Deductive chart parsing in Haskell

Uneson, Marcus

2008

Markham, D. 1999. 'Listeners and disguised voices: the imitation and perception of dialectal accent'. *Forensic Linguistics* 6, 289-299.

McGehee, F. 1937. 'The reliability of the identification of the human voice' *Journal of General Psychology* 17, 249-271.

Reich, A. & J. Duke. 1979. 'Effects of selected voice disguises upon speaker identification by listening'. *Journal of the Acoustical Society of America* 66, 1023-1028.

Rodman, R. & M. Powell. 2000. 'Computer recognition of speakers who disguise their voice'. *The international conference on signal processing applications and technology ICSPAT2000.* http://www.csc.ncsu.edu/faculty/rodman/Computer%20Recognition%20of%20Speakers%20Who%20Disguise%20Their%20Voice.pdf (accessed 19 April 2008).

Rose, P. & S. Duncan. 1995. 'Naïve auditory identification and discrimination of similar voices by familiar listeners'. *Forensic Linguistics* 2, 1-17.

Schiller, N.O. & O. Köster. 1998. 'The ability of expert witnesses to identify voices: a comparison between trained and untrained listeners'. *Forensic Linguistics* 5, 1-9.

Sjöström, M. 2005. 'Earwitness identification: can a switch of dialect fool us?'. Masters paper, Dept of Philosophy and Linguistics, Umeå university.

Thompson, C.P. 1987. 'A language effect in voice identification'. *Applied Cognitive Psychology* 1:2, 121-131.

Van Lancker, D., J. Kreiman & K.T.D. Wickens. 1985. 'Familiar voice recognition: patterns and parameters, Part 2: Recognition of rate-altered voices' *Journal of Phonetics* 13, 39-52.

Yarmey, A.D. 1995. 'Earwitness speaker identification' *Psychology, Public Policy, and Law* 1, 792-816.

Yarmey, A.D., A.L. Yarmey, M.J. Yarmey & L. Parliament. 2001. 'Commonsense beliefs and the identification of familiar voices'. *Applied Cognitive Psychology* 15, 283-299.

Zetterholm, E. 2003. *Voice imitation: a phonetic study of perceptual illusions and acoustic success*. Lund: Dept. of Linguistics and Phonetics, Lund University.

Zetterholm, E. 2006. 'Same speaker – different voices: a study of one impersonator and some of his different imitations'. *Proceedings SST2006*, 70-75. Auckland, New Zealand.

Lund University, Dept. of Linguistics and Phonetics
Working Papers 53 (2008), 159–177

Deductive chart parsing in Haskell

Marcus Uneson

1 Introduction

Given a formal grammar and a string of tokens, the problem of *parsing* amounts to deciding whether the string is recognized by the grammar; and, if so, returning some suitable representation of its structure. Parsing has ubiquitous applications as a preprocessing step in computational linguistics, for instance in speech recognition, machine translation, and information extraction.

Grammars describing natural language (as opposed to formal grammars, explicitly designed to minimize ambiguities) are notoriously ambiguous, and naive parsing algorithms often have time complexity $O(a^n)$ in the length of the input sequence. *Chart parsing*, originally proposed by Earley (Earley 1970) and Cocke, Kasami, and Younger (Kasami 1965, Younger 1967), is a family of widely used dynamic programming algorithms which achieve $O(n^3)$ running times, by saving partial parses, *items*, in a *chart*, or lookup table. Chart parsing has been generalized (Shieber, Schabes & Pereira 1995) to *deductive parsing*, where a simple, dedicated natural deduction prover allows the parsing process to be described declaratively. In this framework, a particular parsing algorithm corresponds to a particular logic with a particular set of inference rules and axioms; thus, imperatively rather diverse top-down and bottom-up algorithms can be expressed relatively uniformly.

The present paper describes an attempt to transfer (the chart parsing part of) the deduction engine of Shieber et al. from the logical into the functional programming paradigm, with the hope of reaping well-known functional benefits such as referential transparency and higher-order functions without sacrificing either too much declarativity or speed in the process. We draw the outlines of a reasonably efficient deduction engine in the purely functional language Haskell.

The paper is organized as follows. In Section 2, we present the notation and mechanics used for grammar, chart, and parsing logic. The bulk of the paper is made up by the implementational notes in Section 3, where we

summarize the representations and roles of the protagonists of chart parsing: tokens, symbols, rules, grammar, items, chart, deduction engine, inference rules. Section 4 describes filtering based on the left-corner relation, Section 5 deals with extracting parse trees, and Section 6 discusses briefly time and space complexity. We conclude with some remarks on future directions. Five variations on Kilbury bottom-up chart parsing expressed as parsing logics are given in Appendix A.

2 Chart parsing and deductive systems

2.1 Context-free grammars (CFGs)

A *context-free grammar* G is usually described as a tuple $G = (N, \Sigma, P, S)$, where N and Σ are disjoint sets of *nonterminal* and *terminal* symbols, respectively; P is a set of *productions* or *rules* and $S \in N$ is the start symbol. The nonterminals are also called *categories* and the set $V = N \cup \Sigma$ are the *symbols* of the grammar. Each production in P is of the form $A \rightarrow \alpha$ where $A \in N$ is a nonterminal and $\alpha \in V^*$ is a sequence of symbols. As is usual, we will reserve A, B, C for denoting single nonterminals, and α, β, γ for arbitrary strings of terminals and nonterminals. When discussing rules generically, or when it is otherwise clear what rule is being referred to, we will also informally use 'lhs' and 'rhs' with the obvious interpretations.

