Parsing String Generating Hypergraph Grammars

Sebastian Seifert and Ingrid Fischer

Lehrstuhl für Informatik 2, Universität Erlangen–Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
Sebastian.T.Seifert@stud.informatik.uni-erlangen.de,
Ingrid.Fischer@informatik.uni-erlangen.de

Abstract. A string generating hypergraph grammar is a hyperedge replacement grammar where the resulting language consists of string graphs i.e. hypergraphs modeling strings. With the help of these grammars, string languages like $a^n b^n c^n$ can be modeled that can not be generated by context-free grammars for strings. They are well suited to model discontinuous constituents in natural languages, i.e. constituents that are interrupted by other constituents. For parsing context-free Chomsky grammars, the Earley parser is well known. In this paper, an Earley parser for string generating hypergraph grammars is presented, leading to a parser for natural languages that is able to handle discontinuities.

1 Discontinuous Constituents in German

One (of many) problems when parsing German are **discontinuous constituents** [1]. Discontinuous constituents are constituents which are separated by one or more other constituents and still belong together on a semantic or syntactic level. An example[1] for a discontinuous constituent is

(1) Er hat schnell gearbeitet.
He has fast worked.
He (has) worked fast.

The verb phrase *hat gearbeitet ((has) worked)*[2] is distributed; the finite verb part, the auxiliary verb *hat (has)*, is always in the second position in a German declarative sentence. The infinite verb part, the past participle *gearbeitet (worked)*, is usually in the last position of a declarative sentence, only a few exceptions like relative clauses or appositions can be put after the infinite verb part. Another (more complicated) German example of discontinuous constituents is

[1] The German examples are first translated word by word into English to explain the German sentence structure and then reordered into a correct English sentence.
[2] The present perfect in German can be translated either in present perfect or in past tense in English.

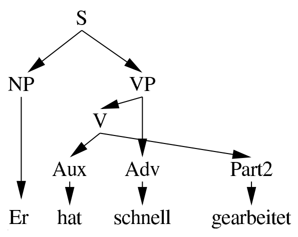


Fig. 1. The phrase structure tree for the German sentence *Er hat schnell gearbeitet.* (*He (has) worked fast.*)

(2) Kleine grüne Autos habe ich keine gesehen die mir gefallen hätten.
Small green cars have I none seen that me pleased would have.
I did not see any small green cars that would have pleased me.

Here, the noun phrase *keine kleinen grünen Autos, die mir gefallen hätten* (*any small green cars that would have pleased me*) is distributed in three parts[3]. There are also discontinuous conjunctions as in

(3) Weder Max noch Lisa haben die Aufgabe verstanden.
Neither Max nor Lisa have the task understood.
Neither Max nor Lisa understood the task.

In this example, the discontinuity of *weder ... noch* is also present in the English translation *neither ... nor.*

It is typical for all the above examples that the syntactical or semantical connection between the two parts of the discontinuous constituent cannot be expressed with general context–free Chomsky grammars. One would wish for a phrase structure tree as shown in Figure 1 for example (1). Of course it is possible to construct a weakly equivalent context–free Chomsky grammar to parse such a sentence, but it must contain some work-around for the discontinuous constituent like attaching one part of the discontinuous constituent in another production than the other. The main advantage of context-free string generating hypergraph grammars lies in their possibility to describe discontinuous constituents in a context–free formalism. A more detailed description can be found in [3]. It is desireable to have a parser based on this representation formalism for discontinuous constituents. With the help of a parser, larger grammar descriptions can be developed and tested. An often used parser for context–free Chomsky grammars is the Earley parser [4]. The main goal of this paper is to describe an Earley parser for context–free string generating hypergraph grammars. Therefore, first a short introduction into this type of grammars is given and their application in natural language modeling is shown. Related work is shortly described. Finally the modified Earley parser is described in detail.

[3] This syntactic phenomenon is also called *split topicalization* [2].

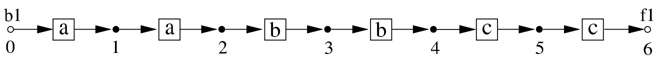


Fig. 2. The hypergraph representing $a^2b^2c^2$

2 String Generating Hypergraph Grammars

Hypergraph grammars have been studied extensively in the last decades. Introductions and applications can be found in [5], [6]. A subset of hypergraph grammars, context–free string generating hypergraph grammars, are described in detail in [7], [8], [5]. In this section, only an overview of the most important definitions is given.

A **labeled directed hyperedge** is a labeled edge that is not only connected to one source and one target node as an ordinary graph edge, but to

- a sequence of arbitrary length of source nodes $\langle s_1, s_2, \ldots, s_n \rangle$ and
- a sequence of arbitrary length of target nodes $\langle t_1, t_2, \ldots, t_m \rangle$ with $n, m \geq 0$.

The points where the hyperedge is connected to hypergraph nodes are called **tentacles**. The **type** (m, n) of a hyperedge consists of the number of source tentacles m and the number of target tentacles n.

