Tree-structured chart parsing with left-corner and look-ahead constraints

Paul W. Placeway

February 28, 2000
CMU-LTI-00-161

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper investigates a method of improving the memory efficiency of a chart parser. We propose a technique to reduce the number of active arcs created in the process of parsing. We describe the algorithm in detail, including a description of how to integrate several well-known techniques that also improve parsing efficiency. We also provide empirical results that demonstrate the effectiveness of this technique.

Keywords: NLP, natural-language parsing, chart parsing, machine translation

1 Motivation

Chart parsers are very inviting from a computational point of view. There are several good reasons for this. The basic algorithm is easy to understand, even for non-experts, making a chart parser easy to correctly implement and debug. Also, because little pre-processing is required before beginning to parse, a chart parser is a useful tool for aiding the development of a grammar, allowing the developer a very rapid edit-test cycle [6][18].

Unfortunately, basic chart parsers are also somewhat inefficient, compared to Generalized LR type parsers [22][6]. The extent to which this is true has been a matter of some debate; unfortunately some comparisons have obscured the issue somewhat due to the inclusion of some feature (other than basic operation) in only one of the parsers under examination [13].

This work was motivated by a desire to preserve the advantages of the chart parser, while addressing several of the shortcomings. We found that by addressing the most obvious shortcoming of the basic chart algorithm, we could produce a parser that retains most of the clarity of implementation and transparency of operation of a standard chart parser while running quite efficiently.

We were also motivated by the good results that Carolyn Rosé and Alon Lavie have reported applying a chart parser with left-corner constraints to the problems of robust parsing [19]. The problems of ambiguity are exacerbated by robust parsing techniques, and LCFlex exhibits good behavior in such cases. Since we are dealing with a more traditional parsing problem, but also have problems with ambiguity, we were curious to see if a chart parser could be adapted to cope with the problems of large-scale machine-translation [14, 9]

2 Basic Algorithm

2.1 Tree-Structured Grammar

One basic shortcoming of a simple chart parser [11, 2, 25] is that it does not make efficient use of its grammar. Consider the following context-free rules:

$$\begin{array}{lllllll}
S & \leftarrow & NP & VP & & & (1)\\
NP & \leftarrow & NP & PP & & & (2)\\
NP & \leftarrow & Det & N & & & (3)\\
N & \leftarrow & N & Comma & Conj & N & (4)\\
N & \leftarrow & N & Comma & N & & (5)\\
NP & \leftarrow & N & & & & (6)\\
N & \leftarrow & N & Conj & N & & (7)\\
\ldots & & & & & &
\end{array}$$

One of the interesting features of this set of rules is that there is a lot of redundancy in the prefixes of the right-hand-sides. Unfortunately, a naïve implementation of a chart parser will not take advantage of this redundancy. Consider what would happen inside a “standard” chart parser using this grammar fragment. Suppose first input token was an N; then the parser would start active arcs for rules (4), (5), (6), and (7) above. Further suppose that the second input token was a Comma; then the parser would create active arcs to advance the active arcs from rules (4) and (5) that we created before.

In contrast, a shift-reduce parser [8, 22, 6, 25] will often use a grammar that has been optimized to eliminate this redundancy [8]. Since chart parsing and shift-reduce parsing are substantially similar [25], many techniques used in shift-reduce parsing can be applied to a chart parser.

Consider this grammar once again from the stand-point of a Push-Down Automaton (PDA) [8]. We will call reducing a sequence of symbols and producing a left-hand-side symbol a *reduction*, and we will call traversing a right-hand-side symbol a *shift*. (The obvious motivation for these particular terms is their use in the shift-reduce parser literature [8].)

