LR RECURSIVE TRANSITION NETWORKS FOR EARLEY AND TOMITA PARSING

Mark Perlin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Internet: perlin@cs.cmu.edu

ABSTRACT*

Efficient syntactic and semantic parsing for ambiguous context-free languages are generally characterized as complex, specialized, highly formal algorithms. In fact, they are readily constructed from straightforward recursive transition networks (RTNs). In this paper, we introduce LR-RTNs, and then computationally motivate a uniform progression from basic LR parsing, to Earley's (chart) parsing, concluding with Tomita's parser. These apparently disparate algorithms are unified into a single implementation, which was used to automatically generate all the figures in this paper.

1. INTRODUCTION

Ambiguous context-free grammars (CFGs) are currently used in the syntactic and semantic processing of natural language. For efficient parsing, two major computational methods are used. The first is Earley's algorithm (Earley, 1970), which merges parse trees to reduce the computational dependence on input sentence length from exponential to cubic cost. Numerous variations on Earley's dynamic programming method have developed into a family of chart parsing (Winograd, 1983) algorithms. The second is Tomita's algorithm (Tomita, 1986), which generalizes Knuth's (Knuth, 1965) and DeRemer's (DeRemer, 1971) computer language LR parsing techniques. Tomita's algorithm augments the LR parsing "set of items" construction with Earley's ideas.

What is not currently appreciated is the continuity between these apparently distinct computational methods.

- Tomita has proposed (Tomita, 1985) constructing his algorithm from Earley's parser, instead of DeRemer's LR parser. In fact, as we shall show, Earley's algorithm may be viewed as one form of LR parsing.
- Incremental constructions of Tomita's algorithm (Heering, Klint, and Rekers, 1990) may similarly be viewed as just one point along a continuum of methods.

* This work was supported in part by grant R29 LM 04707 from the National Library of Medicine, and by the Pittsburgh NMR Institute.

The apparent distinctions between these related methods follows from the distinct complex formal and mathematical apparati (Lang, 1974; Lang, 1991) currently employed to construct these CF parsing algorithms.

To effect a uniform synthesis of these methods, in this paper we introduce LR Recursive Transition Networks (LR-RTNs) as a simpler framework on which to build CF parsing algorithms. While RTNs (Woods, 1970) have been widely used in Artificial Intelligence (AI) for natural language parsing, their representational advantages have not been fully exploited for efficiency. The LR-RTNs, however, are efficient, and shall be used to construct:

(1) a nondeterministic parser,
(2) a basic LR(0) parser,
(3) Earley's algorithm (and the chart parsers), and
(4) incremental and compiled versions of Tomita's algorithm.

Our uniform construction has advantages over the current highly formal, non-RTN-based, nonuniform approaches to CF parsing:

- Clarity of algorithm construction, permitting LR, Earley, and Tomita parsers to be understood as a family of related parsing algorithm.
- Computational motivation and justification for each algorithm in this family.
- Uniform extensibility of these syntactic methods to semantic parsing.
- Shared graphical representations, useful in building interactive programming environments for computational linguists.
- Parallelization of these parsing algorithms.
- All of the known advantages of RTNs, together with efficiencies of LR parsing.

All of these improvements will be discussed in the paper.

2. LR RECURSIVE TRANSITION NETWORKS

A *transition network* is a directed graph, used as a finite state machine (Hopcroft and Ullman, 1979). The network's nodes or edges are labelled; in this paper, we shall label the nodes. When an input sentence is read, state moves from node to node. A sentence is *accepted* if reading the entire sentence directs the network traversal so as to arrive at an

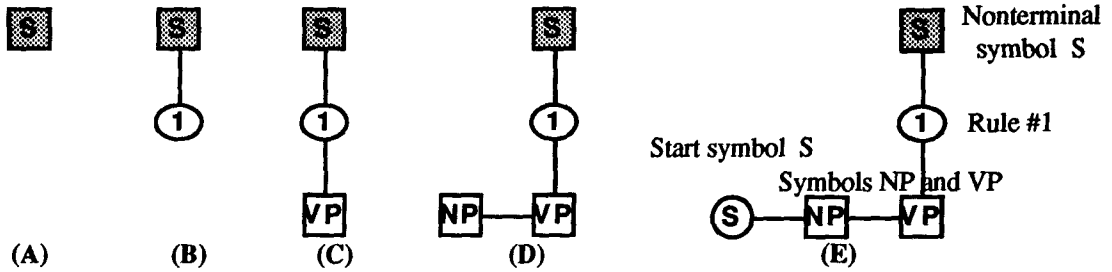

Figure 1. Expanding Rule#1: S→NP VP. **A.** Expanding the nonterminal symbol S. **B.** Expanding the rule node for Rule #1. **C.** Expanding the symbol node VP. **D.** Expanding the symbol node NP. **E.** Expanding the start node S.

accepting node. To increase computational power from regular languages to context-free languages, *recursive transition networks* (RTNs) are introduced.