A **hypergraph** (E, V, s, t, l, b, f) consists of

- a finite set of hyperedges E
- a finite set of nodes V
- a source function $s : E \longrightarrow V^*$ assigning a sequence of source nodes to each edge
- a target function $t : E \longrightarrow V^*$ assigning a sequence of target nodes to each edge
- a labeling function $l : E \longrightarrow A$ assigning a label from a given alphabet A to each edge; the label of an edge determines its type
- a sequence of external source nodes b
- a sequence of external target nodes f

A hypergraph has the **type** (m, n) if it has m external source nodes and n external target nodes. A hypergraph H with nodes $\{v_0, v_1, \ldots, v_n\}$ and edges $\{e_1, \ldots, e_n\}$ is called a **string graph**, if $s(e_i) = v_{i-1}$ and $t(e_i) = v_i$, $i = 1 \ldots n$. The node v_0 is the only external source node and v_n is the only external target node. This string hypergraph is a hypergraph based representation of a normal string. The letters (or words) of the string are the labels of the hyperedges. The letters labeling the hyperedges must be ordered in the string graph in the same way as in the underlying string. The hypergraph representation of the string $a^2b^2c^2$ is show in Figure 2.

A **hyperedge replacement rule** consists of a left hand side edge that is replaced by the hypergraph on the right hand side of the rule. Hyperedge and hypergraph must have the same type. In Figure 3 the hyperedge replacement rules generating the string language $a^nb^nc^n$ are shown.

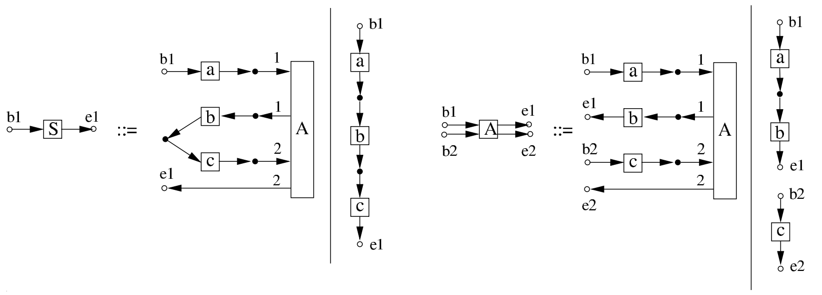


Fig. 3. Hyperedge replacement rules to generate the string language $a^n b^n c^n$.

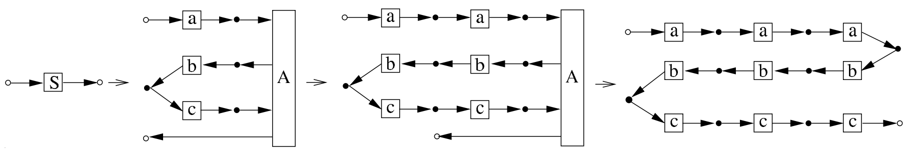


Fig. 4. The derivation of $a^3 b^3 c^3$ using the grammar given in Fig. 3.

When replacing a hyperedge with a graph as given in a production, the edge is first removed from its original host graph. In the resulting hole, the right hand side graph of the production is inserted except for its external nodes. The attaching nodes of the tentacles of the removed hyperedge are used instead: the ith source or target node of the removed edge replaces the ith external source or target node of the right hand side hypergraph. The derivation for $a^3 b^3 c^3$ is given in Figure 4 using the productions given in Figure 3. In Figure 5 the corresponding derivation tree is shown.

A **context–free hyperedge replacement grammar** $G = (T, N, P, S)$ consists of

– a finite set of terminal edge labels T
– a finite set of nonterminal edge labels N
– a finite set of productions P where each production has one hyperedge labeled with a nonterminal label on its left hand side
– a starting hyperedge labeled S.

The productions given in Figure 3 are part of a hypergraph grammar with terminal symbols $\{a, b, c\}$, nonterminal symbols $\{A, S\}$ and start symbol S.

A context–free hyperedge replacement grammar is a **string generating grammar** if the language generated by the grammar consists only of string graphs. This is the case in Figure 3. $a^n b^n c^n$ cannot be generated by a context–free Chomsky grammar.

Please note that for the rest of this paper we assume that the string generating hypergraph grammars are reduced, cycle–free and ϵ–free [9]. There are

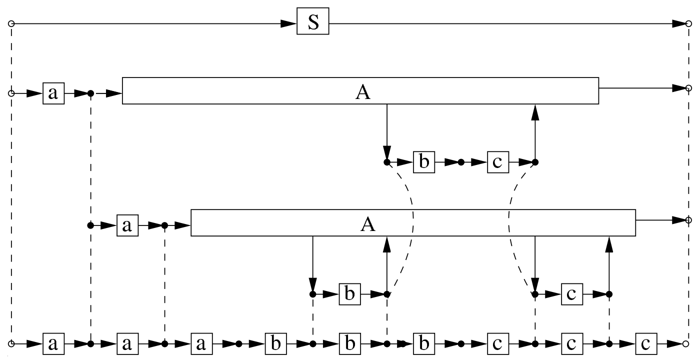


Fig. 5. The derivation tree for the derivation given in Fig. 4. The dotted lines mark the nodes must be matched.

