Semiring Parsing

Joshua Goodman*
Microsoft Research

We synthesize work on parsing algorithms, deductive parsing, and the theory of algebra applied to formal languages into a general system for describing parsers. Each parser performs abstract computations using the operations of a semiring. The system allows a single, simple representation to be used for describing parsers that compute recognition, derivation forests, Viterbi, n-best, inside values, and other values, simply by substituting the operations of different semirings. We also show how to use the same representation, interpreted differently, to compute outside values. The system can be used to describe a wide variety of parsers, including Earley's algorithm, tree adjoining grammar parsing, Graham Harrison Ruzzo parsing, and prefix value computation.

1. Introduction

For a given grammar and string, there are many interesting quantities we can compute. We can determine whether the string is generated by the grammar; we can enumerate all of the derivations of the string; if the grammar is probabilistic, we can compute the inside and outside probabilities of components of the string. Traditionally, a different parser description has been needed to compute each of these values. For some parsers, such as CKY parsers, all of these algorithms (except for the outside parser) strongly resemble each other. For other parsers, such as Earley parsers, the algorithms for computing each value are somewhat different, and a fair amount of work can be required to construct each one. We present a formalism for describing parsers such that a single simple description can be used to generate parsers that compute all of these quantities and others. This will be especially useful for finding parsers for outside values, and for parsers that can handle general grammars, like Earley-style parsers.

Although our description format is not limited to context-free grammars (CFGs), we will begin by considering parsers for this common formalism. The input string will be denoted $w_1 w_2 \ldots w_n$. We will refer to the complete string as the sentence. A CFG G is a 4-tuple $\langle N, \Sigma, R, S \rangle$ where N is the set of nonterminals including the start symbol S, Σ is the set of terminal symbols, and R is the set of rules, each of the form $A \rightarrow \alpha$ for $A \in N$ and $\alpha \in (N \cup \Sigma)^*$. We will use the symbol $\Rightarrow$ for immediate derivation and $\stackrel{*}{\Rightarrow}$ for its reflexive, transitive closure.

We will illustrate the similarity of parsers for computing different values using the CKY algorithm as an example. We can write this algorithm in its iterative form as shown in Figure 1. Here, we explicitly construct a Boolean chart, $chart[1..n, 1..|N|, 1..n+1]$. Element $chart[i, A, j]$ contains *TRUE* if and only if $A \stackrel{*}{\Rightarrow} w_i \ldots w_{j-1}$. The algorithm consists of a first set of loops to handle the singleton productions, a second set of loops to handle the binary productions, and a return of the start symbol's chart entry.

Next, we consider probabilistic grammars, in which we associate a probability with every rule, $P(A \rightarrow \alpha)$. These probabilities can be used to associate a probability

* One Microsoft Way, Redmond, WA 98052. E-mail: joshuago@microsoft.com

```
boolean chart[1..n, 1..|N|, 1..n+1] := FALSE;
for s := 1 to n /* start position */
    for each rule A → w_s ∈ R
        chart[s, A, s+1] := TRUE;
for l := 2 to n /* length, shortest to longest */
    for s := 1 to n−l+1 /* start position */
        for t := 1 to l−1 /* split length */
            for each rule A → BC ∈ R
                /* extra TRUE for expository purposes */
                chart[s, A, s+l] := chart[s, A, s+l] ∨
                    (chart[s, B, s+t] ∧ chart[s+t, C, s+l] ∧ TRUE);
return chart[1, S, n+1];
```

Figure 1
CKY recognition algorithm.

```
float chart[1..n, 1..|N|, 1..n+1] := 0;
for s := 1 to n /* start position */
    for each rule A → w_s ∈ R
        chart[s, A, s+1] := P(A → w_s);
for l := 2 to n /* length, shortest to longest */
    for s := 1 to n−l+1 /* start position */
        for t := 1 to l−1 /* split length */
            for each rule A → BC ∈ R
                chart[s, A, s+l] := chart[s, A, s+l] +
                    (chart[s, B, s+t] × chart[s+t, C, s+l] × P(A → BC));
return chart[1, S, n+1];
```

Figure 2
CKY inside algorithm.

with a particular derivation, equal to the product of the rule probabilities used in the derivation, or to associate a probability with a set of derivations, $A \stackrel{*}{\Rightarrow} w_i \ldots w_{j-1}$ equal to the sum of the probabilities of the individual derivations. We call this latter probability the inside probability of i, A, j. We can rewrite the CKY algorithm to compute the inside probabilities, as shown in Figure 2 (Baker 1979; Lari and Young 1990).