An RTN is a forest of disconnected transition networks, each identified by a *nonterminal* label. All other labels are *terminal* labels. When, in traversing a transition network, a nonterminal label is encountered, control recursively passes to the beginning of the correspondingly labelled transition network. Should this labelled network be successfully traversed, on exit, control returns back to the labelled calling node.

The linear text of a context-free grammar can be cast into an RTN structure (Perlin, 1989). This is done by expanding each grammar rule into a linear chain. The top-down expansion amounts to a partial evaluation (Futamura, 1971) of the rule into a computational expectation: an eventual bottom-up data-directed instantiation that will complete the expansion.

Figure 1, for example, shows the expansion of the grammar rule #1 S→NP VP. First, the nonterminal S, which labels this connected component, is expanded as a nonterminal node. One method for realizing this nonterminal node, is via Rule#1; its rule node is therefore expanded. Rule#1 sets up the expectation for the VP symbol node, which in turn sets up the expectation for the NP symbol node. NP, the first symbol node in the chain, creates the start node S. In subsequent processing, posting an instance of this start symbol would indicate an expectation to instantiate the entire chain of Rule#1, thereby detecting a nonterminal symbol S. Partial instantiation of the Rule's chain indicates the partial progress in *sequencing* the Rule's right-hand-side symbols.

The expansion in Figure 1 constructs an **LR**-RTN. That is, it sets up a **L**eft-to-right parse of a **R**ightmost derivation. Such derivations are developed in the next Section. As used in AI natural language parsing, RTNs have more typically been LL-RTNs, for effecting parses of leftmost derivations (Woods, 1970), as shown in Figure 2A. (Other, more efficient, control structures have also been used (Kaplan, 1973).) Our shift from LL to LR, shown in Figure 2B, uses the chain expansion to set up a subsequent *data-driven* completion, thereby permitting greater parsing efficiency.

In Figure 3, we show the RTN expansion of the simple grammar used in our first set of examples:

```
S  → NP VP
NP → N | D N
VP → V NP .
```

Chains that share identical prefixes are merged (Perlin, 1989) into a directed acyclic graph (DAG) (Aho, Hopcroft, and Ullman, 1983). This makes our RTN a forest of DAGs, rather than trees. For example, the shared NP start node initiates the chains for Rules #2 and #3 in the NP component.

In augmented recursive transition networks (ATNs) (Woods, 1970), semantic constraints may be expressed. These constraints can employ case grammars, functional grammars, unification, and so on (Winograd, 1983). In our RTN formulation, semantic testing occurs when instantiating rule nodes: failing a constraint removes a parse from further

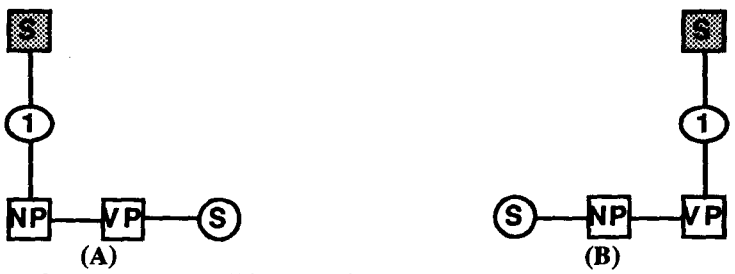

Figure 2. **A.** An LL-RTN for S→NP VP. This expansion does not set up an expectation for a data-driven leftward parse. **B.** The corresponding LR-RTN. The rightmost expansion sets up subsequent data-driven leftward parses.

processing. This approach applies to every parsing algorithm in this paper, and will not be discussed further.