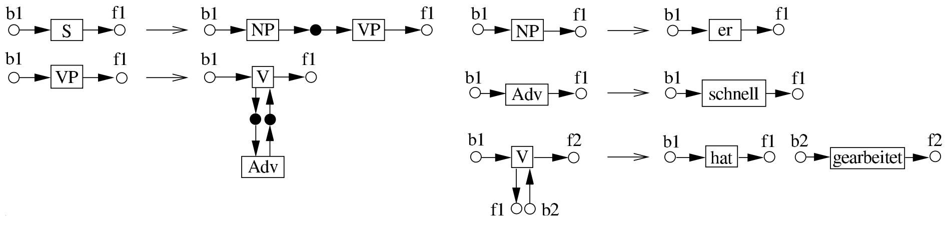


Fig. 6. hyperedge replacement productions for *Er hat schnell gearbeitet. (He (has) worked fast.)*

no unconnected nodes. If a hyperedge replacement grammar generates a string language, the start symbol must have one source and target tentacle. Each node is connected to at most one source tentacle and one target tentacle; otherwise no string language will be generated [10].

In the next section, a natural language example for context–free string generating hyperedge replacement grammars is provided.

3 A Natural Language Example

The productions shown in Figure 6 are necessary to generate example (1) *Er hat schnell gearbeitet. (He (has) worked fast.)*. Most of them resemble the usual grammar productions used in Chomsky grammars for natural languages. $S \rightarrow NP\ VP$ splits a sentence S into a noun phrase (*NP*) and a verbal phrase (*VP*). In our example, the noun phrase becomes the personal pronoun *er (he)*. The verbal phrase has two parts, an adverb *Adv* and the verb *V*. This is the most interesting rule, since the use of hypergraphs becomes evident. With their help, it is possible to split the verb *V* into two parts. The two parts are separated by the adverb. On the right hand side of the corresponding rule, the hyperedge labeled *V* modeling the verb has two source nodes and two target nodes. Figuratively spoken, one enters the verbal phrase through $b1$, leaves it for the adverb through $f1$, reenters for the second half of the verbal phrase through $b2$, and leaves again through $f2$. Finally, the verb has to be derived into its two halves, the auxiliary *hat* (*has*) and the past participle *gearbeitet* (*worked*). *Adv* is derived into *schnell* (*fast*).

4 Related Work

There are a lot of occurences of discontinuous constituents in natural languages like parenthical placement, right node raising, relative clause extraposition, scrampling and heavy NP shift. It is not possible to ignore these phenomena or to always use a work–around. A formalism is needed that can handle these problems. The most famous German treebank, *the Negra corpus* [11] and the largest English treebank, the Penstate treebank [12], contain notations for discontinuous constituents. Several proposals have been made on how to extend context–free grammars. Chomsky himself suggested transformations that move constituents around in addition to grammars [13]. Transformations should help to eliminate the need for discontinuous constituents. Other formalisms like *Lexical Functional Grammar* [14] have a context–free backbone extended with feature structures. Unification of feature structures after parsing ensures that discontinuous constituents that belong together are found. This is called *functional uncertainty.* Nevertheless, discontinuous constituents have to be split into seperate parts within the context-free backbone. An overview on discontinuous constituents in HPSG (*Head–Driven Phrase Structure Grammar*) using a similar mechanism can be found in [15]. Other approaches based on phrase structure grammars separate word order from the necessary constituents [16]. These ideas have been extended in [17] where the *discontinuous phrase structure grammar* is formally defined. As an extension for *Definite Clause Grammars*, *Static Discontinuity Grammars* are presented in [18]. In this approach, several rules can be applied in parallel. There may be gaps between the phrases generated by the rules. This way, several rules can handle one discontinuous constituent in parallel. Compared to all these approaches, string generating hypergraph grammars have one advantage: the discontinuous constituent is modelled with one symbol! The only implemented parser for general hypergraph grammars is described in [19]. This parser is similar to the Cocke–Kasami–Younger parser [16] for general context–free Chomsky grammars and embedded into *Diagen*, a diagram editor generator.

5 Earley–Parsing

The Earley parser for general context–free string grammars was first presented in [4] and is now widely used and has been extended in several ways [9] within natural language processing. It implements a top–down search, avoiding backtracking by storing all intermediate results only once.

