JFP **14** (6): 669–680, 2004.
DOI: 10.1017/S0956796804005106 Printed in the United Kingdom

FUNCTIONAL PEARL

Functional chart parsing of context-free grammars

PETER LJUNGLÖF

Department of Computing Science, Göteborg University and Chalmers University of Technology, Gothenburg, Sweden
(*e-mail:* peb@cs.chalmers.se)

Abstract

This paper implements a simple and elegant version of bottom-up Kilbury chart parsing (Kilbury, 1985; Wirén, 1992). This is one of the many chart parsing variants, which are all based on the data structure of charts. The chart parsing process uses inference rules to add new edges to the chart, and parsing is complete when no further edges can be added. One novel aspect of this implementation is that it doesn't have to rely on a global state for the implementation of the chart. This makes the code clean, elegant and declarative, while still having the same space and time complexity as the standard imperative implementations.

1 Introduction

Chart parsing is really a family of parsing algorithms, all based on the data structure of charts, which is a set of known facts called edges (Kay, 1986; Wirén, 1992). The algorithm is descended from the algorithms of Earley (1970) and Kasami (1965) and Younger (1967). The parsing process uses inference rules to add new edges to the chart, and the parsing is complete when no further edges can be added. In recent years the algorithm has been generalized to deductive parsing (Shieber *et al.*, 1995) and parsing schemata (Sikkel, 1997).

We implement an efficient functional version of bottom-up chart parsing *à la* Kilbury (1985). The novel aspect is that it doesn't have to rely on a global state for the implementation of the chart. The implementation divides the chart into a list of Earley states, named after the top-down Earley parsing algorithm. This makes it possible to parse the input incrementally, building each state in sequence, which in turn gives clean, elegant and declarative code. The elegance is not traded for efficiency, since the worst-case complexity is cubic in the length of the input, which is as good as any imperative implementation.

1.1 Context-free grammars

The standard way to define a context-free grammar is as a tuple $G = \langle N, \Sigma, P, S \rangle$, where N and Σ are disjoint sets of *nonterminal* and *terminal* symbols respectively,

S	⟶	*NP VP*
VP	⟶	*Verb* | *Verb NP* | *VP PP*
NP	⟶	*Noun* | *Det Noun* | *NP PP*
PP	⟶	*Prep NP*
Verb	⟶	*flies* | *like*
Noun	⟶	*flies* | *time* | *arrow*
Det	⟶	*an*
Prep	⟶	*like*

Fig. 1. The example grammar.

P is a set of *productions* 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 $A \to \alpha$ where $A \in N$ is a nonterminal and $\alpha \in V^*$ is a sequence of symbols.

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 $B \to \beta \in P$. A *sentence* is a phrase recognized by the start symbol, i.e. an S phrase. The *language* accepted by a grammar is the set of sentences of that grammar.

In this paper, we shall assume that the grammar contains no empty productions $A \to \epsilon$. We will also assume that the terminal symbols only appear in unit productions, which are of the form $A \to t$, where $A \in N$ and $t \in \Sigma$. This is no severe restriction, since any context-free grammar can be easily transformed into this form.

1.2 An illustrative example

Figure 1 shows an example of a context-free grammar recognizing simple English sentences. The categories are *S*, *VP*, *NP*, *PP*, *Verb*, *Noun*, *Det* and *Prep*, (standing for sentence, verb phrase, noun phrase, prepositional phrase, verb, noun, determiner and preposition respectively), of which *S* is the starting category. The grammar is ambiguous, both at the lexical level (one word can have several different categories) and at the phrasal level (one phrase can have several different syntactical structures).

1.3 Representing grammars in Haskell

Since the terminals only appear in unit productions, we can split the set of productions into a set of nonterminal productions and a function mapping each terminal to a set of nonterminals. A context-free grammar will then consist of the terminal function, the nonterminal productions and the starting category.

```
type Terminal c t    =  t → {c}
type Productions c   =  {(c, [c])}
type Grammar c t     =  (Terminal c t, Productions c, c)
```

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

```
terminal "an"     =  { Det }          productions  =  { (S,  [NP, VP]),
terminal "arrow"  =  { Noun }                           (VP, [Verb]),
terminal "flies"  =  { Noun, Verb }                     (VP, [Verb, NP]),
terminal "like"   =  { Prep, Verb }                     (VP, [VP, PP])
terminal "time"   =  { Noun }                           (NP, [Noun]),
terminal _        =  { }                                (NP, [NP, PP])
                                                        (PP, [Prep, NP]) }
```

Fig. 2. The example grammar in Haskell.

