Snow White")
    Nothing
```

We can index the positions / nodes in a tree so that we can define various relations between them: dominance, c-command, precedes etc.

```
ghci 16> pos snowwhite
    [[],[0],[0,0],[1],[1,0],[1,0,0],[1,1],[1,1,0]]
```

And we can identify the sub-trees at various positions:

```
ghci 17> subtree snowwhite []
    Just ["S":["NP":"Snow White"] ["VP":["TV":"loved"] ["NP":"the dwarfs"]]]
```

```
ghci 18> subtree snowwhite [0]
    Just ["NP":"Snow White"]
```

```
ghci 19> subtree snowwhite [0,0]
    Just "Snow White"
```

```
ghci 20> subtree snowwhite [1]
    Just ["VP":["TV":"loved"] ["NP":"the dwarfs"]]
```

```
ghci 21> subtree snowwhite [2]
    Nothing
```

```
ghci 22> subtree snowwhite [1,0]
    Just ["TV":"loved"]
```

```
ghci 23> subtree snowwhite [1,1]
    Just ["NP":"the dwarfs"]
```

```
ghci 24> subtree snowwhite [1,1,1]
    Nothing
```

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"`$]$

ghci 26> *length* $ *subtrees snowwhite*
8

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

$\textbf{type}\ Pos = [Int]$
$pos :: ParseTree\ nonterminal\ terminal \rightarrow [Pos]$
$pos\ EmptyTree = [[\,]]$
$pos\ (Leaf\ \_) = [[\,]]$
$pos\ (Branch\ \_\ ts) = [\,] : [i : p \mid (i, t) \leftarrow zip\ [0\,..]\ ts, p \leftarrow pos\ t]$

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

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

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.

ghci 29> *pos tree1*
$[[\,]]$

ghci 30> $zip\ [0\,..]\ [\,]$
$[\,]$

ghci 31> $zip\ [0\,..]$ `['a']`
`[(0, 'a')]`

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

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 "q
```

```
ghci 44> tree5
  ["S":"Snow White" "left" ["AdvP":"early" "and" "quickly"]]
```

ghci 45> *pos tree5*
$[[\,], [0], [1], [2], [2,0], [2,1], [2,2]]$

Now let's examine the definition of the *subtree* function more closely:

$$
\begin{aligned}
&\mathit{subtree} :: \mathit{ParseTree\ nonterminal\ terminal} \rightarrow \mathit{Pos} \rightarrow \mathit{Maybe}\ (\mathit{ParseTree\ nonterminal\ terminal}) \\
&\mathit{subtree}\ t\ [\,] = \mathit{Just}\ t \\
&\mathit{subtree}\ t@(\mathit{Branch}\ \_\ \mathit{ts})\ p@(i : \mathit{is}) = \mathbf{if}\ p \in (\mathit{pos}\ t)\ \mathbf{then}\ (\mathit{subtree}\ (\mathit{ts}\ !!\ i)\ \mathit{is})\ \mathbf{else}\ \mathit{Nothing} \\
&\mathit{subtree}\ (\mathit{Leaf}\ \_)\ (i : \mathit{is}) = \mathit{Nothing} \\
&\mathit{subtree}\ (\mathit{EmptyTree})\ (i : \mathit{is}) = \mathit{Nothing}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{subtrees} :: \mathit{ParseTree\ nonterminal\ terminal} \rightarrow [\mathit{Maybe}\ (\mathit{ParseTree\ nonterminal\ terminal})] \\
&\mathit{subtrees}\ t = [\mathit{subtree}\ t\ p \mid p \leftarrow \mathit{pos}\ t]
\end{aligned}
$$

ghci 46> *tree1*
`"Snow White"`

ghci 47> *:t tree1*
tree1 :: *ParseTree nonterminal* [*Char*]

ghci 48> *pos tree1*
$[[\,]]$

ghci 49> *subtree tree1* [ ]
Just `"Snow White"`

ghci 50> *subtree tree1* [0]
Nothing

ghci 51> *tree2*
[`"NP"` : `"Snow White"`]

ghci 52> *:t tree2*
tree2 :: *ParseTree* [*Char*] [*Char*]

ghci 53> *pos tree2*
$[[\,], [0]]$

```
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"
```

ghci 65> *subtree tree4* [3]
Nothing

ghci 66> *subtree tree4* [0,0]
Nothing

ghci 67> *subtrees tree1*
[*Just* "Snow White"]

ghci 68> *subtrees tree2*
[*Just* ["NP" : "Snow White"], *Just* "Snow White"]

ghci 69> *subtrees tree4*
[*Just* ["S" : "Snow White" "left" "early"], *Just* "Snow White", *Just* "left", *Just* "early"]