An *A phrase* is a sequence of terminals $\beta \in \Sigma^*$ such that $A \Rightarrow^* \beta$ for some $A \in N$, where the rewriting relation $\Rightarrow$ is defined as $\alpha B \gamma \Rightarrow \alpha\beta\gamma$ whenever $\alpha, \gamma \in V^*$ and $B \rightarrow \beta \in P$. A *sentence* is an S phrase, i.e., a phrase recognized by the start symbol. The *language L* accepted by a grammar is the set of sentences of that grammar.

In the rest of this paper, we will make a few further assumptions, typically true for CFGs intended to describe natural language:

1. The grammar contains no empty productions $A \rightarrow \varepsilon$ (that is, the empty string either is not in L, or else it can be handled separately).
2. The grammar is in *Normal Form (NF)*: all rules are either of form $A \rightarrow N^*$ or $B \rightarrow t$, where $A, B \in N$, $t \in \Sigma$. We will refer to the first set of rules P_n as *phrase category rules* and to the second P_t as *preterminal rules*, and to the left-hand sides of P_n and P_t, viewed as sets, as *phrase categories* (N_n) and *preterminals* (N_t), respectively. Intuitively, preterminals correspond to parts of speech and phrase categories to higher syntactic categories; however, although P_n and P_t form a partition of P (they will if the grammar is in NF), N_n and N_t need not be disjoint.


```
S -> NP VP                          N -> hästen | arbete
NP-> N | AdjP NP                    V -> avskyr
AdjP -> Adj                         Adv -> intensivt
VP -> V NP | V AdvP NP              Adj -> intensivt
AdvP -> Adv
```

Figure 1. Example grammar. Alternative rules are separated by '|'. P_n in left column, P_t in right. N_n={S, NP, AdjP, VP, AdvP}; N_t = {N, V, Adv, Adj}.

It should be noted that the NF requirement is no severe restriction – any context-free grammar can be easily transformed into NF. Figure 1 gives an example grammar.

2.2 Chart parsing

The *chart* of chart parsing is a data structure holding a set of *items*, which we can think of as the discovered parsing facts up to the current point in the input string. An item has the general form $[A \rightarrow \alpha\gamma, i, j]$ which can be read 'in the process of identifying an instance of category A starting from i, we are currently at j and have so far found α; it remains to find γ'. The dot thus represents the position j. The set of all items with the dot at k is called Earley state k.

New items may be added to the chart in three ways. In *prediction*, we consult the grammar to find possible expansions of a needed nonterminal. In *scanning*, we examine the tokens of the input string. Scanning will be done once per input token and does not depend on the grammar. In *completion*, finally, we combine two known facts into a third. Prediction will result in items with empty α and $i = j$; scanning will yield items with empty γ and $j = i + 1$; and completion will intuitively 'move the dot', i.e., shift the first category in γ and add it to α. A state of type $[A \rightarrow \alpha\cdot, i, j]$ (i.e. γ is empty) is known as a *passive* state; this corresponds to a confirmed category A spanning $w_i \ldots w_j$. Conversely, a state with non-empty γ is called an *active* state, meaning that it is only partially confirmed and still looking for continuations. The parse succeeds if the item $[S \rightarrow \alpha\cdot, 0, n]$ is in the chart (see also Figure 3).

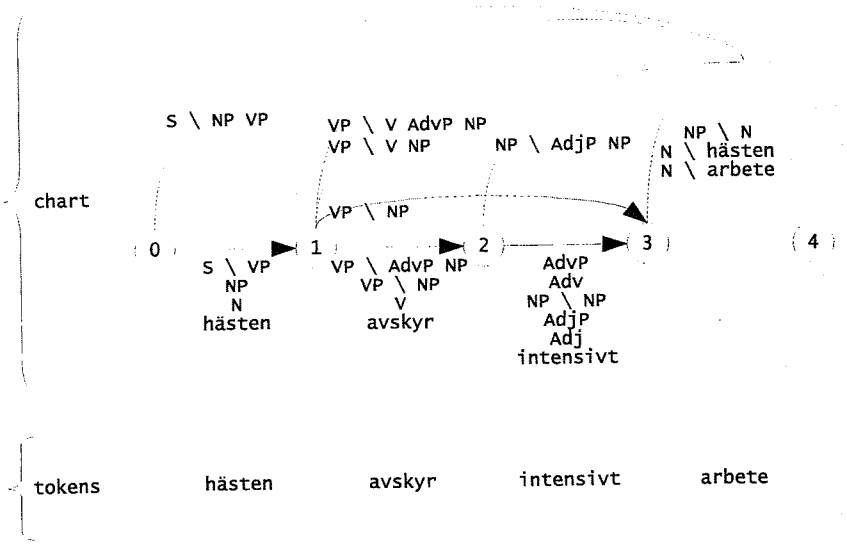

Figure 2. Chart parsing of sentence *hästen avskyr intensivt arbete* 'the horse detests work intensively/intensive work' according to the grammar above. The notation '\' denotes an active arc, where the '\' separates a category to the left from its 'wish list', the still missing components to the right. For instance, VP \ AdvP NP on the arc from 1 to 2 can be read 'if we, starting at the target of this arc, can find an AdvP followed by an NP, then we will also have found a VP starting at its source'. Passive arcs are simply arcs with empty wish lists; these correspond to confirmed nonterminals for some token substring (although we omit the '\' to reduce clutter).