Note that we abuse Haskell notation somewhat and write sets with braces in analogy to lists. Our example grammar can be defined in Haskell as shown in Figure 2. Of course, there are more efficient ways to encode the grammar functions, such as search trees or hash tables.

1.4 Representing sets in Haskell

The type of sets will be an abstract data type in this paper. In analogy to lists, we write it with braces. The main operations on sets we require are set union ($\cup$) and set difference ($\setminus$).

$$\begin{array}{lll} (\cup) & :: & Ord\ c \Rightarrow \{c\} \rightarrow \{c\} \rightarrow \{c\} \\ (\setminus) & :: & Ord\ c \Rightarrow \{c\} \rightarrow \{c\} \rightarrow \{c\} \end{array}$$

We assume that there are efficient coercion functions between sets and sorted lists. Since they are obvious from the context, we won't clutter up the code with them. Instead we use braces around a sorted sequence, when forming a set.

We also use set comprehension notation in some places, but only for mapping and filtering. Furthermore, we are careful that a set comprehension preserves ordering.

When building the chart we will make use of a function *limit* that builds the minimal fixed point set from an initial set and a function giving new elements. This function will terminate if there is a finite fixed point, which in the case of chart parsing will be the final chart. The function *union* used in the definition calculates the union of a list of sets.

```
limit :: Ord a ⇒ (a → {a}) → {a} → {a}
limit more start                 =  limit' start start
  where limit' old new
                | new' == { }    =  old
                | otherwise      =  limit' (new' ∪ old) (new' \ old)
              where new'         =  union (map more new)
```

One simple and efficient implementation of sets is sorted lists. In that case we can replace all braces with list brackets.

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

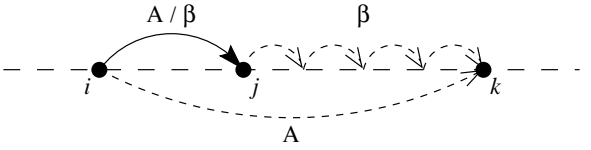
2 Chart parsing

The basic idea in chart parsing is to build a set of edges from a small number of inference rules. The edges say what kind of phrase a certain part of the input might be. The inference rules say how to combine the edges into new ones. Depending on the parsing strategy the inference rules might look different, but the main idea always remains the same.

2.1 Edges and the chart

A chart is a set of items called edges. Each edge is of the form $\langle i, j : A/\beta \rangle$, where $0 \leqslant i \leqslant j$ are integers, A is a category and β is a sequence of categories. If β is empty the edge is called passive, and we write it as $\langle i, j : A \rangle$; otherwise it is called active. For an input of n words, we create $n + 1$ nodes labelled $0 \ldots n$. The ith word in the input will then span the nodes $i - 1$ to i.

The intended meaning of a passive edge $\langle i, j : A \rangle$ is that we have found the category A spanning the nodes i and j. The meaning of an active edge $\langle i, j : A/\beta \rangle$ is that if we can find the categories β between j and some k, then we will know that A spans between i and k. The following diagram illustrates this, where the dashed arrows denote the edges still to be found:



The key idea of chart parsing is that we start with an empty chart, to which we add new edges by applying some inference rules. We write the rules in a natural deduction style, where the following rule means that if the edges $e_1 \ldots e_k$ are in the chart, and the property ϕ holds, add the edge e to the chart.

$$\frac{e_1 \ \ldots \ e_k}{e} \ \phi$$

The property is of the form $A \to \beta$, which means that this particular production is in the grammar. We write t_i for the ith token in the input sequence.

2.2 Kilbury bottom-up chart parsing

There are different strategies for chart parsing, which are reflected in the inference rules. In the bottom-up strategy, we start by adding the input tokens as passive edges, and then build the chart upwards. Since we are assuming that there are no empty productions $A \to \epsilon$ in the grammar, we do not have to consider that possibility in the inference rules.