Notice how similar the inside algorithm is to the recognition algorithm: essentially, all that has been done is to substitute $+$ for $\vee$, $\times$ for $\wedge$, and $P(A \rightarrow w_s)$ and $P(A \rightarrow BC)$ for *TRUE*. For many parsing algorithms, this, or a similarly simple modification, is all that is needed to create a probabilistic version of the algorithm. On the other hand, a simple substitution is not always sufficient. To give a trivial example, if in the CKY recognition algorithm we had written

$$chart[s, A, s+l] := chart[s, A, s+l] \ \vee \ chart[s, B, s+t] \wedge chart[s+t, C, s+l];$$

instead of the less natural

$$chart[s, A, s+l] := chart[s, A, s+l] \ \vee \ chart[s, B, s+t] \wedge chart[s+t, C, s+l] \wedge TRUE;$$

larger changes would be necessary to create the inside algorithm.

Besides recognition, four other quantities are commonly computed by parsing algorithms: derivation forests, Viterbi scores, number of parses, and outside probabilities. The first quantity, a derivation forest, is a data structure that allows one to

efficiently compute the set of legal derivations of the input string. The derivation forest is typically found by modifying the recognition algorithm to keep track of "back pointers" for each cell of how it was produced. The second quantity often computed is the Viterbi score, the probability of the most probable derivation of the sentence. This can typically be computed by substituting $\times$ for $\wedge$ and max for $\vee$. Less commonly computed is the total number of parses of the sentence, which, like the inside values, can be computed using multiplication and addition; unlike for the inside values, the probabilities of the rules are not multiplied into the scores. There is one last commonly computed quantity, the outside probabilities, which we will describe later, in Section 4.

One of the key points of this paper is that all five of these commonly computed quantities can be described as elements of **complete semirings** (Kuich 1997). The relationship between grammars and semirings was discovered by Chomsky and Schützenberger (1963), and for parsing with the CKY algorithm, dates back to Teitelbaum (1973). A complete semiring is a set of values over which a multiplicative operator and a commutative additive operator have been defined, and for which infinite summations are defined. For parsing algorithms satisfying certain conditions, the multiplicative and additive operations of any complete semiring can be used in place of $\wedge$ and $\vee$, and correct values will be returned. We will give a simple normal form for describing parsers, then precisely define complete semirings, and the conditions for correctness.

We now describe our normal form for parsers, which is very similar to that used by Shieber, Schabes, and Pereira (1995) and by Sikkel (1993). This work can be thought of as a generalization from their work in the Boolean semiring to semirings in general. In most parsers, there is at least one chart of some form. In our normal form, we will use a corresponding, equivalent concept, **items**. Rather than, for instance, a chart element $chart[i, A, j]$, we will use an item $[i, A, j]$. Furthermore, rather than use explicit, procedural descriptions, such as

$$chart[s, A, s+l] := chart[s, A, s+l] \ \vee \ chart[s, B, s+t] \wedge chart[s+t, C, s+l] \wedge TRUE$$

we will use **inference rules** such as

$$\frac{R(A \rightarrow BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]}$$

The meaning of an inference rule is that if the top line is all true, then we can conclude the bottom line. For instance, this example inference rule can be read as saying that if $A \rightarrow BC$ and $B \stackrel{*}{\Rightarrow} w_i \ldots w_{k-1}$ and $C \stackrel{*}{\Rightarrow} w_k \ldots w_{j-1}$, then $A \stackrel{*}{\Rightarrow} w_1 \ldots w_{j-1}$.

The general form for an inference rule will be

$$\frac{A_1 \cdots A_k}{B}$$

where if the conditions $A_1 \ldots A_k$ are all true, then we infer that B is also true. The A_i can be either items, or (in an extension of the usual convention for inference rules) rules, such as $R(A \rightarrow BC)$. We write $R(A \rightarrow BC)$ rather than $A \rightarrow BC$ to indicate that we could be interested in a value associated with the rule, such as the probability of the rule if we were computing inside probabilities. If an A_i is in the form $R(\ldots)$, we call it a rule. All of the A_i must be rules or items; when we wish to refer to both rules and items, we use the word **terms**.