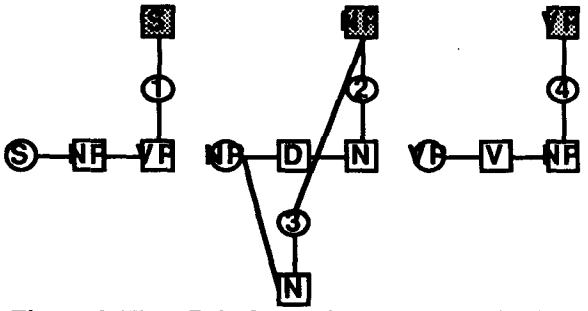

Figure 3. The RTN of an entire grammar. The three connected components correspond to the three nonterminals in the grammar. Each symbol node in the RTN denotes a *subsequence* originating from its leftmost start symbol.

3. NONDETERMINISTIC DERIVATIONS

A grammar's RTN can be used as a template for parsing. A sentence (the data) directs the instantiation of individual rule chains into a parse tree. The RTN *instances* exactly correspond to parse-tree nodes. This is most easily seen with *nondeterministic* rightmost derivations.

Given an input sentence of n words, we may derive a sentence in the language with the nondeterministic algorithm (Perlin, 1990):

```
Put an instance of nonterminal
  node S into the last column.
From right to left, for every
  column:
  From top to bottom, within the
    column:
    (1) Recursively expand the
      column top-down by
      nondeterministic selection of
      rule instances.
    (2) Install the next (leftward)
      symbol instance.
```

In substep (1), following selection, a rule node and its immediately downward symbol node are instantiated. The instantiation process creates a new object that inherits from the template RTN node, adding information about column position and local link connections.

For example, to derive "I Saw A Man" we would nondeterministically select and instantiate the correct rule choices #1, #4, #2, and #3, as in Figure 4. Following the algorithm, the derivation is (two dimensionally) top-down: top-to-bottom and right-to-left. To actually use this nondeterministic derivation algorithm to obtain all parses, one might enumerate and test all possible sequences of rules. This, however, has exponential cost in n, the input size. A more efficient approach is to reverse the top-down derivation, and recursively generate the parse(s) bottom-up from the input data.

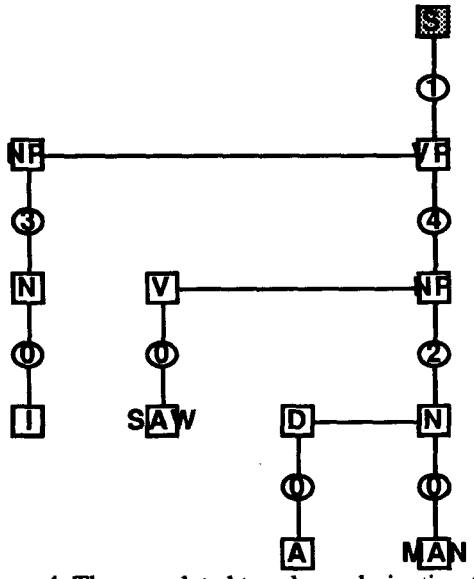

Figure 4. The completed top-down derivation (parse-tree) of "I Saw A Man". Each parse-tree symbol node denotes a *subsequence* of a recognized RTN chain. Rule #0 connects a word to its terminal symbol(s).

4. BASIC LR(0) PARSING

To construct a parser, we reverse the above top-down nondeterministic derivation technique into a bottom-up deterministic algorithm. We first build an inefficient LR-parser, illustrating the reversal. For efficiency, we then introduce the Follow-Set, and modify our parser accordingly.

4.1 AN INEFFICIENT BLR(0) PARSER

A simple, inefficient parsing algorithm for computing all possible parse-trees is:

```
Put an instance of start node S
  into the 0 column.
From left to right, for every
  column:
  From bottom to top, within the
    column:
    (1) Initialize the column with
      the input word.
    (2) Recursively complete the
      column bottom-up using the
      INSERT method.
```

This reverses the derivation algorithm into bottom-up generation: bottom-to-top, and left-to-right. In the inner loop, the Step (1) initialization is straightforward; we elaborate Step (2).

Step (2) uses the following method (Perlin, 1991) to insert instances of RTN nodes:

```
INSERT(instance)
{
 ASK instance
   (1) Link up with predecessor
     instances.
   (2) Install self.
   (3) ENQUEUE successor instances
     for insertion. }
```

In (1), links are constructed between the instance and its predecessor instances. In (2), the instance becomes available for cartesian product formation. In (3), the computationally nontrivial step, the instance enqueues any successor instances within its own column. Most of the INSERT action is done by instances of symbol and rule RTN nodes.