With these terms in mind, let us re-examine this fragment, re-arranging the LHS to be on the right:

$$\begin{array}{lllllll}
NP & VP & & & \rightarrow & S & (1) \\
NP & PP & & & \rightarrow & NP & (2) \\
Det & N & & & \rightarrow & NP & (3) \\
N & Comma & Conj & N & \rightarrow & N & (4) \\
N & Comma & N & & \rightarrow & N & (5) \\
N & Conj & N & & \rightarrow & N & (7) \\
N & & & & \rightarrow & NP & (6) \\
\ldots & & & & & &
\end{array}$$

Finally, let's use a downward- (upward-) diagonal arrow $\searrow$ ($\nearrow$) to indicate that the RHS up to this point comes from the previous (next) rule:

$$\begin{array}{lllllllll}
NP & & VP & & & \ldots\ldots\ldots\ldots\ldots & \rightarrow & S & (1) \\
 & \searrow & & & & & & & \\
 & & PP & & & \ldots\ldots\ldots\ldots\ldots & \rightarrow & NP & (2) \\
Det & & N & & & \ldots\ldots\ldots\ldots\ldots & \rightarrow & NP & (3) \\
 & & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & & & \rightarrow & NP & (6) \\
 & \nearrow & & & & & & & \\
N & & Comma & & Conj & N & \rightarrow & N & (4) \\
 & & & \searrow & & & & & \\
 & & & & N & \ldots\ldots & \rightarrow & N & (5) \\
 & \searrow & & & & & & & \\
 & & Conj & & N & \ldots\ldots & \rightarrow & N & (7) \\
\ldots & & & & & & & &
\end{array}$$

Obviously fewer symbols appear in this grammar. Its structure is also quite indicative of how it will be used, and the optimization that inspired its creation. But first, we need to build the tree-structured grammar.

2.2 Building a Tree-Structured Grammar

Properly, this grammar is structured into a *trie* [7]. Each node in the trie contains a `symbol`, and two sets: the `shifts` and the `reductions`. The shifts are a set of (pointers to) trie nodes;

the reductions are a set of rules (possibly containing other useful information such as unification equations).

We build the grammar trie by "tree-ifying" the first rule in the grammar by producing a list of trie nodes (linked through the `shifts` field), with a final node containing an entry in the `reductions` field pointing to the original rule. [1]

```
tree-ify (rule, rhs)
    new-t = new node
    new-t.symbol = first(rhs)
    if length(rhs) == 1
        new-t.reductions = rule
    else
        new-t.shifts = tree-ify (rule, rest(rhs))
    return new-t

add-to-tree (rule, rhs, t-node)
    if lhs is empty
      push rule onto t-node.reductions
    else if (fist(rhs) == n.symbol) for some n in t-node.shifts
      recursively call add-to-tree (rule, rest(rhs), n)
    else push tree-ify (rule, rhs) onto t-node.shifts

build-tree (rule-list)
    tree = new node
    tree.shifts = tree-ify (first(rule-list), first(rule-list).rhs)
    foreach r in rest(rule-list)
        add-to-tree (r, r.rhs, tree)
    return tree
```

2.3 Using a Tree-Structured Grammar

Once the grammar tree has been built, applying it in the chart-parser is quite straightforward. The principle difference between this algorithm and one presented in e.g. Allen [2, p. 55] is that in the classic formulation of the chart algorithm, for each extendible active arc, *a* new active arc is added, and for each completable active arc, *a* new constituent is added to the agenda. In contrast, when using the tree-grammar *several* new active arcs and *several* new constituents may be created from a single extendible active arc.

The other notable difference is in the representation of the children of an active arc. We represent the children of an active arc with a *reversed-order* linked list in the `traceback` field of the arc. Because one active arc could spawn several arcs in turn *representing different rules*, the addition of a new child on a successfully extended arc must not disturb the children of any sister arcs. As noted by Carroll [6, p. 55], an efficient way to handle this is to simply build the child set as an up-tree [7], and then trace-back the list to build the final set of children should the arc successfully finish.

[1]The pseudo-code in this paper is in a C/C++/Java-like notation. An element of a structure is denoted by `structure.element`; lists are assumed to have a linked list structure, accessed by the operators first() and rest().