States resulting from completion are depicted with solid arcs (shown as one n-labeled arc instead of n single-labeled ones, to reduce clutter), and those yielded by prediction are shown dashed (irrelevant ones omitted).

The chart shows the situation just before the last token *arbete* is scanned. When this happens, a terminal *arbete* will be completed from 3 to 4; this in turn completes a noun from 3 to 4; this in turn completes an NP from 3 to 4; this in turn completes a) a VP from 1 to 4 and b) an NP from 2 to 4; the latter in turn completes another VP from 1 to 4; the two VPs complete an S from 0 to 4 (in two ways, reflecting the ambiguity of the sentence).

The parsing process can usefully be visualized as a digraph with nodes 0 ... n corresponding to the spaces between the n input tokens (including before first and after last token). In this view, items correspond to the arcs of the graph, and a parse is just a path through the graph from node 0 to node n. Ambiguous sentences have more than one path through the graph. Figure 2 shows a chart under construction.

2.3 Chart parsing as a grammatical deduction system

If we interpret the arcs in the digraph view of chart parsing as inference rules, reformulating the problem as one of general deduction (Shieber, Schabes & Pereira 1995) is not far-fetched. That is, parsing consists in specifying a set of inference rules and axioms – a *parsing logic*. Exploiting the rules, the axioms, and/or previously proved statements, we may prove new statements about the grammatical status of a sequence of tokens. Our goal, analogous to traditional chart parsing above, is to prove that the theorem $[S \rightarrow \alpha\cdot, 0, n]$ is derivable in the logic, and we may view statements proved on the way as lemmata.

Inference rules will generally be of form:

$$\frac{A_1, A_2, \ldots, A_n}{B} \qquad P_1, P_2, \ldots, P_m$$

Here, $A_1 \ldots A_n$ are antecedents and B is the consequent. The predicates $P_1 \ldots P_m$ are side conditions on $A_1 \ldots A_n$ and B; in the case of parsing, these will express constraints given by the grammar. (If we wish, we may consider an axiom as just a special case of inference rule with $n = 0$, i.e., the validity of B is decidable from side conditions alone.)

For Earley parsing, the initial start state; the goal; and the rules scan, predict, and complete can be translated to a parsing logic as shown in Figure 3 (from Shieber, Schabes & Pereira 1995, slightly modified).

3 Deductive chart parsing in Haskell

3.1 Implementational considerations

It should be of no surprise that the logical programming paradigm in some ways is a more natural tool for describing a logical deductive system – after all, that is the kind of problem it was constructed to solve. For instance, we

$$[S' \rightarrow \cdot S, 0, 0] \qquad \text{(start axiom)}$$

$$[S' \rightarrow S\cdot, 0, n] \qquad \text{(goal)}$$

$$\frac{[A \rightarrow \alpha \cdot B \beta, i, j]}{[B \rightarrow \cdot \gamma, j, j]} \quad B \rightarrow \gamma \qquad \text{(inference rule: predict)}$$

$$\frac{[A \rightarrow \cdot t, j, j]}{[A \rightarrow t \cdot, j, j+1]} \quad t = w_j \qquad \text{(inference rule: scan)}$$

$$\frac{[A \rightarrow \alpha \cdot B \beta, i, k], [B \rightarrow \gamma \cdot, k, j]}{[A \rightarrow \alpha B \cdot \beta, i, j]} \qquad \text{(inference rule: complete)}$$

Figure 3. An Earley parsing logic, comprising a start axiom, a goal, and three inference rules (scan, predict, complete).

need to manage state in order to keep track of lemmas; this is efficiently and intuitively done by just adding new information to the database. Similarly, the non-determinism handling implicit in the backtracking mechanism allows the programmer to treat the relations expressed in logical formulae as were they functions.

By comparison, putting on the functional hair shirt may sometimes seem like dressing for yet another self-flagellation exercise. Some algorithms do rely on state in one way or another – most formulations of chart parsing, in traditional-imperative guises as well as logical-deductive, belong to this group. In pure functional programming, however, there is no concept of mutable memory. A function behaves like its mathematical counterpart: given some particular arguments, it will always return a particular value. When expressed functionally, such algorithms must have the state threaded. We may try to minimize the obtrusiveness of threading the state, perhaps by

carefully wrapping it up in a single argument or hiding it in a state monad, but declarativity is still at risk. Furthermore, logic formulae express relations, not functions. Implementationally, this means we will have to handle non-determinism in one way or another.

On the other hand, the pure functional paradigm offers some undisputed, general benefits (otherwise there would be little motivating its use in parsing in the first place). Among them we find higher-order functions and (crucially) referential transparency, offering a wide array of compile-time program transformations and optimizations as well as much simplified reasoning. In the case of Haskell, we also get things like static, polymorphic typing; lazy evaluation; and monadic effects.