We also define the usual structural relations over nodes in the tree: dominance, sisterhood, c-command, branching nodes, precedence.

ghci 70> **let** *snowwhite* = *Branch* "S" [*Branch* "NP" [*Leaf* "Snow White"], *Branch* "VP" [*Branch* "TV" [*Leaf* "loved"],

ghci 71> *snowwhite*
["S" : ["NP" : "Snow White"] ["VP" : ["TV" : "loved"] ["NP" : "the dwarfs"]]]

ghci 72> *properDominance snowwhite*
[([], [0]), ([], [0,0]), ([], [1]), ([], [1,0]), ([], [1,0,0]), ([], [1,1]), ([], [1,1,0]), ([0], [0,0]), ([1], [1,0]), ([1], [1,0,0]), ([1], [1,1]), ([1], [1,1,0]), ([1,0], [1,0,0]), ([1,1], [1,1,0])]

ghci 73> *nodeProperDominance snowwhite*
[("S", "NP"), ("S", "VP"), ("S", "TV"), ("S", "NP"), ("VP", "TV"), ("VP", "NP")]

ghci 74> *dominance snowwhite*
[([], []), ([], [0]), ([], [0,0]), ([], [1]), ([], [1,0]), ([], [1,0,0]), ([], [1,1]), ([], [1,1,0]), ([0], [0]), ([0], [0,0]), ([0,0], [0,0]), ([1], [1]), ([1], [1,0]), ([1], [1,0,0]), ([1], [1,1]), ([1], [1,1,0]), ([1,0], [1,0]), ([1,0], [1,0,0]), ([1,0,0], [1,0,0]), ([1,1], [1,1]), ([1,1], [1,1,0]), ([1,1,0], [1,1,0])]

ghci 75> *nodeDominance snowwhite*

```
[("S","S"),("S","NP"),("S","VP"),("S","TV"),("S","NP"),("NP","NP"),("VP","VP"),
("VP","TV"),("VP","NP"),("TV","TV"),("NP","NP")]
```

ghci 76> *sisterhood snowwhite*

```
[([0],[1]),([1],[0]),([1,0],[1,1]),([1,1],[1,0])]
```

ghci 77> *nodeSisterhood snowwhite*

```
[("NP","VP"),("VP","NP"),("TV","NP"),("NP","TV")]
```

ghci 78> *cCommand snowwhite*

```
[([0],[1]),([0],[1,0]),([0],[1,0,0]),([0],[1,1]),([0],[1,1,0]),([1],[0]),([1],[0,0]),([1,0],
[1,1]),([1,0],[1,1,0]),([1,1],[1,0]),([1,1],[1,0,0])]
```

ghci 79> *nodeCCommand snowwhite*

```
[("NP","VP"),("NP","TV"),("NP","NP"),("VP","NP"),("TV","NP"),("NP","TV")]
```

ghci 80> *branchingPos snowwhite*

```
[[],[1]]
```

ghci 81> *nodeBranchingPos snowwhite*

```
["S","VP"]
```

ghci 82> *precedence snowwhite*

```
[([0],[1]),([0],[1,0]),([0],[1,0,0]),([0],[1,1]),([0],[1,1,0]),([0,0],[1]),([0,0],[1,0]),
([0,0],[1,0,0]),([0,0],[1,1]),([0,0],[1,1,0]),([1,0],[1,1]),([1,0],[1,1,0]),([1,0,0],[1,
1]),([1,0,0],[1,1,0])]
```

ghci 83> *nodePrecedence snowwhite*

```
[("NP","VP"),("NP","TV"),("NP","NP"),("TV","NP")]
```

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:

$$split2 :: [a] \to [([a],[a])]$$
$$split2\ [\,] = [([\,],[\,])]$$
$$split2\ (x:xs) = [([\,],(x:xs))] + (map\ (\lambda(ys,zs) \to ((x:ys),zs))\ (split2\ xs))$$

ghci 84> *split2* `"a"`
`[("","a"),("a","")]`

ghci 85> *split2* `"ab"`
`[("","ab"),("a","b"),("ab","")]`

ghci 86> *split2* `"abc"`
`[("","abc"),("a","bc"),("ab","c"),("abc","")]`

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

$$splitN :: Int \to [a] \to [[[a]]]$$
$$splitN\ n\ xs$$
$$\mid n \leqslant 1 = error\ \texttt{"cannot split"}$$
$$\mid n \equiv 2 = [[ys,zs] \mid (ys,zs) \leftarrow split2\ xs]$$
$$\mid otherwise = [ys:rs \mid (ys,zs) \leftarrow split2\ xs, rs \leftarrow splitN\ (n-1)\ zs]$$