When parsing with string grammars, positions at the beginning of the string, between the letters and at the end of the string to be parsed are numbered. When

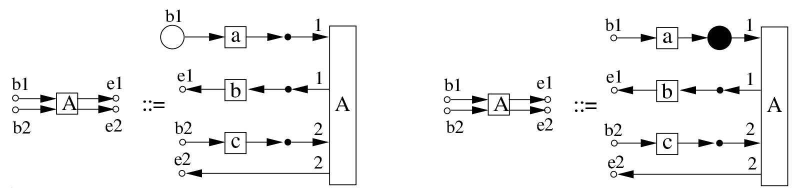


Fig. 7. Two examples for "dotted" rules using a grammar rule shown in Fig. 3.

parsing *aabbcc* seven positions are necessary, ranging from 0 to 6. From position 0 to position 1 the first a can be found, from 1 to 2 the second a, etc. The last letter c lies between 5 and 6. This numbering scheme is easily transferred onto string hypergraphs; the nodes in the hypergraph are numbered from 0 to 6 as done at the bottom of Fig. 2. These position numbers are necessary for the main data structure of the Earley algorithm, the **chart**. When parsing a string $s_0 s_1 \dots s_{n-1}$ consisting of n letters, the chart is a $(n+1) \times (n+1)$ table. In our running example we have a 7×7 table.

In this table, sets of **chart entries** are stored. Entries are never removed from the chart and are immutable after creation. A chart entry at position (i, j) in the chart contains information about the partial derivation trees for parsing the substring $s_i \dots s_{j-1}$. This information consists of the currently used grammar production and information about the progress made in completing the subtree given by this production. For string grammars, chart entries are visualized by so called **dotted rules**, where the dot marks the parsing progress. If the dot is at the end of the rule's right hand side, the chart entry is finished or **inactive**, else, it is **active**, i.e. ready to accept a terminal or an inactive chart entry. This concept can easily be transferred to hypergraphs. Two dotted hypergraph rules are shown in Fig. 7. The node larger than the others symbolizes the **dot**. It marks what parts of the right hand side of the rule have already been found. On the left side of Fig. 7, nothing has been found yet, as the dot is at a external source node of the graph. On the right side, a has already been found, the next symbol that has to be found is A. If the dot, also called the current node, is one of the external target nodes, the edge is **inactive**, otherwise it is **active**. Both rules shown in Fig. 7 are active.

The classical Earley algorithm consists of three steps that alternate until the possibilities to apply one of them are exhausted. These steps are called `shift`, `predict` and `complete`. The algorithm processes one terminal of the input string from left to right at a time by applying the `shift` operation. `shift` extends all active chart entries ending at the position of the new terminal symbol and expect this terminal symbol (it follows their dot). The extended chart entries are added to the chart. `complete` performs the analogous operation for inactive chart entries. Since an inactive chart entry represents a complete derivation subtree, active chart entries that expect the left hand side symbol of the inactive entry can be extended, advancing the dot by one position. `predict` is applied whenever an active entry is inserted into the chart. If the inserted entry expects a nonterminal symbol, `predict` inserts new chart entries starting at the active entry's ending position and carrying the expected symbol on their rule's left hand side. If these prediction entries become inactive during parsing, the prediction was correct and they will be used to `complete` the chart entry for which they were predicted. A successful parse has been found when a chart entry with the grammar's start symbol S at its rule's left hand side has been inserted into position $(0, n)$ (where n is the length of the input string).

Our variant of the Earley algorithm consists of modifications of these three main steps (procedure names differ slightly to avoid misunderstandings), and is thus very similar to the classical Earley algorithm. The main difference lies in the role of inactive chart entries. In the classical algorithm, an inactive chart entry is always **finished**, i.e. all right hand side symbols have been matched to a part of the input string. In our algorithm, a chart entry becomes inactive when the current node dot reaches an external target node of the right hand side hypergraph. While this inactive chart entry will already be used for the `complete` step, the chart entry is not necessarily finished, as there may be several external source and target nodes. `predict` will insert **continuation entries**, active entries that restart the parsing of an inactive entry through another external source node, if it encounters a nonterminal hyperedge over which the dot already has stepped before.

In our extension of the Earley algorithm, a **chart entry** e consists of the following information:

- **rule(e)**, the hypergraph rule that is used
- **currentnode(e)**, the dot inside the rule's right hand side
- **entrynode(e)**, one of the external source nodes of the rule
- **from(e)**, the index of the first symbol of the input string that is covered by this chart entry.
- **to(e)**, the index of the last covered symbol of the input string plus one
- **predecessor(e)**[4], the active chart entry extended to create e, or null
- **continuation–of(e)**[5], an inactive chart entry that has been continued by this edge or one of its predecessors, or null
- **parts(e,h)**, for any hyperedge h that is part of the rule's right hand side, either a chart entry describing a derivation of this edge, or null

The entry node is the external source node used to "enter" the hypergraph during parsing. In Fig. 7 the entry node is both times $b1$. Both *from* and *to* are integers between $0 \ldots n$ where n is the length of the input string. The substring ranging from s_{from} to $s_{\mathit{to}-1}$ has been matched to a path in the hypergraph between the entry node and the current node. E.g. the chart entry from 3 to 5 encompasses $s_3 s_4$ of the string to be parsed.

parts(e,h) is defined for hyperedges h that are part of the right hand side hypergraph of *rule(e)* and for which some derivation has been found during the

[4] Actually, only either *parts* or *predecessor* is necessary for the algorithm. We use *parts* for algorithmic purposes, but include *predecessor* for the visualization of the chart.