With this in mind, the basic tree-structured grammar chart parser algorithm shown in figure 1 is quite straight-forward.

```
basic_tree_parse (words):

    loop:
        if (agenda empty)
            if (no more words)
                break
            else
                Add next word

        e = remove next entry from agenda

        add e to chart.

        foreach r in rules-started-by (e.constituent) do:
            new-arc = make-arc (e.end, r, traceback = (item) )
            arcs-add (new-arc)

        foreach rule in single-rules-completed-by (e.constituent) do:
            new = make-entry (e.start, e.end, rule.LHS,
                              children = list(list(e)))
            agenda-add (new)

        foreach arc in arcs-continued-by (e.start, e.constituent) do:
            foreach node in arc.tnode.shiftlist
                new-arc = make-arc (e.end, node,
                                    traceback = (cons(e, arc.traceback))
                arc-add (new-arc)

            foreach rule in arc.tnode.reducelist
                let new-children = reverse (cons(e, arc.traceback))
                new = make-entry (first(new-children).start,
                                  e.end, rule.LHS,
                                  children = list (new-children))

                agenda-add (new)

        //end loop

    Do stuff with final chart here.

//end Basic_tree_parse
```

Figure 1: The Basic Tree-Structured Grammar Chart Parser

For an efficient implementation of this algorithm, there are several data structures that must be implemented in an efficient way.

- The agenda must be implemented in a way that supports quick addition and removal. This is because every entry in the chart must first be added and then later removed from the agenda.

- The root of the grammar tree contains only shift actions (assuming that there are no ϵ-reductions), and these are searched in the `rules-started-by` and `single-rules-completed-by` functions. The speed of the parser can be substantially improved by instead putting these initial grammar-tree entries in a table so that these two functions are unit-time operations.

  (We shall note here that searching down a list, such as the LISP `ASSOC` function, is *not* a unit-time operation, and therefore not to be considered 'quick'.)

- Active arcs should be stored in such a way that a new active arc can be added quickly.

- Active arcs should also be stored in such a way that the `arcs-continued-by` operation is quick – preferably a unit-time operation such as a table- or hash-lookup.

3 Complete Algorithm

There are a considerable number of features that can be added to the basic algorithm above to turn it into what could be considered a 'modern' parser. These features include unification, and various search restrictions (e.g. left-corner and look-ahead constraints).

3.1 Unification and Packing

This parser was developed to support "augmented context-free" unification grammar [23] using a Lexical Functional Grammar [10] inspired formalism. We will note briefly here that unification grammars pose some potentially *serious* computational complexity problems, raising the computational difficulty of parsing from solvable in polynomial-time to at least NP-complete, and sometimes all they way to Turing-complete (i.e. unsolvable) in the worst case[20, p. 66]. For a thorough discussion of this see Shieber [20] and Barton, et. al. [4].

Following Tomita [22] and Carroll [6], we also apply packing of locally ambiguous constituents. As Briscoe & Carroll [5] and Alshawi, et. al. [3] noted, packing can be used to save considerable amounts of duplicated work, but when working within a unification grammar framework both the constituent structures *and* the unification feature structures must be packed in order to see any actual gains.

In the algorithm presented below, packing is hidden within the `agenda-add` operation. Since an entry actually entered in the chart may have been used to complete other rules, and since its unification value will have been used in this case, we cannot simply pack into constituents already in the chart unless we then recalculate the unification values for any other constituent for which the chart entry in question is a descendant [12]. Implementing this adds a lot of extra book-keeping, so we simply chose to pack only into the agenda (since we know that agenda items have not been used and thus may be packed together safely) [13]. The details of handling packing in this way are actually quite interesting; Lavie and Rosé treat this in considerable detail in [13].