ghci 87> *splitN* 1 `"a"`
$***Exception: cannot\ split$

ghci 88> *splitN* 3 `"a"`
`[["","","a"],["","a",""],["a","",""]]`

ghci 89> *splitN* 4 `"a"`
`[["","","","a"],["","","a",""],["","a","",""],["a","","",""]]`

ghci 90> *splitN* 3 `"ab"`
`[["","","ab"],["","a","b"],["","ab",""],["a","","b"],["a","b",""],["ab","",""]]`

ghci 91> *splitN* 3 `"abc"`
`[["","","abc"],["","a","bc"],["","ab","c"],["","abc",""],["a","","bc"],["a","b","c"],["a","bc",""],["ab","","c"],["ab","c",""],["abc","",""]]`

And now we define a function that generates all finite strings of length n defined over an input alphabet:

$$
\begin{aligned}
&\mathit{gener} :: \mathit{Int} \to \mathit{String} \to [\mathit{String}] \\
&\mathit{gener}\ 0\ \mathit{alphabet} = [[\,]] \\
&\mathit{gener}\ n\ \mathit{alphabet} = [x : \mathit{xs} \mid x \leftarrow \mathit{alphabet}, \mathit{xs} \leftarrow \mathit{gener}\ (n-1)\ \mathit{alphabet}]
\end{aligned}
$$

```
ghci 92> gener 0 "abc"
  [""]
```

```
ghci 93> gener 1 "abc"
  ["a","b","c"]
```

```
ghci 94> gener 2 "abc"
  ["aa","ab","ac","ba","bb","bc","ca","cb","cc"]
```

```
ghci 95> length $ gener 2 "abc"
  9
```

```
ghci 96> gener 3 "abc"
  ["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"]
```

```
ghci 97> length $ gener 3 "abc"
  27
```

```
ghci 98> gener 4 "abc"
  ["aaaa","aaab","aaac","aaba","aabb","aabc","aaca","aacb","aacc","abaa","abab",
  "abac","abba","abbb","abbc","abca","abcb","abcc","acaa","acab","acac","acba",
  "acbb","acbc","acca","accb","accc","baaa","baab","baac","baba","babb","babc",
  "baca","bacb","bacc","bbaa","bbab","bbac","bbba","bbbb","bbbc","bbca","bbcb",
  "bbcc","bcaa","bcab","bcac","bcba","bcbb","bcbc","bcca","bccb","bccc","caaa",
  "caab","caac","caba","cabb","cabc","caca","cacb","cacc","cbaa","cbab","cbac",
  "cbba","cbbb","cbbc","cbca","cbcb","cbcc","ccaa","ccab","ccac","ccba","ccbb",
  "ccbc","ccca","cccb","cccc"]
```

```
ghci 99> length $ gener 4 "abc"
  81
```

We use this function to define a generator for the infinite set of all finite strings over an input alphabet:

$$
\begin{aligned}
&\mathit{gener}' :: \mathit{Int} \to \mathit{String} \to [\mathit{String}] \\
&\mathit{gener}'\ n\ \mathit{alphabet} = \mathit{gener}\ n\ \mathit{alphabet} +\!\!+\ \mathit{gener}'\ (n+1)\ \mathit{alphabet} \\
&\mathit{generateAll} :: \mathit{String} \to [\mathit{String}] \\
&\mathit{generateAll}\ \mathit{alphabet} = \mathit{gener}'\ 0\ \mathit{alphabet}
\end{aligned}
$$

```
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","aabc","aaca","aacb","aacc","abaa","abab","abac",
  "abba","abbb","abbc","abca","abcb","abcc","acaa","acab","acac","acba","acbb",
  "acbc","acca","accb","accc","baaa","baab","baac","baba","babb","babc","baca",
  "bacb","bacc","bbaa","bbab","bbac","bbba","bbbb","bbbc","bbca","bbcb","bbcc",
  "bcaa","bcab","bcac","bcba","bcbb","bcbc","bcca","bccb","bccc","caaa","caab",
  "caac","caba","cabb","cabc"]
```

```
ghci 101> take 400 $ 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","aabc","aaca","aacb","aacc","abaa","abab",
    "abac","abba","abbb","abbc","abca","abcb","abcc","acaa","acab","acac","acba",
    "acbb","acbc","acca","accb","accc","baaa","baab","baac","baba","babb","babc",
    "baca","bacb","bacc","bbaa","bbab","bbac","bbba","bbbb","bbbc","bbca","bbcb",
    "bbcc","bcaa","bcab","bcac","bcba","bcbb","bcbc","bcca","bccb","bccc","caaa",
    "caab","caac","caba","cabb","cabc","caca","cacb","cacc","cbaa","cbab","cbac",
    "cbba","cbbb","cbbc","cbca","cbcb","cbcc","ccaa","ccab","ccac","ccba","ccbb",
    "ccbc","ccca","cccb","cccc","aaaaa","aaaab","aaaac","aaaba","aaabb","aaabc",
    "aaaca","aaacb","aaacc","aabaa","aabab","aabac","aabba","aabbb","aabbc",
    "aabca","aabcb","aabcc","aacaa","aacab","aacac","aacba","aacbb","aacbc",
    "aacca","aaccb","aaccc","abaaa","abaab","abaac","ababa","ababb","ababc",
    "abaca","abacb","abacc","abbaa","abbab","abbac","abbba","abbbb","abbbc",
    "abbca","abbcb","abbcc","abcaa","abcab","abcac","abcba","abcbb","abcbc",
    "abcca","abccb","abccc","acaaa","acaab","acaac","acaba","acabb","acabc",
    "acaca","acacb","acacc","acbaa","acbab","acbac","acbba","acbbb","acbbc",
    "acbca","acbcb","acbcc","accaa","accab","accac","accba","accbb","accbc",
    "accca","acccb","acccc","baaaa","baaab","baaac","baaba","baabb","baabc",
    "baaca","baacb","baacc","babaa","babab","babac","babba","babbb","babbc",
    "babca","babcb","babcc","bacaa","bacab","bacac","bacba","bacbb","bacbc",
    "bacca","baccb","baccc","bbaaa","bbaab","bbaac","bbaba","bbabb","bbabc",
    "bbaca","bbacb","bbacc","bbbaa","bbbab","bbbac","bbbba","bbbbb","bbbbc",
    "bbbca","bbbcb","bbbcc","bbcaa","bbcab","bbcac","bbcba","bbcbb","bbcbc",
    "bbcca","bbccb","bbccc","bcaaa","bcaab","bcaac","bcaba","bcabb","bcabc",
    "bcaca","bcacb","bcacc","bcbaa","bcbab","bcbac","bcbba","bcbbb","bcbbc",
    "bcbca","bcbcb","bcbcc","bccaa","bccab","bccac","bccba","bccbb","bccbc",
    "bccca","bcccb","bcccc","caaaa","caaab","caaac","caaba","caabb","caabc",
    "caaca","caacb","caacc","cabaa","cabab","cabac","cabba","cabbb","cabbc",
    "cabca","cabcb","cabcc","cacaa","cacab","cacac","cacba","cacbb","cacbc",
    "cacca","caccb","caccc","cbaaa","cbaab","cbaac","cbaba","cbabb","cbabc",
    "cbaca","cbacb","cbacc","cbbaa","cbbab","cbbac","cbbba","cbbbb","cbbbc",
    "cbbca","cbbcb","cbbcc","cbcaa","cbcab","cbcac","cbcba","cbcbb","cbcbc",
    "cbcca","cbccb","cbccc","ccaaa","ccaab","ccaac","ccaba","ccabb","ccabc",
    "ccaca","ccacb","ccacc","ccbaa","ccbab","ccbac","ccbba","ccbbb","ccbbc",
    "ccbca","ccbcb","ccbcc","cccaa","cccab","cccac","cccba","cccbb","cccbc",
    "cccca","ccccb","ccccc","aaaaaa","aaaaab","aaaaac","aaaaba","aaaabb","aaaabc",
    "aaaaca","aaaacb","aaaacc","aaabaa","aaabab","aaabac","aaabba","aaabbb",
    "aaabbc","aaabca","aaabcb","aaabcc","aaacaa","aaacab","aaacac","aaacba",
    "aaacbb","aaacbc","aaacca","aaaccb","aaaccc","aabaaa","aabaab","aabaac",
    "aababa","aababb","aababc","aabaca","aabacb","aabacc"]
```