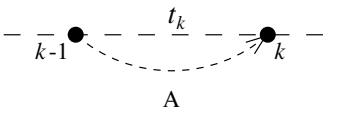
Scan

All the categories for the kth input token t_k are added as passive edges spanning the nodes $k - 1$ and k. Since we assume the terminals only appear in unit productions,

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

this is the only necessary inference rule dealing with terminals.

$$\frac{}{\langle k{-}1, k : A\rangle} \quad A \to t_k$$

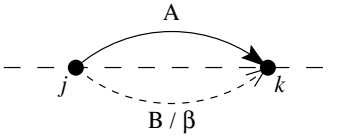


The diagram illustrates the inference rule, where the dashed arrow denotes the edge to be added. In the following two diagrams, the edges above the line correspond to the triggering edges.

Predict

If we have found a passive edge for the category A, spanning the nodes j and k, and there is a production in the grammar that looks for an A, we add that production as an edge. And since we have found the category A in the right-hand side, we only have the categories after A left to look for.

$$\frac{\langle j, k : A\rangle}{\langle j, k : B/\beta\rangle} \quad B \to A\beta$$

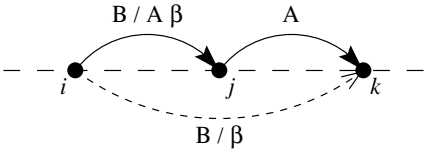


This particular variant of bottom-up parsing is called Kilbury parsing, first described in Kilbury (1985). The difference to the traditional bottom-up strategy is that we do not add $\langle j, j : B/A\beta\rangle$ as an edge, since we end up with the edge above. Apart from saving some extra work, it will also keep the chart acyclic.

Combine

If we have an active edge looking for an A at the jth node, and there is a passive edge labelled A spanning j and k, we can move the active edge forward to the kth node.

$$\frac{\langle i, j : B/A\beta\rangle \quad \langle j, k : A\rangle}{\langle i, k : B/\beta\rangle}$$



The parsing succeeds if there exists a passive edge for the starting category, spanning the whole chart; that is if $\langle 0, n : S\rangle$ is in the chart, where n is the number of input tokens.

2.3 The chart as a directed graph

A nice way to visualize a chart is as a directed graph. Since the grammar doesn't contain any empty productions, the chart will be acyclic. This in turn makes it possible to implement the algorithm elegantly in Haskell.

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

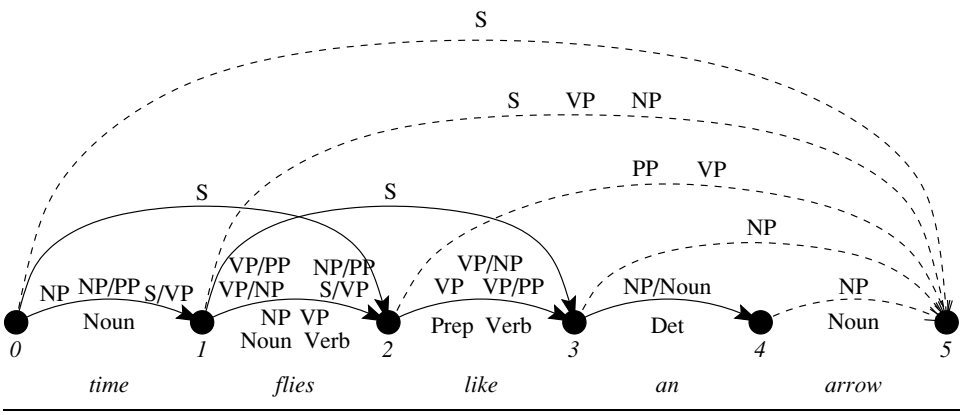


Fig. 3. The chart after the first four words have been incorporated.

Recalling our example grammar in section 1.2, we can depict how the chart will look like for a given sentence. In Figure 3 we see the chart after the first four words in the sentence "time flies like an arrow". The dashed arrows denote the edges that will be created when the fifth word has been incorporated. Each arrow actually symbolizes a set of edges, which are written above and below the edge. For example, there are eight edges between node 1 and 2, of which four are active and four passive.

3 Kilbury parsing in Haskell

We shall implement chart parsing in an incremental way, by starting with the first input token and then applying all the inference rules. Then we add the second token and apply the inference rules again; and so on until we have added the last token, when we will have a final chart.