[5] 'of' refers to the function's value, not its parameter.

parsing process up to now. Its value is the terminal or the chart entry representing the derivation of this edge. It is set during the `complete` step.

We will now introduce an Earley–style algorithm for parsing strings generated by a hypergraph grammar with the properties mentioned at the end of section 2. While we explain the algorithm in detail, a running example will be provided in Figures 8 to 14. The example visualizes all chart entries created when parsing *aabbcc* with the grammar given in Fig. 3. The chart entries are numbered for convenience. For each chart entry e, first *from(e)* and *to(e)* is given, followed by *rule(e)* and *entrynode(e)*. *currentnode(e)* is given indirectly by drawing this node larger than the others. The current node is set to the entry node of the right hand side in the example. The brackets () following each terminal or nonterminal edge h represent *parts(e,h)*. These brackets are empty in Fig. 8 if no parts have been found yet. The last three columns are useful to visualize how a chart entry is created: *continuation-of* points to the inactive chart entry of which the current entry is a continuation entry. *predecessor* points to the active chart entry from which the current entry has been grown in the `complete` step. In the last column, the operation that created the chart entry is stated.

The main procedure, `parse`, returns all possible derivation trees for a given string of terminals and the start symbol S.[6]

```
procedure parse:
  returns: trees, a set of derivation trees
  parameters: S, a nonterminal label
              input, a string of terminal symbols
    trees := {}
    for all rules r where label(left-hand-side(r)) = S
       p := initial-prediction(r)
       if p is not in chart[0, 0]
         insert p into chart[0, 0]
         predict-for(p)
    for i = 0 .. length(input)-1
       shift(i, input[i])
    for all inactive entries e in chart[0, length(input)]
       if label(left-hand-side(rule(e))) = S
         insert generate-tree(e) into trees
    return trees
```

Since we are using a top–down parsing approach, it is necessary to recursively predict the leftmost parts of possible derivation trees, starting with the start symbol S, and using the procedure `predict-for` (detailed below).

`initial-prediction(r)` creates a chart entry from 0 to 0 using rule r. Its current node is set to the only possible entry node, since S is of the type $(1, 1)$. *parts* is null. In Fig. 8, two entries are inserted for the two S rules of the grammar. *predecessor* and *continuation-of* are null.

The main part of parsing happens in the `shift`–loop: one terminal symbol at a time is used to complete existing, active chart entries. Further predictions and

[6] The indentation of the pseudo-code marks the end of loops and alternatives.

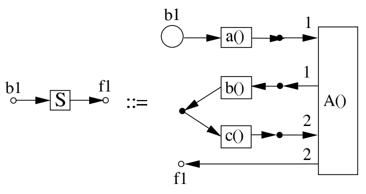
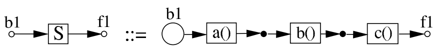
	from	to	rule with current node and parts	entry node	conti nuation of	prede cessor	
1.	0	0		1	–	–	initial–prediction(...)
2.	0	0		1	–	–	initial–prediction(...)

Fig. 8. Chart entries for strings ending at node 0 when parsing *aabbcc*.

completions are performed recursively, as detailed below in the procedure `shift`. Finally, each derivation tree we found during a successful parse is represented by a chart entry with the label S from 0 to n. The corresponding derivation tree is trivially built by recursion on the entry's *parts*.

Next, the function `predict-for` is explained:

```
procedure predict-for:
  returns: nothing
  parameters: e, an active chart entry
    h := hyperedge following currentnode(e)
    if label(h) is terminal label, abort
    if parts(e,h) is defined
       // we have reached a hyperedge that has already been traversed
       c := generate-continuation(parts(e,h),e)
       if c is not in chart[to(e), to(e)]
          insert c into chart[to(e), to(e)]
          predict-for(c)
    else
       for all rules r where label(left-hand-side(r)) = label(h)
        c := generate-prediction(e,r)
          if c is not in chart[to(e), to(e)]
            insert c into chart[to(e), to(e)]
             predict-for(c)
```

`predict-for` inserts **prediction entries** for an active chart entry e that expects a nonterminal symbol. What e expects to parse next is determined by looking at the hyperedge h following the dot. In the case of a terminal, no prediction is necessary, since the entry will be completed if the matching terminal is shifted.