Using our INSERT method, a new *symbol instance* in the parse-tree links with predecessor instances, and installs itself. If the symbol's RTN node leads upwards to a rule node, one new rule instance successor is enqueued; otherwise, not.

Rule instances enqueue their successors in a more complicated way, and may require cartesian product formation. A rule instance must instantiate and enqueue all RTN symbol nodes from which they could possibly be derived. At most, this is the set

SAME-LABEL(rule) =
{ N ∈ RTN | N is a symbol node, and
the label of N is identical to the label of the
rule's nonterminal successor node }.

For every symbol node in SAME-LABEL(rule), instances may be enqueued. If X ∈ SAME-LABEL(rule) immediately follows a start node, i.e., it begins a chain, then a single instance of it is enqueued.

If Y ∈ SAME-LABEL(rule) does not immediately follow a start node, then more effort is required. Let X be the unique RTN node to the left of Y. Every instantiated node in the parse tree is the root of some subtree that spans an interval of the input sentence. Let the *left border* j be the position just to left of this interval, and k be the rightmost position, i.e., the *current column.*

Then, as shown in Figure 5, for every instance x of X currently in position j, an instance y (of Y) is a valid extension of subsequence x that has support from the input sentence data. The cartesian product

{ x | x an instance of X in column j }
× { rule instance}

forms the set of all valid predecessor pairs for new instances of Y. Each such new instance y of Y is enqueued, with some x and the rule instance as its two predecessors. Each y is a parse-tree node representing further progress in parsing a subsequence.

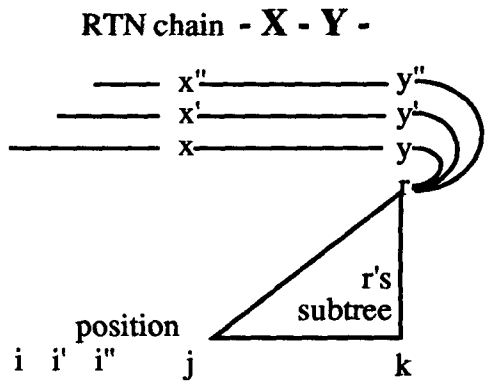

Figure 5. The symbol node Y has a left neighbor symbol node X in the RTN. The instance y of Y is the root of a parse-subtree that spans (j+1,k). Therefore, the rule instance r enqueues (at least) all instances of y, indexed by the predecessor product: {x in column j} × {r}.

4.2. USING THE FOLLOW-SET

Although a rule parse-node is restricted to enqueue successor instances of RTN nodes in SAME-LABEL(rule), it can be constrained further. Specifically, if the sentence data gives no evidence for a parse-subtree, the associated symbol node instance need never be generated. This restriction can be determined column-by-column as the parsing progresses.

We therefore extend our bottom-up parsing algorithm to:

```
Put an instance of start node S
  into the 0 column.
From left to right, for every
  column:
  From bottom to top, within the
    column:
    (1) Initialize the column with
      the input word.
    (2) Recursively complete the
      column bottom-up using the
      INSERT method.
    (3) Compute the column's
      (rightward) Follow-Set.
```

With the addition of Step (3), this defines our *Basic LR(0)*, or *BLR(0)*, parser. We now describe the Follow-Set.

Once an RTN node X has been instantiated in some column, it sets up an expectation for

- The RTN node(s) Y_g that immediately follow it;
- For each immediate follower Y_g, all those RTN symbol nodes $W_{g,h}$ that initiate chains that could recursively lead up to Y_g.

This is the *Follow-Set* (Aho, Sethi, and Ullman, 1986). The Follow-Set(X) is computed directly from the RTN by the recursion:

```
Follow-Set(X)
 LET Result ← ∅
 For every unvisited RTN node Y
   following X:
   Result ←  { Y } ∪
       IF Y's label is a terminal
         symbol,
         THEN ∅;
         ELSE Follow-Set of the
           start symbol of Y's label
 Return Result
```

As is clear from the recursive definition,

Follow-Set ($\cup_g \{X_g\}$) = $\cup_g$ Follow-Set (X_g).