In the following, we present the main data types and implementational notes of an attempt to transfer logical-deductive chart parsing into the functional paradigm, including the issues of state and non-determinism. Although the examples are given in Haskell (Peyton Jones et al. 1999), we hope they should be understandable with some experience in any functional language. Otherwise, Hudak, Peterson & Fasel 2000 is a good introduction to Haskell itself.

Except functions from the Prelude, we will use the following standard libraries:

```
> import qualified Data.Map as M
> import Data.Maybe (listToMaybe)
> import Data.List
> import Data.Array (listArray, (!))
> import Control.Monad (guard)
```

Documentation for Prelude and libraries comes with the distributions, and can also be found on `www.haskell.org`. The source code is available from the author's web page.

3.2 Token string

We represent the tokens of the input sequence by a function abstraction, allowing us to hide some constant-time access mechanism behind the curtains without exposing any details. Given some input tokens as a list, `mkTokens` produces a pair of a function which takes an zero-based integer position to the corresponding token; and the upper bound of the function domain (i.e. the length of `tokens`).

```
> type Tokens a = (Int -> a, Int)
> mkTokens tokens = (\n -> (a!n), ltok)
>   where a = listArray (0, ltok-1) tokens
>         ltok = length tokens
```

3.3 Symbols and rules

A symbol in the grammar is either a terminal of type `t` or a non-terminal of type `nt`, respectively. In most cases `nt` and `t` will both be `String`; but since Haskell lets us play polymorphically as long as we want, there is no need to be unnecessarily specific. Apart from these two, we will introduce a special, extra-grammatical dummy, useful when declaring axioms (see below). Predicates to recognize these three types will allow us help us classify symbols without pattern matching, allowing us to keep the symbol type abstract.

Representing rules is equally straightforward: a rule is basically a pair of type `(Symbol t nt, [Symbol t nt])`:

```
> data Symbol a b = T a | N b | D b deriving (Ord)
>
> isNonTerm :: Symbol a b -> Bool
> isNonTerm (N _)  = True
> isNonTerm  _     = False
>
> isTerm :: Symbol a b -> Bool
> isTerm (T _)  = True
> isTerm  _     = False
>
> isDummy :: Symbol a b -> Bool
> isDummy (D _) = True
> isDummy  _    = False
>
> newtype Rule a b = Rule ((Symbol a b), [Symbol a b])
>
> lhs :: Rule a b -> (Symbol a b)
> lhs (Rule (x, ys)) = x
>
> rhs :: Rule a b -> [Symbol a b]
> rhs (Rule (x, ys)) = ys
>
> lhsRhs :: Rule a b -> ((Symbol a b), [Symbol a b])
> lhsRhs (Rule (x,ys)) = (x,ys)
```

3.4 Grammar

Designing the grammar, in contrast, requires considerably more thought. It is much used; thus, it should be designed for efficiency and – even more importantly – for convenient updates to more efficient implementations yet to be written. We will thus make it an abstract data type (ADT).

From the description of chart parsing, we note that for top-down prediction, we will need efficient access to all rules matching a given lhs; and for bottom-up prediction, to all rules where a given category matches the first member of the rhs, which we will denote rhs1. Thus, these two central operations will have type


```
> lhsToRules :: Grammar a b -> Symbol a b -> [Rule a b]
> rhs1ToRules :: Grammar a b -> Symbol a b -> [Rule a b]
```

There are some efficiency pitfalls, however. In a typical CFG for natural language, $|N|$ is on the order of $10^1 - 10^4$, $|T|$ on the order of $10^3 - 10^5$. A given preterminal may have thousands of expansions – `lhsToRules` may thus easily return thousands of rules, at most one of which will match the current input symbol. Of course, we might filter them later, but much more efficient would be not to have them returned in the first place.

We therefore add another rule accessor, `lhsToRulesConstrained`, which in addition to `lhs` takes the current token `w` as argument. If `lhs` is not a preterminal, the function will behave just as `lhsToRules` does. If `lhs` is a preterminal, however, `lhsToRulesConstrained` instead searches in the precalculated, much smaller set of possible preterminals for `w`. Thus, for preterminals, `lhsToRulesConstrained grammar lhs w` will return either 0 or 1 rules, rather than $O(|T|)$.

```
> lhsToRulesConstrained :: Grammar a b -> Symbol a b -> a -> [Rule a b]
```

The current implementation is omitted here, but it first partitions P into P_n and P_t. Then both rule sets are indexed separately when the grammar is built, from lhs to rules and from rhs1 to rules.

Out of the four possible combinations, indexing from lhs to rules for preterminals often isn't very rewarding – for many natural language CFGs, where $|T| \gg |N|$, this means mapping a small set to a large, which is not very useful as far as indexing operations go (at most it might be used as a baseline for efficiency comparisons). On the other hand, it is instructive to note that we pay no price for including it – Haskell's lazy evaluation allows us to define structures just in case we'll need them later, without performance loss.

```
> data Grammar t nt = Grammar
>     { nTermGrammar :: M.Map (Symbol t nt) [Rule t nt]
>     , termGrammar  :: M.Map (Symbol t nt) [Rule t nt]
>     , termGrammarR :: M.Map (Symbol t nt) [Rule t nt]
>     , rhsFirst     :: M.Map (Symbol t nt) [Rule t nt]
>     }
>
> fromRuleLists :: (Ord a, Ord b) =>
>   ([Rule a b], [Rule a b]) -> Symbol a b -> Grammar a b
> fromRuleLists (nrules,trules) = Grammar ntg tg tgrev pts rhs1
>   where
>     ntg  = M.fromListWith (++) $ map (\ r@(Rule (k, _))  -> (k,[r])) nrules
>     tg   = M.fromListWith (++) $ map (\ r@(Rule (k, _))  -> (k,[r])) trules
>     tgrev= M.fromListWith (++) $ map (\ r@(Rule (v, [k])) -> (k,[r])) trules
>     rhs1 = M.fromListWith (++) $ map (\ r@(Rule (_, (h:_))) -> (h,[r])) nrules
```

```
>
```