If h has not been used for parsing before (*parts(e,h)* = *null*), we proceed analogous to the classical Earley algorithm:

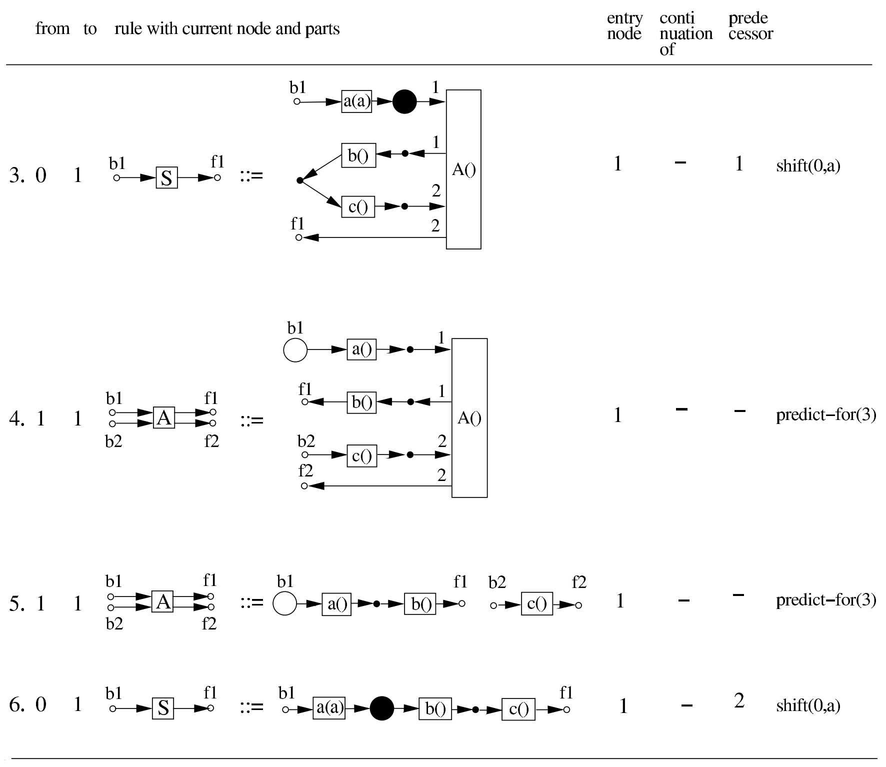


Fig. 9. Chart entries for strings ending at node 1 when parsing *aabbcc*.

The function `generate-prediction` returns a new active chart entry *c* that may become the root of a possible derivation tree of the hyperedge following *currentnode(e)*, using the rule *r*. No part of it has yet been matched to the input string. *c* starts and ends at *to(e)*, *rule(c) := r*, *currentnode(c)* is set to the begin node that corresponds to *currentnode(e)*, using the unique mapping between a hyperedge's nodes and its replacement hypergraph's nodes. *parts(c,x)* is undefined for all hyperedges *x*. *predecessor* and *continuation-of* are null. This entry is inserted into the chart.

In Fig. 9, two such prediction entries are shown. In chart entry 3, after having shifted over the first *a* in the input string, the hyperedge following the current node is labeled *A*. Two chart entries must be predicted for the two rules with left hand side *A*.

A phenomenon that is new compared to parsing with context-free Chomsky grammars (e.g. the classical Earley algorithm) is that multiple, separate substrings may form a derivation of the same nonterminal symbol. In order to cope with this, we introduce the concept of a **continuation chart entry**. When, during parsing, the current node reaches a hyperedge of the right hand side hypergraph to which a portion of the input string has already been matched, we

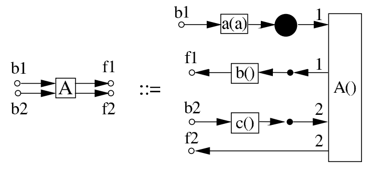
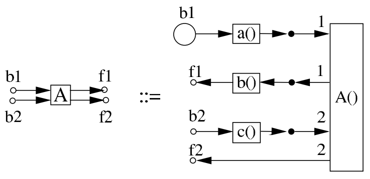
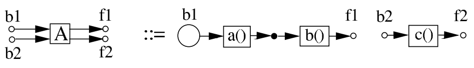
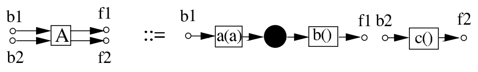
	from	to	rule with current node and parts	entry node	conti nuation of	prede cessor	
7.	1	2		1	–	4	shift(1,a)
8.	2	2		1	–	–	predict–for(7)
9.	2	2		1	–	–	predict–for(7)
10.	1	2		1	–	5	shift(1,a)

Fig. 10. Chart entries for strings ending at node 2 when parsing *aabbcc*.

predict an active chart entry that is consistent with the last chart entry that represented this nonterminal. `generate-continuation` returns such a continuation entry that represents an already partially matched possible derivation of h. This nonterminal hyperedge has already been "entered" once through a source node and has been "left" again through a target node. Now this hyperedge is "reentered" through another source node. The chart entry c returned by *generate-continuation(p,e)* starts and ends at *to(e)*, *rule(c) := rule(p)*, $\forall x$ *parts(c,x) := parts(p,x)*, *predecessor(c) := null*, c is a continuation of p. *currentnode(c)* is determined the same way as above.

In Fig. 13, the chart entry 15 is predicted from the chart entry 14. In chart entry 14 the hyperedge A is reentered through its second source node. $A(11)$ states that this hyperedge has been handled before in chart entry 11, whose continuation will now be predicted.