3.2 Left corner and look-ahead

We also apply left-corner [24, 17, 19, 8] and look-ahead constraints. Much has been written about these constraints; once examined as simple principles, the actual restrictions make much more sense. Our actual implementation of the calculation of the FIRST and FOLLOW relations follows the presentation in Aho, Sethi, & Ullman [1, pp. 188-189].

3.2.1 Look-ahead

The essence of look-ahead is this: don't create a constituent that cannot possibly be followed by the next word.

Only create a constituent of category c ending at e if c is a member of the set of categories that can occur to the immediate left of the category of the next word (at $e + 1$).

We implement this by pre-calculating a PRECEDE relation: if $b \in$ FOLLOW(a), then $a \in$ PRECEDE(b). Then, while processing the input word at e, we union together all the PRECEDE sets for each basic category of the next word ($e + 1$). Finally, we allow the creation of c only if c is a member of this union.

3.2.2 Left-corner

The essence of the left-corner restriction is this: don't create a constituent (or even start a rule that would create a constituent [16]) that you know you won't use (i.e. that no active arc is looking for).

In the naïve chart implementation, this is quite straight-forward. Whenever either a singleton (unit-length) rule is about to be evaluated, or a multiple-constituent rule is about to be started, at starting position s, check to see if the rule's LHS category c is a member of the union of the downward-left-corner sets of all current active arcs which start at s.

We implement this by pre-calculating the (downward) FIRST relation. We begin each parse by creating a vector of sets at least as long as the input sentence, and adding the elements of FIRST(Start) to this vector for the first word (where Start is the top-most symbol in the grammar). Then, whenever we add an active arc a, expecting a new category $a.c$ to begin at s, we union FIRST$(a.c)$ into the left-corner vector of sets at offset s.

3.2.3 Integrating Left-corner constraints into the Grammar Tree

There is one potential problem with structuring the grammar into a tree: several techniques, notably the left-corner constraint, involve computing set-membership or set-intersection questions about aspects of various rules.

In the naïve implementation of the grammar, this is straight-forward. Taking the example of the left-corner constraint, before "starting" a rule, one can ask whether the rule will be used if it can be completed. This question can be formalized as asking whether the LHS category of the rule c is a member of the (downward) left-corner set for the current position (at offset s).

The situation becomes more complicated when the grammar is structured into a tree. The notion of "the" category that can be derived from a rule becomes the *set* of categories that can be derived from some point in the tree.

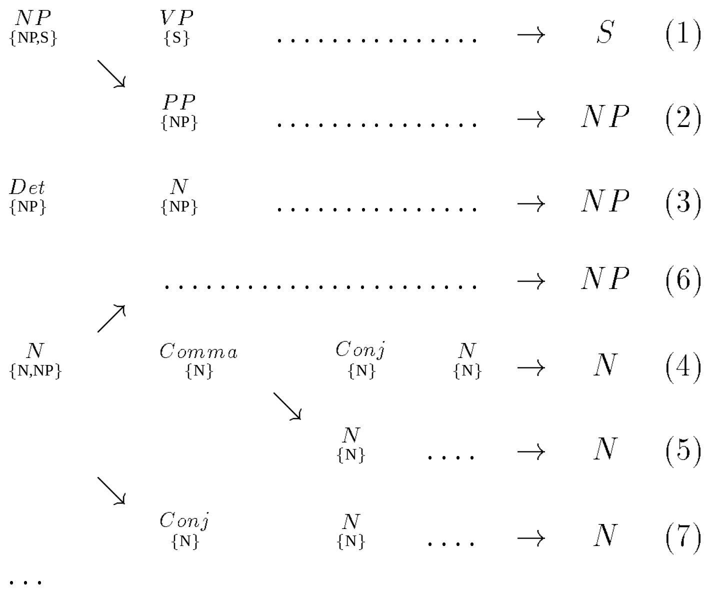
With this in mind, we can easily re-formulate the left-corner constraint to ask if there is a *non-empty intersection* between the set of all categories that could be completed from the point in the tree corresponding to the category of the current entry and the left-corner set for the current position.