We now give an example of an item-based description, and its semantics. Figure 3 gives a description of a CKY-style parser. For this example, we will use the inside

Item form:
$[i, A, j]$

Goal:
$[1, S, n+1]$

Rules:

$$\frac{R(A \rightarrow w_i)}{[i, A, i+1]} \quad \text{Unary}$$

$$\frac{R(A \rightarrow BC) \quad [i, B, k] \quad [k, C, j]}{[i, A, j]} \quad \text{Binary}$$

Figure 3
Item-based description of a CKY parser.

semiring, whose additive operator is addition and whose multiplicative operator is multiplication. We use the input string xxx to the following grammar:

$$\begin{array}{lcll} S & \rightarrow & XX & 1.0 \\ X & \rightarrow & XX & 0.2 \\ X & \rightarrow & x & 0.8 \end{array} \tag{1}$$

Our first step is to use the unary rule,

$$\frac{R(A \rightarrow w_i)}{[i, A, i+1]}$$

The effect of the unary rule will exactly parallel the first set of loops in the CKY inside algorithm. We will instantiate the free variables of the unary rule in every possible way. For instance, we instantiate the free variable i with the value 1, and the free variable A with the nonterminal X. Since $w_1 = x$, the instantiated rule is then

$$\frac{R(X \rightarrow x)}{[1, X, 2]}$$

Because the value of the top line of the instantiated unary rule, $R(X \rightarrow x)$, has value 0.8, we deduce that the bottom line, $[1, X, 2]$, has value 0.8. We instantiate the rule in two other ways, and compute the following chart values:

$$\begin{array}{lcl} [1, X, 2] & = & 0.8 \\ [2, X, 3] & = & 0.8 \\ [3, X, 4] & = & 0.8 \end{array}$$

Next, we will use the binary rule,

$$\frac{R(A \rightarrow BC) \quad [i, B, k] \quad [k, C, j]}{[i, A, j]}$$

The effect of the binary rule will parallel the second set of loops for the CKY inside algorithm. Consider the instantiation $i = 1$, $k = 2$, $j = 3$, $A = X$, $B = X$, $C = X$,

$$\frac{R(X \rightarrow XX) \quad [1, X, 2] \quad [2, X, 3]}{[1, X, 3]}$$

We use the multiplicative operator of the semiring of interest to multiply together the values of the top line, deducing that $[1, X, 3] = 0.2 \times 0.8 \times 0.8 = 0.128$. Similarly,

$$\begin{aligned} [1, X, 3] &= 0.128 \\ [2, X, 4] &= 0.128 \\ [1, S, 3] &= 0.64 \\ [2, S, 4] &= 0.64 \end{aligned}$$

There are two more ways to instantiate the conditions of the binary rule:

$$\frac{R(S \rightarrow XX) \quad [1, X, 2] \quad [2, X, 4]}{[1, S, 4]}$$

$$\frac{R(S \rightarrow XX) \quad [1, X, 3] \quad [3, X, 4]}{[1, S, 4]}$$

The first has the value $1 \times 0.8 \times 0.128 = 0.1024$, and the second also has the value 0.1024. When there is more than one way to derive a value for an item, we use the additive operator of the semiring to sum them up. Thus, $[1, S, 4] = 0.2048$. Since $[1, S, 4]$ is the goal item for the CKY parser, we know that the inside value for *xxx* is 0.2048. The goal item exactly parallels the return statement of the CKY inside algorithm.

1.1 Earley Parsing

Many parsers are much more complicated than the CKY parser, and we will need to expand our notation a bit to describe them. Earley's algorithm (Earley 1970) exhibits most of the complexities we wish to discuss. Earley's algorithm is often described as a bottom-up parser with top-down filtering. In a probabilistic framework, the bottom-up sections compute probabilities, while the top-down filtering nonprobabilistically removes items that cannot be derived. To capture these differences, we expand our notation for deduction rules, to the following:

$$\frac{A_1 \cdots A_k}{B} \, C_1 \cdots C_j$$

$C_1 \cdots C_j$ are **side conditions**, interpreted nonprobabilistically, while $A_1 \cdots A_k$ are **main conditions** with values in whichever semiring we are using.[1] While the values of all main conditions are multiplied together to yield the value for the item under the line, the side conditions are interpreted in a Boolean manner: if all of them are nonzero, the rule can be used, but if any of them are zero, it cannot be. Other than for checking whether they are zero or nonzero, their values are ignored.