Next, `shift` and `complete-with` are explained:

```
procedure shift:
  returns: nothing
  parameters: position, a non-negative integer
              t, a terminal symbol
    for all active chart entries where to(e) = position
      h := hyperedge following currentnode(e)
```

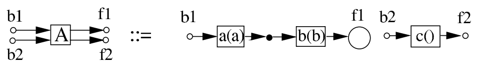
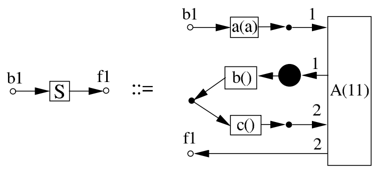
	from	to	rule with current node and parts	entry node	conti nuation of	prede cessor	
11.	1	3		1	–	10	shift(2, b)
12.	0	3		1	–	3	complete–with(11)

Fig. 11. Chart entries for strings ending at node 3 when parsing *aabbcc*.

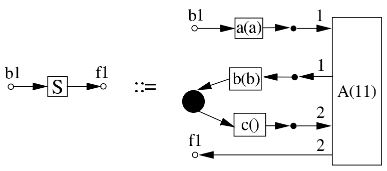
	from	to	rule with current node and parts	entry node	conti nuation of	prede cessor	
13.	0	4		1	–	12	shift(3,b)

Fig. 12. Chart entries for strings ending at node 4 when parsing *aabbcc*.

```
      if(label(h) = t)
         insert completion(e,t) into chart[from(e), to(e)+1]
         if completion(e,t) is inactive
            complete-with(completion(e,t))
         else
            predict-for(completion(e,t))


procedure complete-with:
  returns: nothing
  parameters: ia, an inactive chart entry
    for all active entries e where to(e) = from(ia)
       if expects(e,ia)
          insert completion(e,ia) into chart[from(ia), to(ia)]
          if completion(e,ia) is inactive
             complete-with(completion(e,ia))
          else
             predict-for(completion(e,ia))
```

`shift` and `complete-with` perform similar operations: they try to extend active chart entries with a new part. If this completion is possible (see below), the resulting chart entry represents a larger part of the input string and is inserted into the chart.

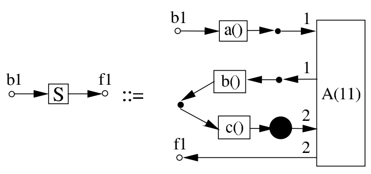
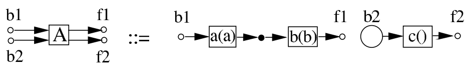
	from	to	rule with current node and parts	entry node	continuation of	predecessor	
14.	0	5		1	–	13	shift(4,c)
15.	5	5		2	11	–	predict–for(14)

Fig. 13. Chart entries for strings ending at node 5 when parsing *aabbcc*.

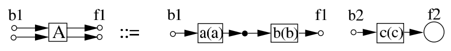
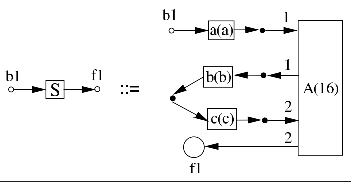
	from	to	rule with current node and parts	entry node	continuation of	predecessor	
16.	5	6		2	11	15	shift(5,c)
17.	0	6		1	–	14	complete–with(16

Fig. 14. Chart entries for strings ending at node 6 when parsing *aabbcc*.

If the completed entry is inactive, other chart entries can be completed with it. If it is active (expecting more input), we insert prediction entries.

The function `expects(e,ia)` is extended compared to the original Earley algorithm. `expects(e,ia)` determines if a given inactive edge *ia* will be accepted for completion of *e*. Please note that parsing of an inactive edge is not necessarily finished; an edge is inactive if the current node, the dot, has reached a target node of the rule. If the label or type of the left hand side of *ia*'s rule differs from *e*'s expected nonterminal edge label or type, *expects(e,ia)* is false. If the node used to enter *ia*, *entrynode(ia)*, does not correspond to *currentnode(e)*, *expects(e,ia)* is false. And if *ia* is a continuation chart entry, but the inactive entry that has been continued does not match *parts(e,h)*, *ia* represents a different derivation of the hyperedge than the one we assumed the last time it was traversed; therefore, *expects(e,ia)* is false. It is true otherwise.

The function `completion(e,x)` creates a new chart entry, either by accepting a terminal symbol *x*, or by accepting an inactive edge *x* into the partial derivation tree. Let *h* be the hyperedge following the current node inside *e*. The new chart entry c := *completion(e,x)* is identical to e, except for the following modifications: The substring of the input covered by c reaches from the start of the active entry e, *from(e)*, to the end of the inactive entry x, *to(x)*, or to *to(e)+1* for a terminal label x. *predecessor(c)* is set to e. *parts(c,h)* is set to x, i.e. c's derivation consists of the same subtrees as e's except for the new part x. If x is a continuation edge, *parts(e,h)* is already defined; the change of definition is intended, since all information held by *parts(e,h)* is also held by x. Furthermore, the current node pointer is advanced over h, using *currentnode(x)* to determine the correct target node of h, unless x is a terminal symbol. This is possible because of the unique mapping between the source and target nodes of h and the source and target nodes of the replacement hypergraph.