3.1 Building the chart

The incremental strategy makes it possible to represent the chart as a list of states, each state being all the edges ending in a particular node. The states will be called Earley states, since this is how to represent the parsing state in Earley's original parsing algorithm, which in turn can be seen as an implementation of top-down chart parsing. So, a chart will be a list of sets of edges.

$$\begin{array}{lcl} \textbf{type}\ \mathit{Chart}\ c & = & [\mathit{State}\ c] \\ \textbf{type}\ \mathit{State}\ c & = & \{\mathit{Edge}\ c\} \end{array}$$

The edge $\langle j,k : A/\alpha \rangle$ is a 4-tuple of the two nodes j and k, the category A and the list of categories α. But the ending node k need not be remembered, since it is implicit in the position of the edge's state in the chart list.

$$\textbf{type}\ \mathit{Edge}\ c \quad = \quad (\mathit{Int}, c, [c])$$

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

The main function builds a chart from a given grammar and the input sequence.

```
buildChart                :: Ord c ⇒ Grammar c t → [t] → Chart c
buildChart (terminal, productions, _) input = finalChart
  where finalChart        = map buildState initialChart
        initialChart      = ... defined below ...
        buildState        = ... defined below ...
```

The inference rules can be translated to set comprehensions in a natural way, which is done in the definitions of *initialChart* and *buildState* below. For clarity, these functions are presented on the top-level, even though they are actually in the scope of the grammar and the input sequence.

The initial chart

The initial chart consists of the results of the Scan inference rule applied to the input tokens. Each Earley state k (except for the empty 0th state) will consist of all edges $\langle i,k : A\rangle$ such that $i = k - 1$ and $A \to t_k$, where t_k is the kth input token.

```
initialChart  :: Chart c
initialChart  = { } : map initialState (zip [0 . .] input)
  where initialState (i, sym) = { (i, cat, []) | cat ∈ terminal sym }
```

Observe that we use a set comprehension here, which is legal, since the corresponding map does not change the order between the elements, as explained in section 1.4.

Building the final state

Both the Predict and the Combine inference rule only apply to passive edges, which means that an active edge will not lead to any new edges being added to the chart. The Combine rule only applies to passive edges because the active edge $\langle i, j : B/A\beta\rangle$ ends in the jth node, and since we have no cycles in the graph, $j < k$, where k is the Earley state to which the new edge $\langle i,k : B/\beta\rangle$ will be added.

```
buildState                :: State c → State c
buildState                = limit more
  where more (j, a, [])   = { (j, b, bs) | (b, a' : bs) ∈ productions, a == a' }
                          ∪ { (i, b, bs) | (i, b, a' : bs) ∈ finalChart !! j, a == a' }
        more _            = { }
```

The function *buildState* calculates the minimal fixed point set via the *limit* function defined in section 1.4. The auxiliary function *more* applies the Predict and Combine rules to an edge, giving a set of new edges to be added to the state.

The first set comprehension corresponds to the Predict inference rule, and the second to the Combine rule. The Predict rule only has to find the productions in the grammar that are looking for the found category a, and the Combine rule searches in the previous jth Earley state for the active productions looking for an a.

We also have to show that the two set comprehensions are legal, i.e. that the mappings retain the order between the elements. But this is true since the variables

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

a and *j* are fixed, and the order between the other variables is retained in the comprehensions.

Observe that *buildState* makes use of *finalChart*, which in turn is built by calling the function *buildState*. This is permitted because *buildState* only looks up previously built states in *finalChart*, and we use lazy evaluation.

3.2 Extracting the parse trees

To make things more interesting, we shall calculate parse trees from the final chart. Parse trees record the syntactic structure of the input, and we will use a general definition of trees consisting of a parent node and a list of child trees. For simplicity we skip the terminal leaves and use an empty children list instead.

$$\textbf{data}\ \mathit{Tree}\ c \quad = \quad \mathit{Node}\ c\ [\mathit{Tree}\ c]$$

When extracting the parse trees, we only need the passive edges from the chart. Further definitions become simpler if we use a special type for the passive edges.

$$\textbf{type}\ \mathit{Passive}\ c \quad = \quad (\mathit{Int}, \mathit{Int}, c)$$

To build the parse trees from the chart, we form the list *edgeTrees* by pairing each passive edge with the parse trees corresponding to that edge. The reason for this is that while building the parse trees for an edge, we might want to look up the parse trees for another edge.