Figure 4 gives an item-based description of Earley's parser. We assume the addition of a distinguished nonterminal S' with a single rule $S' \rightarrow S$. An item of the form $[i, A \rightarrow \alpha \bullet \beta, j]$ asserts that $A \Rightarrow \alpha\beta \stackrel{*}{\Rightarrow} w_i \ldots w_{j-1}\beta$.

1 The side conditions may depend on any purely local information—the values of $A_1 \ldots A_k, B$, or $C_1 \ldots C_j$, as well as constant global functions, such as $R(X) \neq \sin(Y)$ (assuming here X and Y are variables in the A, B, C). The side conditions usually cannot depend on any contextual information, such as the grandfather of A_1, which would not be well defined, since there might be many derivations of A_1. Of course, one could encode the grandfather of A_1 as a variable in the item A_1, and then have a dependency on that variable. This would guarantee that the context was unique and well defined.

Item form:
$[i, A \rightarrow \alpha \bullet \beta, j]$

Goal:
$[1, S' \rightarrow S \bullet, n+1]$

Rules:

$$\frac{}{[1, S' \rightarrow \bullet S, 1]} \quad \text{Initialization}$$

$$\frac{[i, A \rightarrow \alpha \bullet w_j \beta, j]}{[i, A \rightarrow \alpha w_j \bullet \beta, j+1]} \quad \text{Scanning}$$

$$\frac{R(B \rightarrow \gamma)}{[j, B \rightarrow \bullet \gamma, j]} \; [i, A \rightarrow \alpha \bullet B\beta, j] \quad \text{Prediction}$$

$$\frac{[i, A \rightarrow \alpha \bullet B\beta, k] \quad [k, B \rightarrow \gamma \bullet, j]}{[i, A \rightarrow \alpha B \bullet \beta, j]} \quad \text{Completion}$$

Figure 4
Item-based description of Earley parser.

The prediction rule includes a side condition, making it a good example. The rule is:

$$\frac{R(B \rightarrow \gamma)}{[j, B \rightarrow \bullet \gamma, j]} \; [i, A \rightarrow \alpha \bullet B\beta, j]$$

Through the prediction rule, Earley's algorithm guarantees that an item of the form $[j, B \rightarrow \bullet \gamma, j]$ can only be produced if $S \stackrel{*}{\Rightarrow} w_1 \ldots w_{j-1} B\delta$ for some δ; this top-down filtering leads to significantly more efficient parsing for some grammars than the CKY algorithm. The prediction rule combines side and main conditions. The side condition, $[i, A \rightarrow \alpha \bullet B\beta, j]$, provides the top-down filtering, ensuring that only items that might be used later by the completion rule can be predicted, while the main condition, $R(B \rightarrow \gamma)$, provides the probability of the relevant rule. The side condition is interpreted in a Boolean fashion, while the main condition's actual probability is used.

Unlike the CKY algorithm, Earley's algorithm can handle grammars with epsilon (ϵ), unary, and n-ary branching rules. In some cases, this can significantly complicate parsing. For instance, given unary rules $A \rightarrow B$ and $B \rightarrow A$, a cycle exists. This kind of cycle may allow an infinite number of different derivations, requiring an infinite summation to compute the inside probabilities. The ability of item-based parsers to handle these infinite loops with relative ease is a major attraction.

1.2 Overview

This paper will simplify the development of new parsers in three important ways. First, it will simplify specification of parsers: the item-based description is simpler than a procedural description. Second, it will make it easier to generalize parsers across tasks: a single item-based description can be used to compute values for a variety of applications, simply by changing semirings. This will be especially advantageous for parsers that can handle loops resulting from rules like $A \rightarrow A$ and computations resulting from ϵ productions, both of which typically lead to infinite sums. In these cases, the procedure for computing an infinite sum differs from semi-

ring to semiring, and the fact that we can specify that a parser computes an infinite sum separately from its method of computing that sum will be very helpful. The third use of these techniques is for computing outside probabilities, values related to the inside probabilities that we will define later. Unlike the other quantities we wish to compute, outside probabilities cannot be computed by simply substituting a different semiring into either an iterative or item-based description. Instead, we will show how to compute the outside probabilities using a modified interpreter of the same item-based description used for computing the other values.