The set of completable categories can be recursively defined as the union of:

- The categories of all rules in the reduce list of the current node, and
- The union of the sets of completable categories for each node on the shift list of the current node.

This definition invites the following extension: As one proceeds down a path in the tree, the set of categories that can be derived from the current node in the tree shrinks. With this in mind, one can decorate *each* tree node with the set of all nodes that are completable from that point in the tree, and require the left-corner non-empty intersection constraint to be true for every new active arc.

If we decorate the tree-structured grammar example we used above, the result is this:



For the actual parser, we compute these sets according to the above definition, decorating the tree nodes with the resulting sets.

3.2.4 Left-corner of Look-ahead

This combines together the look-ahead and left-corner restrictions described above. In essence: don't create an active arc which requires a constituent that cannot be started by the next word.

Only create active arc a that would next require category $a.c$ if $a.c$ could be started by the next word.

We implement this by pre-calculating a FIRST-PARENT relation, which is an upward version of the FIRST relation: a is a member of FIRST-PARENT(b) if b could be a left-most descendant of a. Then, while processing the input word at e, we union together the FIRST-PARENT sets for each basic category of the next word ($e + 1$). Finally, we allow the creation of an active arc a looking for category c to start at $e + 1$ if c is a member of the union of FIRST-PARENT sets for $e + 1$.

3.2.5 Implementation notes

We found that in our implementation, it is very important for these constraint-checking functions to be implemented efficiently. This is particularly true for the left-corner-of-look-ahead constraint. Since arc creation and handling is relatively cheap, this check must be less expensive still, or the over-all run-time will go up rather than down. Our initial implementation used an unsorted-list representation of sets, but this slowed down the parser considerably. We now represent these sets as packed bit-vectors, so that set-membership is a unit-time operation, and set-union, set-intersection, and querying for set-emptiness are all quick in practice.

When integrating left-corner constraints into the grammar tree, we noticed that the set of completable categories is quite sparse compared to the left-corner set. Rather than performing a simple set-intersection and set-non-empty operation, it turned out to be much faster to ask if the intersection between these two sets was non-empty by performing a sparse-set to dense-set intersection operation, short-circuiting when any part of the intersection was discovered. For the actual implementation, we found it best to simply decorate the tree with a list of completable categories, and to perform the restriction check by looking down this list for an element that was also in the left-corner set.

This particular implementation choice is probably quite language-dependent, however, and should be reconsidered for a truly optimal implementation.

3.2.6 Other details

The end-point restriction on starting rules is from Carroll [6, p. 55]. Carroll's actual restriction was on the minimum number of tokens needed to complete the new rule. Unfortunately, because the tree-grammar can encode several rules of different lengths together in one starting arc, we could only apply this restriction in this more limited form. (We actually tried annotating the tree with an indication of minimum length needed to complete any node, but this didn't save any appreciable run-time over and above this simple version.)

Figure 1 shows the main loop of the full tree-structured grammar chart parser, including unification and the restrictions discussed above. The actual implementation of the parser follows this structure exactly. We used object-oriented design to hide techniques such as packing and agenda ordering in the implementation of the agenda; table insertion and look-up details are in the implementations of the chart and (separate) active arcs. In doing so, we can clearly separate the core algorithm from these other (important) additional techniques.

4 Experimental Results

To demonstrate the effectiveness of the tree-parsing technique, we selected the sentences used for regression-testing of the KANT system [14, 9]. Catalyst is a controlled-language machine-translation system for translating heavy equipment manuals.