Given these maps, the accessor functions above might look like

```
> lhsToRulesConstrained g sym w
>   | isNonTerm sym = concat $ M.lookup sym (nTermGrammar g)
>                   ++ filter ((==sym) . lhs) (rhs1ToRules g (T w))
>   | isTerm sym = []
>
>
> lhsToRules g sym
>   | isNonTerm g sym = concat $ M.lookup sym (termGrammar g)
>                     ++ M.lookup sym (nTermGrammar g)
>
> rhs1ToRules g sym
>   | isNonTerm sym = fromMaybe [] (M.lookup sym (rhsFirst g))
>   | isTerm sym = M.findWithDefault
>       (error $ "(rhs1ToRules): unknown terminal: " ++ show sym)
>       sym (termGrammarR g)
```

The grammar also provides functions for construction, inspection, validation, filtering, etc. We omit most of those here, but we will return to filtering issues in Section 4.

3.5 Items

The form we use to store established facts, or lemmata, corresponds to what we have said in Section 2. However, we will want to declare some indexing function(s) for efficient lookup in the chart, discussed below. We also add a field `ifound` for representing the parsed results space-efficiently (see Section 5).

```
> data Item a b =
>   Item { ifrom :: Int
>        , ilhs :: (Symbol a b)
>        , ifound :: [(Int, Symbol a b, Int)]
>        , itofind :: [Symbol a b]
>        , idot :: Int
>        } deriving (Eq,Ord)
>
>
> indexDotToFind :: Item a b -> (Int, Maybe (Symbol a b))
> indexDotToFind i = (idot i, listToMaybe $ itofind i)
```

3.6 Store

The chart and the grammar are heavily used and will carry most of the responsibility for parsing performance. Thus, for the same reasons as for the grammar, an ADT is appropriate. However, in this ADT we will bundle the chart with an *agenda*. The chart holds items which have already acted as triggers for inference rules, while the agenda contains items which have been

inferred from some other trigger, but not yet been explored themselves. Less high-flown, chart and agenda might be called 'done' and 'todo' lists. Together, the (chart, agenda) pair will be referred to as the *store*.

The main operations on the store will be deleting and returning a single trigger T from the agenda (more on which in 3.8); deriving a set of items derived from T and adding them to the agenda; and adding T to the chart.

Deriving items involves testing the different inference rules against T and (for `combine`) the chart. Since the `combine` operation will search for customers for some category C at some dot position i, it is convenient to index the chart on (i, C). We can then export a `findCustomersAtFor` function directly.

In no case should we have duplicates in either chart or agenda, lest we risk paying an exponential performance price (Shieber, Schabes & Pereira 1995). If we eliminate redundancy in the agenda, however, we get a duplicate-free chart automatically. Again, this calls for indexing considerations. For simplicity, the current implementation reuses the (i, C) indexing mechanism above for searching the chart. However, although this may work for simpler cases, more varied indexing schemes are needed to guarantee logarithmic worst-case access time.

The store also exports several convenience functions for finding passive edges (confirmed categories) beginning and/or ending at certain states: `findCompleted`, `findCompletedAt`, etc, not meant to be heavily used.

3.7 Deduction engine

Imperatively, pseudo-code for the actual deduction process may look like:

1) [initStore]: initialize the store: the chart as empty, the agenda with the axioms of the parsing logic
2) [exhaustAgenda]: if agenda is empty, stop and return chart; otherwise:
   a) delete a trigger from the agenda
   b) try the inference rules on the trigger and add any unseen items to the agenda
   c) add the trigger to the chart
3) repeat from 2

Given the chart operations above, the Haskell phrasing is of comparable size:

```
> import qualified Store as S
>
> initStore ::  (Ord a, Ord b) => Grammar a b -> Tokens a -> S.Store a b
> initStore grammar tokens = S.addItemsTodo (axioms grammar tokens) S.mkEmpty
>
> exhaustAgenda :: (Ord a, Ord b) => [IRule a b] -> S.Chart a b -> S.Chart a b
> exhaustAgenda irules chart= until (S.isEmptyTodo) eAgenda chart
```

```
>   where
>     eAgenda chart = S.addItemDone trigger . S.addItemsTodo newItems $ chart'
>       where
>         newItems = filter (not . S.inChart chart)
>                    $ concatMap (\rule -> rule chart trigger) irules
>         (trigger, chart') = S.deleteTrigger chart
```

An interesting question is precisely what trigger to delete in 2a. It will be of importance only when the trigger is being combined with already known facts, i.e. in the `complete` inference rule (from Figure 3):

$$\frac{[A \to \alpha \cdot B\beta, i, k], [B \to \gamma \cdot, k, j]}{[A \to \alpha B \cdot \beta, i, j]} \quad \text{(inference rule: complete)}$$