Therefore, the Follow-Set of a column's symbol nodes can be deferred to Step (3) of the BLR(0) parsing algorithm, after the determination of all the nodes has completed. By only recursing on unvisited nodes, this traversal of the grammar RTN has time cost O(|G|) (Aho, Sethi, and Ullman, 1986), where $|G| \geq |RTN|$ is the size of the grammar (or its RTN graph). A Follow-Set computation is illustrated in Figure 6.

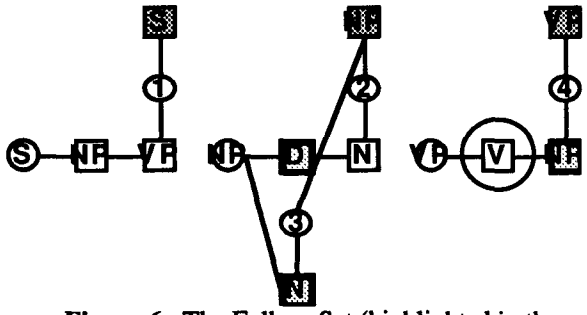

Figure 6. The Follow-Set (highlighted in the display) of RTN node V consists of the immediately following nonterminal node NP, and the two nodes immediately following the start NP node, D and N. Since D and N are terminal symbols, the traversal halts.

The set of symbol RTN nodes that a rule instance r spanning (j+1,k) can enqueue is therefore not

SAME-LABEL(rule),

but the possibly smaller set of RTN nodes

SAME-LABEL(rule) ∩ Follow-Set(j).

To enqueue r's successors in INSERT,

```
LET Nodes = SAME-LABEL(rule) ∩
     Follow-Set(j).
For every RTN node Y in Nodes,
  create and enqueue all instances
  y in Y:
  Let X be the leftward RTN symbol
    node neighbor of Y.
  Let PROD =
    {x | x an instance of X in
     column j} × {r}, if X exists;
    {r}, otherwise.
  Enqueue all members of PROD as
    instances of y.
```

The cartesian product PROD is nonempty, since an instantiated rule anticipates those elements of PROD mandated by Follow-Sets of preceding columns. The pruning of Nodes by the Follow-Set eliminates all bottom-up parsing that cannot lead to a parse-subtree at column k.

In the example in Figure 7, Rule instance r is in position 4, with j=3 and k=4. We have:

SAME-LABEL(r) = $\{N^2, N^3\}$,

i.e, the two symbol nodes labelled N in the sequences of Rules #2 and #3, shown in the LR-RTN of Figure 6.

Follow-Set(3) = Follow-Set($\{D^2\}$)

= $\{N^2\}$.

Therefore, SAME-LABEL(r)∩Follow-Set(3) = $\{N^2\}$.

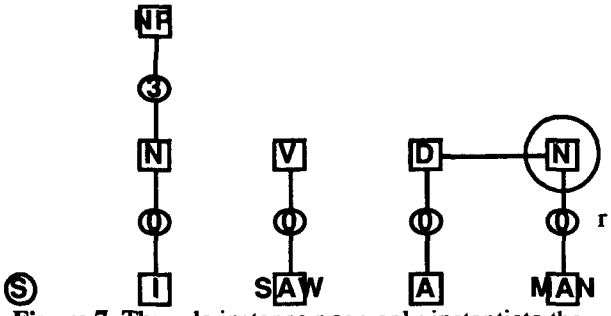

Figure 7. The rule instance r can only instantiate the single successor instance N^2. r uses the RTN to find the left RTN neighbor D of N^2. r then computes the cartesian product of instance d with r as {d}×{r}, generating the successor instance of N^2 shown.

5. EARLEY'S PARSING ALGORITHM

Natural languages such as English are ambiguous. A single sentence may have multiple syntactic structures. For example, extending our simple grammar with rules accounting for Prepositions and Prepositional-Phrases (Tomita, 1986)

```
S  → S PP
NP → NP PP
PP → P NP,
```

the sentence "I saw a man on the hill with a telescope through the window" has 14 valid derivations. In parsing, separate reconstructions of these different parses can lead to exponential cost.

For parsing efficiency, partially constructed instance-trees can be merged (Earley, 1970). As before, parse-node x denotes a point along a *parse-sequence*, say, v-w-x. The left-border i of this parse-sequence is the left-border of the leftmost parse-node in the sequence. All parse-sequences of RTN symbol node X that cover columns i+1 through k may be collected into a single equivalence class X(i,k). For

the purposes of (1) continuing with the parse and (2) disambiguating parse-trees, members of X(i,k) are indistinguishable. Over an input sentence of length n, there are therefore no more than $O(n^2)$ equivalence classes of X.