parser type	total number of nodes	total number of arcs	run-time (CPU seconds)
standard chart	1,284,541	2,632,428	2295
standard, LC-at-end	637,495	1,628,618	1399
standard, LA	1,152,558	2,429,881	2096
standard, LC-at-end & LA	603,024	1,744,601	1304
standard, LC-at-end, LA, & LC-of-LA	603,023	583,827	976
standard, LC-at-end, LC-in-rules, LA, & LC-of-LA	585,237	496,513	970
tree chart	1,284,832	2,186,731	2201
tree, LC-at-end	637,495	1,312,731	1329
tree, LA	1,152,558	2,053,074	2022
tree, LC-at-end & LA	603,024	1,451,624	1213
tree, LC-at-end, LA, & LC-of-LA	603,023	435,285	944
tree, LC-at-end, LC-in-tree LA, & LC-of-LA	585,237	364,227	931

Table 0.1: Parse times for Context-Free parse

4.1 Corpus and test conditions

This is a substantial, and heavy-weight system. The grammar contains 977 rules (958 unique context-free parts), using 544 distinct symbols (not counting unification equations). The grammar handles a substantial subset of American English, as well as special handling of SGML-based mark-up indicators.

The test set we used for these experiments is 1447 sentences taken from the corpus used in [15]. All of these sentences contain some sort of structural ambiguity, and were chosen primarily to investigate the effects of ambiguity on parsing. The sentences range in length from 5 to 40 words (26133 words total), with an average length of just over 18 words per sentence.

The tests were conducted on what is now considered *extremely* modest hardware: a Sun IPX workstation, with a 40 MHz processor, 64 MB of memory, running Lucid Common Lisp 4.0. This machine rates 21.8 SPECint92, or approx. 0.43 SPECint'95, or about 1/31st the speed of a Dell Dimension XPS Pro200n (333 MHz Pentium II Overdrive processor)[21].

All of the results below were collected using the same basic parser. For the "standard" entries, the parser was loaded with a grammar *without* applying the prefix-compression described above, whereas the "tree" entries do use the fully compressed tree-structured grammar. Both configurations were run with various constraints turned on or off. "LC-at-end" refers to left-corner restriction applied just prior to computing the unification function and (if successful) creating a chart entry; "LA" to look-ahead; and "LC-of-LA" refers to the left-corner-of-look-ahead as outlined above. "LC-in-rules" refers to applying the left-corner restriction prior to *starting* a rule, whereas "LC-in-tree" refers to applying the left-corner set-intersection-non-empty restriction in the grammar tree (as described above).

parser type	total number of nodes	total number of arcs	run-time (CPU seconds)
standard chart	877,842	2,002,446	2529
standard, LC-at-end	498,221	1,292,541	1652
standard, LA	790,650	1,870,433	2352
standard, LC-at-end & LA	483,787	1,420,674	1529
standard, LC-at-end, LA, & LC-of-LA	483,786	479,573	1276
standard, LC-at-end, LC-in-rules, LA, & LC-of-LA	468,082	405,100	1288
tree chart	877,998	1,643,099	2466
tree, LC-at-end	498,032	1,040,035	1594
tree, LA	790,650	1,556,824	2312
tree, LC-at-end & LA	483,709	1,175,861	1484
tree, LC-at-end, LA, & LC-of-LA	483,786	353,197	1249
tree, LC-at-end, LC-in-tree LA, & LC-of-LA	467,883	294,100	1257

Table 0.2: Parse times for parse with interleaved unification

4.2 Discussion of results

Table 0.1 lists some comparative results of running the parser on only the context-free spine of the test grammar. Comparing the numbers of arcs generated by the naïve chart versus the tree-structured chart that using the grammar tree does indeed save a substantial number of arc creations. Without any left-corner or look-ahead restrictions, when compared to the naïve implementation, the tree-structured grammar reduced the number of active arcs created by 20%. When employing full left-corner and look-ahead constraints [24, 19, 8] on the parser, the tree-grammar gave a 36% reduction in the number of active arcs.

Table 0.2 lists comparative numbers for running the parser with full unification. As in the context-free case, the tree-structured grammar reduces arc creation by 22% when using no restrictions, and 38% using full left-corner and look-ahead restrictions.