Here, $i \leq k < j$. In principle, the trigger may be either of the two rule antecedents, and we will have to search for the other one. However, if we consistently choose the minimum item in the agenda as trigger, we will know that state i is entirely processed when we get to state j. We will thus be able to make a left-to-right pass, processing lower states before higher and passive edges before active. This means that the trigger will always be the second of the antecedents in the rule. It also means that the first clause of the translation of the `complete` logic below (Appendix A) will never be called and could be deleted (although one might prefer to keep it for declarative purity).

```
>     complete chart (Item i a alpha ((sym@(N _)):beta) k) = do
>       Item _k symbol gamma [] j <- S.findCompletedBeginningAt chart k
>       guard $ symbol == sym
>     ------------------------------------------------------------------
>       return $ Item i a ((k, sym, j):alpha) beta j
>
>     complete chart (Item k (sym@(N _)) gamma [] j) = do
>       Item i a alpha (_sym:beta) _k <- S.findCustomersAtFor chart k sym
>     ------------------------------------------------------------------
>       return $ Item i a ((k, sym, j):alpha) beta j
>
>     complete chart _ = []
```

3.8 Defining a parsing logic

With the rest of the machinery in place, implementing a particular chart parsing algorithm now amounts to specifying a particular item form and a particular parsing logic for such items. In contrast to the rest of the code

described here, parsing logics are expected to be changed and modified often; therefore, doing so should be easy and intuitive.

To this end, we exploit the list monad; this allows us to simulate Prolog-style nondeterminism in a notation which at least reminds of natural deduction. For instance, see below the `predict` inference rule from Figure 3 and a possible rendering in the chart parsing logic.

$$\frac{[A \to \alpha \cdot B\beta, i, j]}{[B \to \cdot\gamma, j, j]} \; B \to \gamma \quad \text{(inference rule: predict)}$$

```
> -- defined elsewhere: mapSnd f (a, b) = (a, f b)
>
> inferenceRules grammar (tokens, ltok) = [predict, ...]
>   where
>
>     predict chart (Item i rhs1 _ toFind j)
>       | not (null toFind) = []
>       | i >= ltok = []
>       | otherwise = do
>           rule <- rhs1ToRules grammar rhs1
>           let (lh, rht)  = mapSnd tail . lhsRhs $ rule
>     ------------------------------------------------------------------
>           return $ Item i lh [(i, rhs1, j)] rht j
```

Rather than stepping through the rest of the logic of Figure 3 explicitly, we express chart parsing according to Kilbury 1985 as a parsing logic in Appendix A.

4 Filtering

Blindly following the inference rules of our parsing logic, the bottom-up approach will use the `predict` inference rule to find new lemmata, some of which can never form part of a sentence. Similarly, top-down generation will `predict` sequences of preterminals (or even words) without considering the input sentence at all. Many of the items thus generated are clearly dead ends, and it would thus be beneficial to filter them out even before they are predicted. In fact, the already described `lhsToRulesConstrained g nt token` is nothing but a primitive but effective filter: if the nonterminal `nt` in grammar `g` happens to be a preterminal, then at most one rule is returned, namely the one matching `token` (intended to be the current word).

We may pursue this further, however. In *left-corner parsing*, first suggested by Rosenkrantz & Lewis 1970, we define a relation $\triangleright$, pronounced 'left-corner', such that

$$\alpha \triangleright \beta \leftrightarrow \exists\gamma.\alpha \Rightarrow \beta\gamma$$

Although α, β could in principle be of any length, in practice the cost of calculating the relation will quickly dominate the benefit for longer strings. We will restrict ourselves to the simplest case where they are single non-terminals (and thus better written A, B). Informally, $A \triangleright B$ iff B can be the first category in some A phrase. If we build a digraph D from the grammar G, with nodes for nonterminals and an edge from A to B iff $A \rightarrow B\beta \in P$, then the left-corner relation can be computed as the transitive closure T of D. We may also find useful the transpose graph of T, the converse relation *right-corner* $B \triangleleft A \leftrightarrow A \triangleright B$. Extending this relation to tokens, we get a *wordlink* relation, $w \blacktriangleleft A$: w has wordlink relation to A whenever w can be the first token in some A phrase. For instance, in the example grammar (Figure 1), we have $S \triangleright \{NP, N, AdjP, Adj\}$; $Adj \triangleleft \{AdjP, NP, S\}$; *intensivt* $\blacktriangleleft \{Adj, Adv, AdjP, NP, S\}$.

None of these relations depend on the input string: thus, they can be precalculated from the grammar and stored, for instance as closures in the `Grammar` data type. Again, thanks to Haskell's laziness we don't need to worry about wasting efforts precalculating things without using them – with little performance cost, we can define whatever predicates we might think useful and let the implementor of the parsing logic choose between them.

With this machinery in place, it is rather straightforward to declare filters for the predict inference rule. The `Control.Monad` library offers `guard` for conditional monadic executions; used in the list monad, it will yield Prolog-like nondeterministic backtracking by sequenced conjunctions of predicates.