Suppose X precedes Y in the RTN. When an instance y of Y is added to position k, $k \leq n$, and the cartesian product is formed, there are only $O(k^2)$ possible equivalence classes of X for y to combine with. Summing over all n positions, there are no more than $O(n^3)$ possible product formations with Y in parsing an entire sentence.

Merging is effected by adding a MERGE step to INSERT:

```
INSERT(instance)
{
  instance ← MERGE(instance)
  ASK instance
    (1) Link up with predecessor
        instances.
    (2) Install self.
    (3) ENQUEUE successor instances
        for insertion. }
```

The parsing merge predicate considers two instantiated sequences equivalent when:

(1) Their RTN symbol nodes X are the same.

(2) They are in the same column k.

(3) They have identical left borders i.

The total number of links formed by INSERT during an entire parse, accounting for every grammar RTN node, is $O(n^3) \times O(|G|)$. The chart parsers are a family of algorithms that couple efficient parse-tree merging with various control organizations (Winograd, 1983).

6. TOMITA'S PARSING ALGORITHM

In our BLR(0) parsing algorithm, even with merging, the Follow-Set is computed at every column. While this computation is just $O(|G|)$, it can become a bottleneck with the very large grammars used in machine translation. By caching the requisite Follow-Set computations into a graph, subsequent Follow-Set computation is reduced. This incremental construction is similar to (Heering, Klint, and Rekers, 1990)'s, asymptotically constructing Tomita's all-paths LR parsing algorithm (Tomita, 1986).

The Follow-Set cache (or *LR-table*) can be dynamically constructed by Call-Graph Caching (Perlin, 1989) during the parsing. Every time a Follow-Set computation is required, it is looked up in the cache. When not present, the Follow-Set is computed and cached as a graph.

Following DeRemer (DeRemer, 1971), each cached Follow-Set node is finely partitioned, as needed, into disjoint subsets indexed by the RTN label name, as shown in the graphs of Figure 8. The partitioning reduces the cache size: instead of allowing all possible subsets of the RTN, the cache graph nodes contain smaller subsets of identically labelled symbol nodes.