It is also interesting to note the huge decrease in the number of arcs created when applying the left-corner-of-look-ahead restriction in the creation of active arcs. This is clearly a win in terms of reducing memory use, and this savings of work is reflected in an improvement of between 28 and 34% in speed, in the context-free case, over applying the left-corner and look-ahead restrictions only to the creation of chart entries (i.e. inactive arcs), and a 19–20% improvement in speed while applying full interleaved unification.

On this corpus, left-corner constraints alone are clearly superior to look-ahead constraints alone. Further, the combination of left-corner and look-ahead constraints are only a modest win over left-corner alone until the left-corner-of-look-ahead restriction is applied as well.

As discussed above, the tree-structured grammar technique is independent of the choice of early vs. late left-corner restrictions. In spite of this, one can consider comparing the naïve grammar structure, with look-ahead constraints and performing left-corner tests before starting rules, with the tree-structured grammar with look-ahead constraints but applying the left-corner constraint to rules late – waiting to apply the left-corner until just before creating the chart entry (i.e. inactive arc) [16]. Even in this case, the tree-structured grammar is superior, producing 14% fewer arcs in the context-free-only test, and 15% fewer when applying full unification to the search.

Of course, creating arcs takes time. The decrease in number of arcs created is reflected in a 4% improvement in the run-time of the context-free case, and a 2.5% improvement when applying full unification. The arc processing in our implementation is quite light-weight, and constituent processing is quite heavy (particularly when unification is included), so we did not expect to enjoy a large gain in overall speed. Nevertheless, we do see a slight improvement in run-time.

Over all, using the tree-structured grammar substantially reduces the number of arcs created in in this chart parser, and cooperates well with the combined left-corner and look-ahead restrictions.

5 Acknowledgments

This work was supported by the CMU Language Technologies Institute.

The author wishes to thank Eric Nyberg, Alon Lavie, Carolyn Rosé, Bob Moore, and the anonymous reviewers for IWPT 2000, for comments that improved this paper. Krzysztof Czuba and Kathy Baker also provided many helpful comments.

This technical report is available on-line from:
`http://www.cs.cmu.edu/~pwp/papers/tree_parse_tr.ps`

```
Parse (words):
    Add words too look-ahead (words)
    loop:
        if (agenda empty)
            if (no more words)
                break
            else
                Add next word

        e = pop next entry off of agenda
        add e to chart.

        unless (e.end == final-position)
            foreach r in rules-started-by (e.constituent) do:
                if (lc-of-look-ahead-licenses (r.LHS)
                    && lc-node-set-licenses (r, e.start))
                    new-arc = make-arc (e.end, r, traceback = list(e) )
                    arcs-add (new-arc)

        foreach r in single-rules-completed-by (e.constituent) do:
            if (look-ahead-licenses (r.LHS)
                && LC-licenses (r.LHS, e.start))
                fs = rule.unify ( (e.fs) )
                if (fs)
                    new = make-entry (e.start, e.end, r.LHS, fs,
                                      children = list(list(e)))
                    agenda-add (new)

        foreach arc in arcs-continued-by (e.start, e.constituent) do:
            foreach node in arc.tnode.shiftlist
                if (lc-of-look-ahead-licenses (rule.LHS, e.end)
                    && lc-node-set-licenses (arc.tnode, arc.start))
                    new-arc = make-arc (e.end, node,
                                        traceback = cons(e, arc.traceback))
                    arc-add (new-arc)

            foreach rule in arc.tnode.reducelist
                let new-children = reverse (cons(e, arc.traceback))
                    child-fs = get-FSs-of (new-children)
                if (look-ahead-licenses (rule.LHS)
                    && LC-licenses (rule.LHS, first(new-children).start))
                    fs = rule.unify ( child-fs )
                    if (fs)
                        new = make-entry (first(new-children).start,
                                          e.end, rule.LHS, fs,
                                          children = list (new-children))
                        agenda-add (new)
    ;;end loop

    Do stuff with final chart here.
```

Figure 2: Tree-Structured Grammar Chart Parser with constraints and Unification

Bibliography

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, Reading, MA, 1985.