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:

ghci 102> $:l$ *Palindromes*

ghci 103> $:t$ *recognize*

recognize :: *String* → *Bool*

ghci 104> *recognize* `"aa"`

True

ghci 105> *recognize* `"aba"`

True

ghci 106> *recognize* `"ab"`

False

ghci 107> *recognize* `"abca"`

False

ghci 108> $:t$ *generate*

generate :: [*String*]

ghci 109> *take* 100 *generate*

```
["","a","b","c","aa","bb","cc","aaa","aba","aca","bab","bbb","bcb","cac","cbc",
"ccc","aaaa","abba","acca","baab","bbbb","bccb","caac","cbbc","cccc","aaaaa",
"aabaa","aacaa","ababa","abbba","abcba","acaca","acbca","accca","baaab",
"babab","bacab","bbabb","bbbbb","bbcbb","bcacb","bcbcb","bcccb","caaac",
"cabac","cacac","cbabc","cbbbc","cbcbc","ccacc","ccbcc","ccccc","aaaaaa",
"aabbaa","aaccaa","abaaba","abbbba","abccba","acaaca","acbbca","acccca",
"baaaab","babbab","baccab","bbaabb","bbbbbb","bbccbb","bcaacb","bcbbcb",
"bccccb","caaaac","cabbac","caccac","cbaabc","cbbbbc","cbccbc","ccaacc",
"ccbbcc","cccccc","aaaaaaa","aaabaaa","aaacaaa","aababaa","aabbbaa","aabcbaa",
"aacacaa","aacbcaa","aacccaa","abaaaba","abababa","abacaba","abbabba",
"abbbbba","abbcbba","abcacba","abcbcba","abcccba","acaaaca","acabaca",
"acacaca"]
```

ghci 110> $:t$ *parse*

parse :: *String* → [*ParseTree String String*]

```
ghci 111> parse "aa"
    [["A-pair" : "a" "a"]]
```

```
ghci 112> parse "abba"
    [["A-pair" : "a" ["B-pair" : "b" "b"] "a"]]
```

```
ghci 113> parse "abccba"
    [["A-pair" : "a" ["B-pair" : "b" ["C-pair" : "c" "c"] "b"] "a"]]
```

```
ghci 114> parse "abcacba"
    [["A-pair" : "a" ["B-pair" : "b" ["C-pair" : "c" "a" "c"] "b"] "a"]]
```

3 Intro to parser combinators

```
ghci 115> :l ParserCombinatorsMini
```

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:

ghci 116$>$ $:i\ ParseState$

type $ParseState\ partialParse\ remainingInput = [(partialParse, [remainingInput])]$ -- Defined at ParserCombinatorsMini.hs:7:6

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 $(partialParse, [remainingInput])$ pairs because we might have a locally or globally ambiguous sentence with multiple partial (for local ambiguity) or final (for global ambiguity) parses.

ghci 117$>$ $:i\ Parser$

type $Parser\ input\ parse = [input] \rightarrow ParseState\ parse\ input$ -- Defined at ParserCombinatorsMini.hs:8:6

A parser is just a function from a list of inputs to a parse state, i.e., to a list of $(partial - parse, remaining - inputs)$ 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 $(partial - parse, remaining - inputs)$ pairs and output $(partial - parse, remaining - inputs)$ pairs, where the output $partial - parse$ is usually bigger than the input one and the output list of $remaining - inputs$ is usually smaller.

Instead of this, we discard the $partial - parse$ 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 $(partial - parse, remaining - inputs)$ 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:

$LEXICON:$
$NP \rightarrow Alice \mid Dorothy$
$VP \rightarrow smiled \mid laughed$
$D \rightarrow every \mid some \mid no$
$N \rightarrow dwarf \mid wizard$
$PHRASE\ STRUCTURE\ RULES:$
$S \rightarrow NP\ VP$
$NP \rightarrow D\ N$

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.

$$\begin{aligned}&\mathit{succeed} :: \mathit{parse} \rightarrow \mathit{Parser\ input\ parse}\\&\mathit{succeed\ finalParse\ remainingInputs} = [(\mathit{finalParse}, \mathit{remainingInputs})]\end{aligned}$$

ghci 118> $:t$ *succeed*
succeed :: *parse* → *Parser input parse*

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

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

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:

$$\begin{aligned}&\mathit{failp} :: \mathit{Parser\ input\ parse}\\&\mathit{failp\ remainingInputs} = [\,]\end{aligned}$$

ghci 122> $:t$ *failp*
failp :: *Parser input parse*

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

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

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

$$\begin{aligned}&\mathit{symbol} :: \mathit{Eq\ terminal} \Rightarrow \mathit{terminal} \rightarrow \mathit{Parser\ terminal\ terminal}\\&\mathit{symbol\ t}\ [\,] \qquad\qquad\quad = [\,]\\&\mathit{symbol\ t}\ (x : xs) \mid t \equiv x = [(t, xs)]\\&\quad \mid \mathit{otherwise} = [\,]\end{aligned}$$