When a Follow-Set node has the same subset of

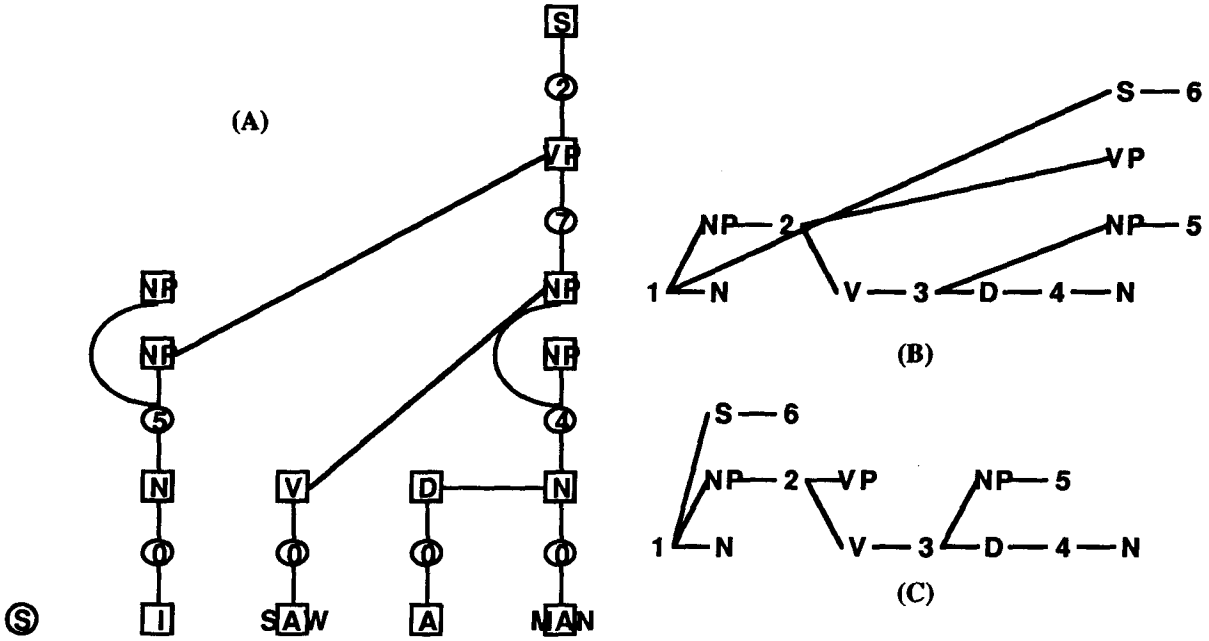

Figure 8. (A) A parse of "I Saw A Man" using the grammar in (Tomita, 1986). (B) The Follow-Set cache dynamically constructed during parsing. Each cache node represents a subset of RTN symbol nodes. The numbers indicate order of appearance; the lettered nodes partition their preceding node by symbol name. Since the cache was created on an as-needed basis, its shape parallels the shape of the parse-tree. (C) Compressing the shape of (B).

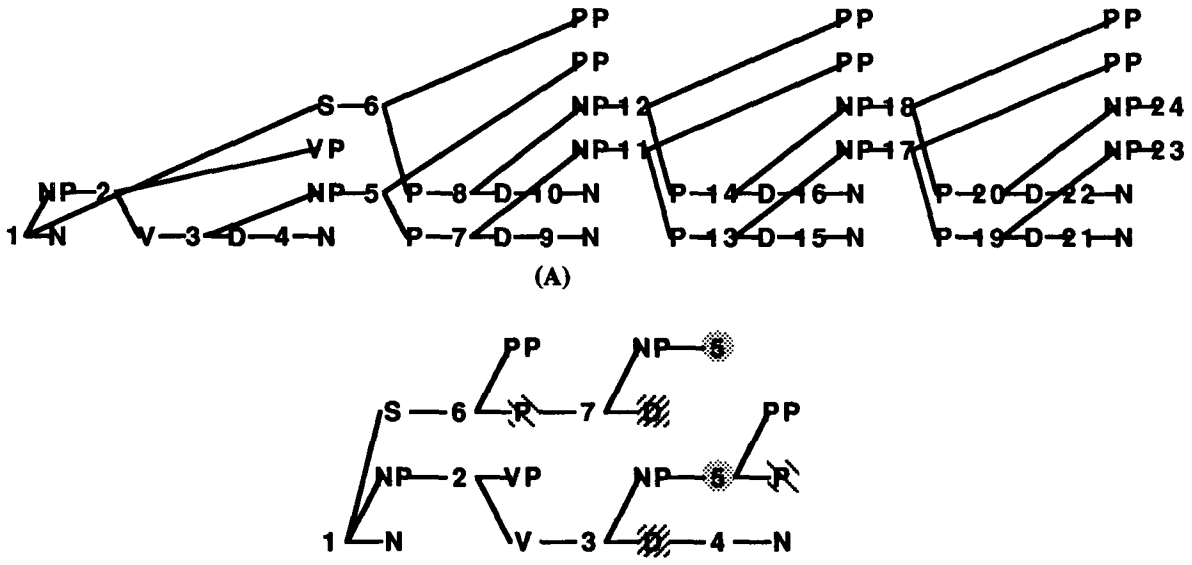

Figure 9. The LR table cache graph when parsing "I Saw A Man On The Hill With A Telescope Through The Window" **(A)** without cache node merging, and **(B)** with merging.

grammar symbol nodes as an already existing Follow-Set node, it is merged into the older node's equivalence class. This avoids redundant expansions, without which the cache would be an infinite tree of parse paths, rather than a graph. A comparison is shown in Figure 9. If the entire LR-table cache is needed, an ambiguous sentence containing all possible lexical categories at each position can be presented; convergence follows from the finiteness of the subset construction.

7. IMPLEMENTATION AND CURRENT WORK

We have developed an interactive graphical programming environment for constructing LR-parsers. It uses the color MAC/II computer in the Object LISP extension of Common LISP. The system is built on CACHE™ (Perlin, © 1990), a general Call-Graph Caching system for animating AI algorithms.

The RTNs are built from grammars. A variety of LR-RTN-based parsers, including BLR(0), with or without merging, and with or without Follow-Set caching have been constructed. Every algorithm described in this paper is implemented. Visualization is heavily exploited. For example, selecting an LR-table cache node will select all its members in the RTN display. The graphical animation component automatically drew all the RTNs and parse-trees in the Figures, and has generated color slides useful in teaching.