[2] ALLEN, J. *Natural Language Understanding*, second ed. Benjamin/Cummings, Redwood City, CA, 1995.

[3] ALSHAWI, H., Ed. *The Core Language Engine*. ACL-MIT Press Series in Natural Language Processing. MIT Press, Cambridge, MA, 1992.

[4] BARTON, G. E., BERWICK, R. C., AND RISTAD, E. S. *Computational Complexity and Natural Language*. Computational Models of Cognition and Perception. MIT Press, Cambridge, MA, 1987.

[5] BRISCOE, T., AND CARROLL, J. Generalized probabilistic lr parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics 19*, 1 (1993), 25–59.

[6] CARROLL, J. A. *Practical Unification-based Parsing of Natural Language*. PhD thesis, University of Cambridge, Computer Laboratory, Sept. 1993.

[7] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. McGraw-Hill and MIT Press, Cambridge, MA, 1990.

[8] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1979.

[9] KAMPRATH, ADOLPHSON, MITAMURA, AND NYBERG. Controlled Language for Multilingual Document Production: Experience with Caterpillar Technical English. In *Proc. Second Int. Workshop on Controlled Language Applications (CLAW '98)* (1998).

[10] KAPLAN, R. M., AND BRESNAM, J. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, MA, 1982, ch. 4, pp. 173–281.

[11] KAY, M. Algorithm schemata and data structures in syntactic processing. In *Readings in Natural Language Processing*. Morgan Kaufmann, San Mateo, CA, 1986 (1980).

[12] LAVIE, A., 1998. personal communication.

[13] LAVIE, A., AND ROSÉ, C. P. Optimal Ambiguity Packing in Context-Free Parsers with Interleaved Unification. In *Proc. 6th Intl. Wkshp on Parsing Technologies* (Feb. 2000).

[14] MITAMURA, T., NYBERG, E., AND CARBONELL, J. An efficient interlingua translation system for multi-lingual document production. In *Proc. 3rd Machine Translation Summit* (1991).

[15] MITAMURA, T., NYBERG, E., TORREJON, E., AND IGO, R. Multiple Strategies for Automatic Disambiguation in Technical Translation. In *Proc. TMI-99* (1999).

[16] MOORE, R., 2000. personal communication.

[17] NEDERHOF, M.-J., AND SARBO, J. J. Increasing the applicability of lr parsing. Tech. rep., Katholieke Universiteit Nijmegen, Dept. of Infomatics, Toernooiveld, 6525 ED Nijmegen, The NETHERLANDS, Mar. 1993. NIJT-93-06.

[18] PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 1997.

[19] ROSÉ, C. P., AND LAVIE, A. LCFLEX: An efficient robust left-corner parser, 1999.

[20] SHIEBER, S. M. *An Introduction to Unification-based Approaches to Grammar*. CSLI Lecture Notes. CSLI, Stanford, 1986.

[21] STANDARD PERFORMANCE EVALUATION CORPORATION. Third Quarter '98 SPEC CPU95 Results. `http://www.spec.org/osg/cpu95/results/res98q3/`.

[22] TOMITA, M. *Efficient Parsing for Natural Language*. Kluwer, Boston, 1986.

[23] TOMITA, M. An efficient augmented-context-free parsing algorithm. *Computational Linguistics 13*, 1–2 (Jan–Jun 1987), 31–46.

[24] VAN NOORD, G. An efficient implementation of the head-corner parser. *Computational Linguistics 23*, 3 (Mar. 1997), 425–456.

[25] VAN NOORD, G., NEDERHOF, M.-J., KOELING, R., AND BOUMA, G. Conventional Natural Language Processing in the NWO Priority Programme on Language and Speech Technology. Tech. rep., Rijksuniversiteit Groningen, Vakgroep Alfa-informatica & BCN, Mar. 1996.