Below is again the `predict` fragment of Appendix A, now with two filters added. By omitting one or both of them or changing their order, we get five different variations.

```
>   predict filters chart (Item i rhs1 _ toFind j)
>     | not (null toFind) = []
>     | i >= ltok = []
>     | otherwise = do
>         rule <- rhs1ToRules grammar rhs1
>         let (lh, rht)  = mapSnd tail . lhsRhs $ rule
>         guard $ bottomUp rht
>         guard $ topDown lh
> ------------------------------------------------------------------------
>         return $ Item i lh [(i, rhs1, j)] rht j
```



```
>         where
>           bottomUp [] = True
>           bottomUp (r:_) = j < ltok && wordLink grammar r (T $ tokens j)
>           topDown lh = any (not null) $ map (S.findCustomersAtFor chart i)
>                                             (rightCorners grammar lh)
```

Generally, filters will give smaller charts, at the cost of some calculation overhead. The smaller charts may additionally save some later calculations. However, the optimal number and order of filters depend on the grammar and the input; and filters need not come with any savings at all.

5 Constructing parse trees

Parsing usually involves not only recognition of a given string of tokens with respect to some grammar, but also reconstruction of all possible parse trees that the grammar licenses for the string. Of course, some grammars and some strings will yield exponentially many such trees, and a naive implementation which stores entire trees directly in the chart will consequently run in exponential space (and time).

A better approach is to add a field to each `Item`, so that each passive (i.e., completed) edge from i to j can be represented by a tuple. That is, as soon as we have found some category between i and j, we record it in the completed item, permitting us to store exponentially many trees in polynomial space.

When the chart is finished we filter out active (i.e., non-finished) edges. The remaining items, corresponding to confirmed phrases, can again be described by a CFG, only now with the original categories replaced by tuples `(category, from, to)`; if the number of possible parse trees is finite, this derived grammar will be non-recursive. If we, as above, represent this simpler grammar by an `lhsToRules` function (cf. Section 3.4), a generic folding function for parses, `foldParse`, might look like:

```
> foldParse  ::
>  --given a non-terminal nt, return relevant grammar rules
>   (Symbol a b -> [Rule a b])
>  --what to do when we encounter a terminal (T t)
>   -> (Symbol a b -> b2)
>  --how to combine the results yielded by different rules
>   -> ([b1] -> b2)
>  --how to combine the results given by different cats in rhs of a single
  production
>   -> (Symbol a b -> [b2] -> b1)
>  --where to start (some nt)
>   -> Symbol a b
>  --folding result
>   -> b2
>
> foldParse expand termF combineRulesF combineRHS symbol = f symbol
>     where f sym | isTerm sym = termF sym
>                 | otherwise  = combineRulesF (map aux (expand sym))
>           aux rule = combineRHS (lhs rule) (map fgt (rhs rule))
```

For instance, we can use `foldParse` to count parse trees:

```
> countPTrees :: (Ord b, Ord a, Show b, Show a) => [I.Item a b] -> Int ->
  Integer
> countPTrees chart n = foldParse
>   (lhsToRules grammar)
>   (const 1)                       -- a -> Integer
>   sum                             -- [Integer] -> Integer
>   (\_ sums -> product sums)       -- [Integer] -> Integer
>   (startSymbol grammar)
>     where grammar = chartToGrammar chart n
>
```

Or we can use it to build parse trees one by one, for later formatting and printing.

```
> data ParseTree a b = PLeaf a | PNode b [ParseTree a b] deriving Ord
>
> buildTrees :: (Ord a, Ord b) =>
>     Grammar a b -> Symbol a b -> [ParseTree (Symbol a b) (Symbol a b)]
> buildTrees grammar = foldParse
>   (lhsToRules grammar)
>   (\term -> [PLeaf term])       -- Symbol a b -> [ParseTree]
>   concat                        -- [[ParseTree]] -> [ParseTree]
>   cproduct                      -- [[ParseTree]] -> [ParseTree]
>     where cproduct lhs rhss = map (PNode lhs) (crossProduct rhss)
```

6 Space and time complexity

For a particular position (i.e., Earley state) k in the input string, each element in the rhs of each phrase category rule may yield an edge for each previous position $j < k$. Thus, the chart corresponding to position k has size $O(n|P_n|\delta)$ where δ is length of the longest rhs, and the size of the entire chart is $O(n^2|P_n|\delta)$. Excepting the initial $O(n)$ axioms, at any time the agenda can contain items referring to state k only; it is thus also of size $O(n|P_n|\delta)$.

As for time, each item on the agenda, in total $O(n^2|P_n|\delta)$, will act as a trigger exactly once, and it may yield at most $O(n|P_n|\delta)$ new items through `complete` (and another $O(|P_n|)$ via `predict`). These must be checked for redundancy against the agenda. Constant-time lookup for the chart and grammar operations would give us a total of $O(n^3|P_n|^2\delta^2)$, matching imperative solutions. Using maps, we retain referential transparency but even with the ideal indexing scheme (cf Section 3.6), we pay a slight performance cost: $O(n^3|P_n|^2\delta^2 \lg(n|P_n|\delta))$.

We have relied on finite maps, with inherent logarithmic access time. Another candidate is hash tables, which in theory should give constant rather than logarithmic time for lookups. However, hash tables use memory destructively; while sometimes useful, they therefore do not fit very well into the pure functional paradigm. Furthermore, in the main Haskell implementation available at the time of writing (ghc 6.8), in practice they are often slower than maps.

The usefulness of the filters of Section 4 also relies on efficient implementations of the relations `wordLink` and `rightCorner` relation. Again, we prefer finite maps.