Fine-grained parallel implementations of BLR(0) on the Connection Machine are underway to reduce the costly cartesian product step to constant time. We are also adding semantic constraints.

8. CONCLUSION

We have introduced BLR(0), a simple bottom-up LR RTN-based CF parsing algorithm. We explicitly expand grammars to RTNs, and only then construct our parsing algorithm. This intermediate step eliminates the complex algebra usually associated with parsing, and renders more transparent the close relations between different parsers.

Earley's algorithm is seen to be fundamentally an LR parser. Earley's *propose* expansion step is a recursion analogous to our Follow-Set traversal of the RTN. By explicating the LR-RTN graph in the computation, no other complex data structures are required. The efficient merging is accomplished by using an option available to BLR(0): merging parse nodes into equivalence classes.

Tomita's algorithm uses the cached LR Follow-Set option, in addition to merging. Again, by using the RTN as a concrete data structure, the technical feats associated with Tomita's parser disappear. His shared packed forest follows immediately from our merge option. His graph stack and his parse forest are, for us, the same entity: the shared parse tree. Even the LR table is seen to derive from this parsing activity, particularly with incremental construction from the RTN.

Bringing the RTN into parsing as an explicit realization of the original grammar appears to be a conceptual and implementational improvement over less uniform treatments.

ACKNOWLEDGMENTS

Numerous conversations with Jaime Carbonell were helpful in developing these ideas. I thank the students at CMU and in the Tools for AI tutorial whose many questions helped clarify this approach.

REFERENCES

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. 1983. *Data Structures and Algorithms.* Reading, MA: Addison-Wesley.

Aho, A.V., Sethi, R., and Ullman, J.D. 1986. *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison-Wesley.

DeRemer, F. 1971. Simple LR(k) grammars. *Communications of the ACM*, 14(7): 453-460.

Earley, J. 1970. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2): 94-102.

Futamura, Y. 1971. Partial evaluation of computation process - an approach to a compiler-compiler. *Comp. Sys. Cont.*, 2(5): 45-50.

Heering, J., Klint, P., and Rekers, J. 1990. Incremental Generation of Parsers. *IEEE Trans. Software Engineering*, 16(12): 1344-1351.

Hopcroft, J.E., and Ullman, J.D. 1979. *Introduction to Automata Theory, Languages, and Computation.* Reading, Mass.: Addison-Wesley.

Kaplan, R.M. 1973. A General Syntactic Processor. In *Natural Language Processing*, Rustin, R., ed., 193-241. New York, NY: Algorithmics Press.

Knuth, D.E. 1965. On the Translation of Languages from Left to Right. *Information and Control*, 8(6): 607-639.

Lang, B. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Proc. Second Colloquium Automata, Languages and Programming*, 255-269. Loeckx, J., ed., (Lecture Notes in Computer Science, vol. 14), New York: Springer-Verlag.

Lang, B. 1991. Towards a Uniform Formal Framework for Parsing. In *Current Issues in Parsing Technology*, Tomita, M., ed., 153-172. Boston: Kluwer Academic Publishers.

Perlin, M.W. 1989. Call-Graph Caching: Transforming Programs into Networks. In *Proc. of the Eleventh Int. Joint Conf. on Artificial Intelligence*, 122-128. Detroit, Michigan, Morgan Kaufmann.

Perlin, M.W. 1990. Progress in Call-Graph Caching, Tech Report, CMU-CS-90-132, Carnegie-Mellon University.

Perlin, M.W. 1991. RETE and Chart Parsing from Bottom-Up Call-Graph Caching, submitted to conference, Carnegie Mellon University.

Perlin, M.W. © 1990. CACHE™: a Color Animated Call-grapH Environment, ver. 1.3, Common LISP MACINTOSH Program, Pittsburgh, PA.

Tomita, M. 1985. An Efficient Context-Free Parsing Algorithm for Natural Languages. In *Proceedings of the Ninth IJCAI*, 756-764. Los Angeles, CA, .

Tomita, M. 1986. *Efficient Parsing for Natural Language.* Kluwar Publishing.

Winograd, T. 1983. *Language as a Cognitive Process, Volume I: Syntax.* Reading, MA: Addison-Wesley.

Woods, W.A. 1970. Transition network grammars for natural language analysis. *Comm ACM*, 13(10): 591-606.