ghci 125> $:t\ \mathit{symbol}$
$\mathit{symbol} :: \mathit{Eq\ terminal} \Rightarrow \mathit{terminal} \rightarrow \mathit{Parser\ terminal\ terminal}$

ghci 126> $:t\ \mathit{symbol}$ `'a'`
symbol `'a'` $:: \mathit{Parser\ Char\ Char}$

ghci 127> $:t\ \mathit{symbol}$ `'b'`
symbol `'b'` $:: \mathit{Parser\ Char\ Char}$

ghci 128> $:t\ \mathit{symbol}$ `'c'`
symbol `'c'` $:: \mathit{Parser\ Char\ Char}$

ghci 129> symbol `'a' "abc"`
`[('a',"bc")]`

ghci 130> symbol `'a' "bac"`
`[]`

ghci 131> symbol `'b' "bac"`
`[('b',"ac")]`

ghci 132> symbol `'c' "cab"`
`[('c',"ab")]`

We can parse a sequence of characters / symbols at a time by specifying a parser for lists of terminals rather than simply terminals:

$$
\begin{aligned}
&\mathit{token} :: \mathit{Eq\ terminal} \Rightarrow [\mathit{terminal}] \rightarrow \mathit{Parser\ terminal}\ [\mathit{terminal}] \\
&\mathit{token\ ts\ xs} \mid \mathit{ts} \equiv \mathit{take\ n\ xs} = [(\mathit{ts}, \mathit{drop\ n\ xs})] \\
&\quad \mid \mathit{otherwise} \qquad\qquad\quad = [\,] \\
&\quad \textbf{where}\ n = \mathit{length\ ts}
\end{aligned}
$$

ghci 133> $:t\ \mathit{token}$
$\mathit{token} :: \mathit{Eq\ terminal} \Rightarrow [\mathit{terminal}] \rightarrow \mathit{Parser\ terminal}\ [\mathit{terminal}]$

ghci 134> $:t\ \mathit{token}$ `"a"`
token `"a"` $:: \mathit{Parser\ Char}\ [\mathit{Char}]$

ghci 135> $:t$ *token* `"ab"`
token `"ab"` :: *Parser Char* [*Char*]

ghci 136> $:t$ *token* `"b"`
token `"b"` :: *Parser Char* [*Char*]

ghci 137> $:t$ *token* `"abc"`
token `"abc"` :: *Parser Char* [*Char*]

ghci 138> *token* `"a"` `"abcd"`
`[("a","bcd")]`

ghci 139> *token* `"a"` `"bacd"`
`[]`

ghci 140> *token* `"ab"` `"abcd"`
`[("ab","cd")]`

ghci 141> $:t$ *token* `"ab"` `"abcd"`
token `"ab"` `"abcd"` :: *ParseState* [*Char*] *Char*

ghci 142> *token* `["ab"]` `["ab","cd"]`
`[(["ab"],["cd"])]`

Finally, we can also define 'looser' parsers that parse an input if it satisfies a predicate (this is what the function *reads* in the *Prelude* actually is).

```
satisfy :: (input → Bool) → Parser input input
satisfy p []             = []
satisfy p (i : is) | p i = [(i, is)]
    | otherwise = []
digit :: Parser Char Char
digit = satisfy isDigit
```

ghci 143> *digit* `"1a"`
`[('1',"a")]`

ghci 144> *digit* "a1"
  [ ]

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

$$\textit{just} :: \textit{Parser input parse} \to \textit{Parser input parse}$$
$$\textit{just}\ p = \textit{filter}\ (\textit{null} \circ \textit{snd}) \circ p$$

ghci 145> *digit* "1"
  [('1',"")]

ghci 146> (*just digit*) "1"
  [('1',"")]

ghci 147> *digit* "1a"
  [('1',"a")]

ghci 148> (*just digit*) "1a"
  [ ]

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., $VP \to V\ NP \mid V\ NP\ PP$), we have a parser combinator that takes two parsers and parses an input if at least one of the two parser parses the input.