$$\begin{array}{l}
\mathit{buildTrees} \ :: \ \mathit{Grammar}\ c\ t \to [t] \to \mathit{Chart}\ c \to [(\mathit{Passive}\ c, [\mathit{Tree}\ c])] \\
\mathit{buildTrees}\ (\mathit{terminal}, \mathit{productions}, _)\ \mathit{input}\ \mathit{chart} \quad = \quad \mathit{edgeTrees} \\
\quad \textbf{where}\ \mathit{edgeTrees} \quad = \quad [\ (\mathit{edge}, \mathit{treesFor}\ \mathit{edge}) \mid \mathit{edge} \in \mathit{passiveEdges}\ ] \\
\qquad\quad \mathit{passiveEdges} \quad = \quad [\ (i, j, \mathit{cat}) \mid \ (j, \mathit{state}) \in \mathit{zip}\ [0\ ..]\ \mathit{chart}, \\
\qquad\qquad\qquad\qquad\qquad\qquad (i, \mathit{cat}, []) \in \mathit{state}\ ] \\
\qquad\quad \mathit{treesFor} \quad = \quad \ldots \mathit{defined\ on\ next\ page} \ldots \\
\qquad\quad \mathit{children} \quad = \quad \ldots \mathit{defined\ on\ next\ page} \ldots
\end{array}$$

The function *treesFor* constructs all the parse trees of an edge $\langle i, j : A \rangle$. It will in turn make use of the function *children* which looks up all the children of that tree. Since *edgeTrees* is used for that, all functions have to be in scope of each other, but for clarity we present the function definitions separately.

Collecting the trees for an edge

The function *treesFor* constructs all the parse trees of an edge $\langle i, j : A \rangle$ by finding a path from i to j for each production $A \to \alpha$. The function *children* finds the path for α while collecting the parse trees for all the visited edges. Since there can be many different paths for α between i and j, we can get many sequences of trees as the result of the *children* function, and each of these sequences is used to form a parse tree for the given edge.

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

There is also the possibility that the edge was created by the Scan inference rule, in which case we just add a parse tree with no children.

```
treesFor              ::  Passive c → [Tree c]
treesFor (i, j, cat)  =   [ Node cat trees |
                            (cat′, rhs) ∈ productions,  cat == cat′,
                            trees ∈ children i j rhs ]
                      ++  [ Node cat [] |
                            i == j − 1,
                            cat ∈ terminal (input !! i) ]
```

Collecting the children

To collect the trees of the children, we make use of lazy evaluation and simply look up the trees in the list *edgeTrees* of edges and trees that is being computed.

```
children                ::  Int → Int → [c] → [[Tree c]]
children i k []         =   [ [] | i == k ]
children i k (cat : cats) =  [ tree : rest | i ⩽ k,
                               ((i′, j, cat′), trees) ∈ edgeTrees,
                               i == i′,  cat == cat′,
                               rest ∈ children j k cats,
                               tree ∈ trees ]
```

Observe that it is important to call the functions in the correct order – we must collect the rest of the children trees (between j and k), before extracing the tree to be put in front (between i and j). Otherwise we get into an infinite loop because we examine the trees before they are constructed.

3.3 The final parsing function

Finally, we can collect the parse trees that correspond to an edge for the starting category spanning the whole input.

```
parse  ::  Ord c ⇒ Grammar c t → [t] → Maybe [Tree c]
parse grammar@(_, _, start) input
       =   lookup (0, length input, start) edgeTrees
  where edgeTrees  =  buildTrees grammar input finalChart
        finalChart =  buildChart grammar input
```

4 Discussion

In this paper we have presented a simple, efficient and purely functional version of bottom-up Kilbury chart parsing. The formulation of the inference rules makes it possible to keep the chart acyclic, which in turn makes it possible to implement efficient lookup for previously found edges. The only abstract data structure we have used is the type of sets, and for these a simple sorted list representation is sufficient.

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

In this final section we first discuss how to improve the efficiency by using finite maps, such as search trees or hash tables. Then we discuss the space and time complexity of the implementation and show that it is as efficient as any imperative implementation. We also discuss where lazy evaluation is used in the definitions.

4.1 Improving efficiency

There are some ways to improve the efficiency of our implementation. Since we are building the chart incrementally – building one Earley state fully before starting to build the next – we can transform the previous states into more efficient data structures. All these efficiency-improving transformations involve creating a finite map from a list. For this we need an efficient implementation of finite maps, such as search trees or hash tables.

Analyzing the grammar

While building the chart and the parse trees, we make heavy use of the grammar, and the first thing we can try is to make the lookup in the grammar more efficient.