Generation of parse trees is of course exponential, as there may be exponentially many of them. However, thanks to laziness, the space requirements for enumerating the parse trees are modest.

7 Conclusion. Future directions

The described deduction engine aims for purity, modularity, and declarativity rather than speed, and there is a slight performance penalty. We have so far not carried out any rigourous benchmarks; while the engine seems reasonably useful for simple practical purposes, an obvious next step involves more principled comparisons. We were careful to design the critical structures (Grammar, Chart) as ADTs to make room for future improvements (although the interfaces still are somewhat experimental).

We simulated non-determinism passably by exploiting the `do` notation and the list monad. Less elegantly, we also relied on explicit state passing. A more uncompromising and challenging approach, not pursued here but certainly worth investigating, implies a formulation which does not depend on global state, as Ljunglöf 2004 does for Kilbury-style chart parsing. An even greater challenge, following Shieber et al., is of course to expand the framework to express more diverse grammar formalisms, such as tree-adjoining and categorial grammars.

Acknowledgement

Inspiration to this paper came from a proof-of-concept implementation by Jan van Eijck (correct, but $O(a^n)$, see van Eijck 2004); and a course assignment by Peter Ljunglöf.

References

Earley, Jay. 1970. 'An efficient context-free parsing algorithm'. *Comm. ACM* 13:2, 94–102.

Eijck, Jan van. 2004. 'Deductive parsing in Haskell'. Unpublished manuscript (version 20040223, retrieved 20070804 from http://homepages.cwi.nl/~jve/).

Hudak, Paul, John Peterson & Joseph Fasel. 2000. *A gentle introduction to Haskell version 98*. Available at http://www.haskell.org/.

Kasami, Tadao. 1965. *An efficient recognition and syntax algorithm for context-free languages*. Technical report AFCLR-65-758. Bedford, MA: Air Force Cambridge Research Laboratory.

Kilbury, James. 1985. 'Chart parsing and the Earley algorithm'. In U. Klenk (ed.), *Kontextfreie Syntaxen und wervandte Systeme*. Tübingen: Niemeyer.

Ljunglöf, Peter. 2004. 'Functional chart parsing of context-free grammars'. *Journal of Functional Programming* 14:6, 669–680.

Peyton Jones, Simon, John Hughes et al. 1999. *Report on the programming language Haskell 98*. University of Yale. Available at http://www.haskell.org/.

Rosenkrantz, Daniel & Philip Lewis II. 1970. 'Deterministic left corner parsing', *Proc. 11th Annual Symposium on Switching and Automata Theory*, 139–152.

Shieber, Stuart, Yves Schabes & Fernando Pereira. 1995. 'Principles and implementation of deductive parsing'. *Journal of Logic Programming* 24:1–2, 3–36.

Younger, Daniel. 1967. 'Recognition of context-free languages in time n^3'. *Information and Control* 10:2, 189–208.

Appendix A.

Kilbury (1985) bottom-up parsing expressed in a functional deductive system, assuming `Item` declared as before. The two `guard` lines add bottom-up and top-down filtering. They can change places, or either one or both of them can be omitted, yielding five different variations of the basic algorithm.

```
> -- defined elsewhere: mapSnd f (a, b) = (a, f b)
>
> goal :: (Eq a, Eq b) => Grammar a b -> Tokens a -> Item a b -> Bool
> goal gr (_, ltok) (Item 0 found _ [] n) = isStartSymbol gr found && n == ltok
> goal _ _ _ = False
>
> axioms grammar (tokens, ltok) = start ++ scan
>   where
>     start = let N x = startSymbol grammar in [Item 0 (D x) [] [N x] 0]
>     scan  = do
>          (j, t) <- map (\i -> (i, tokens i)) [0..ltok-1]
>          ------------------------------------------------
>          return $ Item j (T t) [(j, (T t), j+1)] [] (j+1)
>
> inferenceRules grammar (tokens, ltok) = [predict, complete]
>   where
>
>     complete chart (Item i a alpha ((sym@(N _)):beta) k) = do
>       Item _k symbol gamma [] j <- S.findCompletedBeginningAt chart k
>       guard $ symbol == sym
>     ------------------------------------------------------------------
>       return $ Item i a ((k, sym, j):alpha) beta j
>
>     complete chart (Item k (sym@(N _)) gamma [] j) = do
>       Item i a alpha (_sym:beta) _k <- S.findCustomersAtFor chart k sym
>     ------------------------------------------------------------------
>       return $ Item i a ((k, sym, j):alpha) beta j
>
>     complete chart _ = []
>
>     predict chart (Item i rhs1 _ toFind j)
>       | notNull toFind = []
>       | i >= ltok = []
>       | otherwise = do
>          rule <- rhs1ToRules grammar rhs1
>          let (lh, rht)  = mapSnd tail . lhsRhs $ rule
>          guard $ bottomUp rht
>          guard $ topDown lh
>     ------------------------------------------------------------------
>          return $ Item i lh [(i, rhs1, j)] rht j
>           where
>            bottomUp [] = True
>            bottomUp (r:_) = j < ltok && wordLink grammar r (T (tokens j))
>            topDown lh = any notNull $
>              map (S.findCustomersAtFor chart i) (lh : rightCorners grammar lh)
```