$$\textbf{infixr}\ 4 <\mid>$$
$$(<\mid>) :: \textit{Parser input parse} \to \textit{Parser input parse} \to \textit{Parser input parse}$$
$$(p1 <\mid> p2)\ xs = p1\ xs + p2\ xs$$

ghci 149> $:t\ (<\mid>)$
  $(<\mid>) :: \textit{Parser input parse} \to \textit{Parser input parse} \to \textit{Parser input parse}$

ghci 150> :*t digit*
digit :: *Parser Char Char*

ghci 151> :*t* (*symbol* `'a'`)
(*symbol* `'a'`) :: *Parser Char Char*

ghci 152> **let** *digitOrSymbolA* = *digit* < | > (*symbol* `'a'`)

ghci 153> *digitOrSymbolA* `"1a"`
[(`'1'`,`"a"`)]

ghci 154> *digitOrSymbolA* `"9a"`
[(`'9'`,`"a"`)]

ghci 155> *digitOrSymbolA* `"a1"`
[(`'a'`,`"1"`)]

ghci 156> *digitOrSymbolA* `"b1"`
[ ]

ghci 157> **let** *tokenAorB* = (*token* `"a"`) < | > (*token* `"b"`)

ghci 158> *tokenAorB* `"abc"`
[(`"a"`,`"bc"`)]

ghci 159> *tokenAorB* `"bac"`
[(`"b"`,`"ac"`)]

ghci 160> *tokenAorB* `"cab"`
[ ]

We also want to be able to sequence 2 parsers. For examples, to parse a sentence according to this rule $S \rightarrow NP\ VP$, we first want to parse an NP, then a VP.

$$(<*>) :: Parser\ input\ [parse] \rightarrow Parser\ input\ [parse] \rightarrow Parser\ input\ [parse]$$
$$(p <*> q)\ xs = [(r1 +\!\!+ r2, zs) \mid (r1, ys) \leftarrow p\ xs,$$
$$\quad (r2, zs) \leftarrow q\ ys]$$

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 < $ >
$$(<\$>) :: (input \rightarrow parse) \rightarrow Parser\ s\ input \rightarrow Parser\ s\ parse$$
$$(f <\$> p)\ xs = [(f\ x, ys) \mid (x, ys) \leftarrow p\ xs]$$

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'

ghci 169> **let** $\{digitize :: Parser\ Char\ Int;$
$digitize = f < \$ > digit$
where $f\ c = ord\ c - ord$ `'0'` $\}$

ghci 170> *digit* `"1a"`
`[('1',"a")]`

ghci 171> $:t\ (digit$ `"1a"`$)$
$(digit$ `"1a"`$) :: ParseState\ Char\ Char$

ghci 172> *digitize* `"1a"`
`[(1,"a")]`

ghci 173> $:t\ (digitize$ `"1a"`$)$
$(digitize$ `"1a"`$) :: ParseState\ Int\ Char$

6 Simple application of parser combinators: The Palindrome Grammar

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

ghci 174> $:i\ PARSER$
type $PARSER\ input\ category = Parser\ input\ (ParseTree\ category\ input)$ -- Defined at ParserCombinatorsMini.hs

ghci 175> $:i\ ParseTree$
data $ParseTree\ nonterminal\ terminal = EmptyTree \mid Leaf\ terminal \mid$
$Branch\ nonterminal\ [ParseTree\ nonterminal\ terminal]$ -- Defined at BasicDef.hs:6:6 instance (Eq nonterminal, Eq

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

$epsilonT :: PARSER\ input\ category$
$epsilonT = succeed\ EmptyTree$

ghci 176> *epsilonT* `"i output the EmptyTree for any input"`
`[(,"i output the EmptyTree for any input")]`

We also define a *PARSER* that will build a *Leaf* sub-tree on top of any basic symbol:

$symbolT :: Eq\ input \Rightarrow input \rightarrow PARSER\ input\ category$
$symbolT\ s = (\lambda x \rightarrow Leaf\ x) <\$> symbol\ s$

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

ghci 178> *symbolT* `'a'` `"abc"`
`[('a',"bc")]`

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

ghci 179> **let** {*palindrome* :: *PARSER Char String*;
palindrome = *epsilonT* <|> *symbolT* `'a'` <|> *symbolT* `'b'` <|> *symbolT* `'c'` <|>
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