The Predict inference rule searches for productions with a specific leftmost category. To make this more efficient, we can transform the grammar into a finite map in which leftmost categories can be looked up to get the matching production lookup is then logarithmic in the size of the grammar instead of linear.

While building parse trees, we also look up categories in the grammar, but this time it is the main category of a production we are looking for, which means that we need another finite map to make this lookup more efficient.

Efficient lookup in the chart

While building a new Earley state, we need to look up edges in the previous states. Here it is possible to make two improvements.

First, we can turn the chart into a finite map indexed over integers, using an array, instead of a list. This enables us to find a certain state in the chart in constant time. Second, we can turn each state into a finite map, since we are looking for a certain category in the state.

Efficient lookup for parse trees

While building the parse trees, the *children* function searches for the parse trees of a certain edge in the list *edgeTrees* of edges and trees. This can be improved upon by transforming the list into a finite map, where we can look up edges more efficiently.

https://doi.org/10.1017/S0956796804005106 Published online by Cambridge University Press

4.2 Analysis

Space complexity

An edge in the kth Earley state ends in node k and can start in any node $j < k$. It can be derived from any of the productions in the grammar G, and up to δ categories can have been removed from the right-hand side of its production, where δ is the length of the longest production in G. This means that there will be $O(n|G|\delta)$ edges in the kth Earley state, since $k = O(n)$. Since there are n states altogether, we will have $O(n^2|G|\delta)$ edges in the final chart.

Time complexity

The *limit* function is quadratic in the size of the final fixed point set. Suppose that *limit more start* has m elements. According to the definition in section 1.4, the *more* function will be called exactly once for each added element. And since the result of the application is a subset of the final set, it will at most give m new elements. This means that the final set is calculated as m unions of sets with $O(m)$ elements, which gives us $O(m^2)$. This result is true only if the *more* function has complexity linear in m, which it has with the improvements suggested in section 4.1.

Thus, the time to build one Earley state is $O(n^2|G|^2\delta^2)$, since the size of a state is $O(n|G|\delta)$. And since we have n states in total, the worst-case complexity is $O(n^3|G|^2\delta^2)$. This is also the standard complexity for imperative implementations.

Building the parse trees

The space and time complexity for building the parse trees is, of course, exponential, since there can be an exponential number of trees.

Lazy evaluation

Lazy evaluation is used in two places; when building the final chart as a list of Earley states, where we refer back to the earlier states while building a new state; and when calculating the parse trees, where we refer to the parse trees of an edge, sometimes before the trees of that edge are calculated. Without laziness the code would be much clumsier and less declarative.

Also, laziness will only calculate the parse trees that are used in the final parse result. In the case of our example grammar, this means that the parse tree of the sentence (S) "flies like an arrow" will never be calculated, only the parse tree of the corresponding verb phrase (VP).

Acknowledgements

This paper is a revised version of Chapter 5 from Ljunglöf (2002). I am grateful to Aarne Ranta, Paul Callaghan, and three anonymous referees for many helpful comments and suggestions.

References

Earley, J. (1970) An efficient context-free parsing algorithm. *Comm. ACM*, **13**(2), 94–102.

Kasami, T. (1965) *An Efficient Recognition and Syntax Algorithm for Context-Free Languages*. Technical report AFCLR-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Kay, M. (1986) Algorithm schemata and data structures in syntactic processing. In: Grosz, B., Jones, K. and Webber, B. (editors), *Readings in Natural Language Processing*, pp. 35–70. Morgan Kaufman.

Kilbury, J. (1985) Chart parsing and the Earley algorithm. In: Klenk, U. (editor), *Kontextfreie Syntaxen und wervandte Systeme*. Niemeyer.

Ljunglöf, P. (2002) *Pure Functional Parsing*. Licenciate thesis, Göteborg University and Chalmers University of Technology, Gothenburg, Sweden.

Shieber, S., Schabes, Y. and Pereira, F. (1995) Principles and implementation of deductive parsing. *J. Logic Program.* **24**(1–2), 3–36.

Sikkel, K. (1997) *Parsing Schemata*. Springer-Verlag.

Wirén, M. (1992) *Studies in Incremental Natural-Language Analysis*. PhD thesis, Linköping University, Linköping, Sweden.

Younger, D. H. (1967) Recognition of context-free languages in time n^3. *Infor. & Control*, **10**(2), 189–208.