In Fig. 9, the `shift` over the first a of the string to be parsed is shown. The dot moves over the hyperedge labeled a, the entrynode is still 1 and the predecessor is the first chart entry. In Fig. 11, a `complete-with` is shown. In chart entry 12, the hyperedge labeled A is completed the first time. The inactive chart entry 11 is taken by the active chart entry 3.

The complexity of our algorithm is comparable to the underlying, classical Earley algorithm, $O(n^3)$. The usage of more complicated data structures introduces a constant penalty factor on space and time complexity in several places (or a factor of $O(k)$ where k is the maximum number of right hand side hyperedges in a grammar; but we regard k as a constant). Since the only point in our algorithm where it distinctively differs from classical Earley, aside from transferring its concepts onto string generating hypergraphs grammars, is the prediction of continuation entries, and only one such continuation entry is inserted during prediction (instead of several prediction entries), the complexity class of the algorithm itself does not increase [10].

6 Conclusion

String generating hypergraph grammars are a theoretical concept introduced in [5]. Context–free hyperedge replacement can model string languages that are not context–free in the usual Chomskian sense. In this paper an Earley–based parser was presented for string generating hypergraph grammars. The major extensions compared to the original Earley algorithm are the introduction of inactive chart entries that can be activated again. This is the case when a hypergraph was "entered" through one external source node, "left" through an external target node and "reentered" again through a new external source node. The parser described has been implemented in Java [10]. A German grammar and lexicon is currently developed.

This parser can be extended in may ways as shown in [9]. First, it is interesting to add an agenda and implement bottom–up parsing or parsing with probabilities. For linguistic applications, it is useful to attribute hyperedges and hypergraphs with feature structures that are combined with unification.

References

1. Trask, R.: A dictionary of Grammatical Terms in Linguistics. Roudledge, New York, London (1993)
2. Nolda, A.: Gespaltene Topikalisierung im Deutschen. In: Bericht des III. Ost-West-Kolloquiums für Sprachwissenschaft, Berlin, Germany (2000)
3. Fischer, I.: Modelling discontinuous constituents with hypergraph grammars. In . L. Pfaltz, M. Nagl, B.B., ed.: Applications of Graph Transformation with Industrial Relevance Proc. 2nd Intl. Workshop AGTIVE'03. Volume 3062 of Lecture Notes in Computer Science., Charlottesville, USA,, Springer Verlag (to appear)
4. Earley, J.: An efficient context–free parsing algorithm. Communications of the ACM **13** (1970) 94–102
5. Habel, A.: Hyperedge Replacement: Grammars and Languages. Volume 643 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1992)
6. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations. World Scientific (1997) 95–162
7. Engelfriet, J., Heyker, L.: The string generating power of context-free hypergraph grammars. Journal of Computer and System Sciences **43** (1991) 328–360
8. Engelfriet, J., Heyker, L.: Context–free hypergraph grammars have the same term-generating power as attribute grammars. Acta Informatica **29** (1992) 161–210
9. Jurafsky, D., Martin, J.H.: Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall (2000)
10. Seifert, S.: Ein Earley–Parser für Zeichenketten generierende Hypergraphgrammatiken. Studienarbeit, Lehrstuhl für Informatik 2, Universität Erlangen–Nürnberg (2004)
11. Brants, T., Skut, W.: Automation of treebank annotation. In Powers, D.M.W., ed.: Proceedings of the Joint Conference on New Methods in Language Processing and Computational Natural Language Learning: NeMLaP3/CoNLL98. Association for Computational Linguistics, Somerset, New Jersey (1998) 49–57
12. Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of english: The penn treebank. Computational Linguistics **19** (1994) 313–330
13. Chomsky, N.: Syntactic Structures. The Hague: Mouton (1957)
14. Kaplan, R.M., Maxwell, III, J.T.: An algorithm for functional uncertainty. COLING-88 (1988) 297–302
15. Müller, S.: Continuous or discontinuous constituents? a comparison between syntactic analyses for constituent order and their processing systems. Research on Language and Computation, Special Issue on Linguistic Theory and Grammar Implementation **2** (2004) 209–257
16. Naumann, S., Langer, H.: Parsing — Eine Einführung in die maschinelle Analyse natürlicher Sprache. Leifäden und Monographen der Informatik. B.G. Teubner, Stuttgart (1994)
17. Bunt, H.: Formal tools for the description and processing of discontinuous constituents. In: Discontinuous Constituency. Mouton De Gruyter (1996)
18. Dahl, V., Popowich, F.: Parsing and generation with static discontinuity grammars. New Generation Computers **8** (1990) 245–274
19. Minas, M.: Spezifikation und Generierung graphischer Diagrammeditoren. Shaker-Verlag, Aachen (2001)