$parse :: String \rightarrow [ParseTree\ String\ String]$
$parse = \lambda xs \rightarrow$
$[EmptyTree \mid null\ xs] +\!\!+ [Leaf\ \texttt{"a"} \mid xs \equiv \texttt{"a"}] +\!\!+$
$[Leaf\ \texttt{"b"} \mid xs \equiv \texttt{"b"}] +\!\!+ [Leaf\ \texttt{"c"} \mid xs \equiv \texttt{"c"}] +\!\!+$
$[Branch\ \texttt{"A-pair"}\ [Leaf\ \texttt{"a"}, t, Leaf\ \texttt{"a"}] \mid [\texttt{"a"}, ys, \texttt{"a"}] \leftarrow splitN\ 3\ xs, t \leftarrow parse\ ys] +\!\!+$
$[Branch\ \texttt{"B-pair"}\ [Leaf\ \texttt{"b"}, t, Leaf\ \texttt{"b"}] \mid [\texttt{"b"}, ys, \texttt{"b"}] \leftarrow splitN\ 3\ xs, t \leftarrow parse\ ys] +\!\!+$
$[Branch\ \texttt{"C-pair"}\ [Leaf\ \texttt{"c"}, t, Leaf\ \texttt{"c"}] \mid [\texttt{"c"}, ys, \texttt{"c"}] \leftarrow splitN\ 3\ xs, t \leftarrow parse\ ys]$

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',"bccba"),(["A-pair" : 'a' ["B-pair" : 'b' ["C-pair" : 'c' 'c'] 'b'] 'a'],"")]`

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

ghci 184> $[parse \mid (parse, input) \leftarrow palindrome$ `"abcacba"`$, input \equiv$ `""`$]$
`[["A-pair" : 'a' ["B-pair" : 'b' ["C-pair" : 'c' 'a' 'c'] 'b'] 'a']]`

ghci 185> $head\ [parse \mid (parse, input) \leftarrow palindrome$ `"abcacba"`$, input \equiv$ `""`$]$
`["A-pair" : 'a' ["B-pair" : 'b' ["C-pair" : 'c' 'a' 'c'] 'b'] 'a']`

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

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

LEXICON :
$NP \rightarrow Alice \mid Dorothy$
$VP \rightarrow smiled \mid laughed$
$D \rightarrow every \mid some \mid no$
$N \rightarrow dwarf \mid wizard$
PHRASE STRUCTURE RULES :
$S \rightarrow NP\ VP$
$NP \rightarrow D\ N$

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

$(<*>) :: Parser\ input\ [parse] \rightarrow Parser\ input\ [parse] \rightarrow Parser\ input\ [parse]$
$(p <*> q)\ xs = [(r1 + r2, zs) \mid (r1, ys) \leftarrow p\ xs,$
$\quad (r2, zs) \leftarrow q\ ys]$

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.

$\mathbf{infixl}\ 6 <:>$
$(<:>) :: \mathit{Parser\ input\ category} \rightarrow \mathit{Parser\ input}\ [\mathit{category}] \rightarrow \mathit{Parser\ input}\ [\mathit{category}]$
$(p <:> q)\ xs = [(r : rs, zs) \mid (r, ys) \leftarrow p\ xs, (rs, zs) \leftarrow q\ ys]$

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

$\mathit{collect} :: [\mathit{Parser\ input\ category}] \rightarrow \mathit{Parser\ input}\ [\mathit{category}]$
$\mathit{collect}\ [\,] = \mathit{succeed}\ [\,]$
$\mathit{collect}\ (p : ps) = p <:> \mathit{collect}\ ps$

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

ghci 189> **let** $\{pS = \mathit{collect}\ [pNP, pVP];$
$pNP = \mathit{symbol}$ `"Alice"` $< \mid > \mathit{symbol}$ `"Dorothy"`;
$pVP = \mathit{symbol}$ `"smiled"` $< \mid > \mathit{symbol}$ `"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 $S \rightarrow NP\ VP$, the function *parseAs* takes `"S"` as its label and assumes that a list of parsers for NPs and VPs has been constructed

$\mathit{parseAs} :: \mathit{category} \rightarrow [\mathit{PARSER\ input\ category}] \rightarrow \mathit{PARSER\ input\ category}$
$\mathit{parseAs\ label\ ps} = (\lambda xs \rightarrow \mathit{Branch\ label\ xs}) < \$ > \mathit{collect\ ps}$

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** $\{pS = \mathit{parseAs}$ `"S"` $[pNP, pVP];$
$pNP = \mathit{symbolT}$ `"Alice"` $< \mid > \mathit{symbolT}$ `"Dorothy"`;
$pVP = \mathit{symbolT}$ `"smiled"` $< \mid > \mathit{symbolT}$ `"laughed"` $\}$

ghci 192> pS `["Alice","smiled"]`
`[(["S":"Alice" "smiled"],[])]`

The full parser specification is this:

```
ghci 193> 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"}
```

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

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