In the next section, we describe the basics of semiring parsing. In Section 3, we derive formulas for computing most of the values in semiring parsers, except outside values, and then in Section 4, show how to compute outside values as well. In Section 5, we give an algorithm for interpreting an item-based description, followed in Section 6 by examples of using semiring parsers to solve a variety of problems. Section 7 discusses previous work, and Section 8 concludes the paper.

2. Semiring Parsing

In this section we first describe the inputs to a semiring parser: a semiring, an item-based description, and a grammar. Next, we give the conditions under which a semiring parser gives correct results. At the end of this section we discuss three especially complicated and interesting semirings.

2.1 Semiring

In this subsection, we define and discuss semirings (see Kuich [1997] for an introduction). A semiring has two operations, $\oplus$ and $\otimes$, that intuitively have most (but not necessarily all) of the properties of the conventional $+$ and $\times$ operations on the positive integers. In particular, we require the following properties: $\oplus$ is associative and commutative; $\otimes$ is associative and distributes over $\oplus$. If $\otimes$ is commutative, we will say that the semiring is commutative. We assume an additive identity element, which we write as 0, and a multiplicative identity element, which we write as 1. Both addition and multiplication can be defined over finite sets of elements; if the set is empty, then the value is the respective identity element, 0 or 1. We also assume that $x \otimes 0 = 0 \otimes x = 0$ for all x. In other words, a semiring is just like a ring, except that the additive operator need not have an inverse. We will write $\langle \mathbb{A}, \oplus, \otimes, 0, 1 \rangle$ to indicate a semiring over the set $\mathbb{A}$ with additive operator $\oplus$, multiplicative operator $\otimes$, additive identity 0, and multiplicative identity 1.

For parsers with loops, i.e., those in which an item can be used to derive itself, we will also require that sums of an infinite number of elements be well defined. In particular, we will require that the semirings be **complete** (Kuich 1997, 611). This means that sums of an infinite number of elements should be associative and commutative, just like finite sums, and that multiplication should distribute over infinite sums, just as it does over finite ones. All of the semirings we will deal with in this paper are complete.[2]

All of the semirings we discuss here are also **ω-continuous**. Intuitively, this means that if any partial sum of an infinite sequence is less than or equal to some value,

2 Completeness is a somewhat stronger condition than we really need; we could, instead, require that limits be appropriately defined for those infinite sums that occur while parsing, but this weaker condition is more complicated to describe precisely.

boolean	$\langle \{TRUE, FALSE\}, \vee, \wedge, FALSE, TRUE \rangle$	recognition
inside	$\langle \mathbb{R}_0^\infty, +, \times, 0, 1 \rangle$	string probability
Viterbi	$\langle \mathbb{R}_0^1, \max, \times, 0, 1 \rangle$	prob. of best derivation
counting	$\langle \mathbb{N}_0^\infty, +, \times, 0, 1 \rangle$	number of derivations
derivation forest	$\langle 2^{\mathbb{E}}, \cup, \cdot, \emptyset, \{\langle\rangle\} \rangle$	set of derivations
Viterbi-derivation	$\langle \mathbb{R}_0^1 \times 2^{\mathbb{E}}, \max_{Vit}, \times_{Vit}, \langle 0, \emptyset \rangle, \langle 1, \{\langle\rangle\} \rangle \rangle$	best derivation
Viterbi-n-best	$\langle \{topn(X) \mid X \in 2^{\mathbb{R}_0^1 \times \mathbb{E}}\}, \max_{Vit\text{-}n}, \times_{Vit\text{-}n}, \emptyset, \{\langle 1, \{\langle\rangle\} \rangle\} \rangle$	best n derivations

Figure 5
Semirings used: $\langle A, \oplus, \otimes, 0, 1 \rangle$.

then the infinite sum is also less than or equal to that value.[3] This important property makes it easy to compute, or at least approximate, infinite sums.

There will be several especially useful semirings in this paper, which are defined in Figure 5. We will write $\mathbb{R}_a^b$ to indicate the set of real numbers from a to b inclusive, with similar notation for the natural numbers, $\mathbb{N}$. We will write $\mathbb{E}$ to indicate the set of all derivations in some canonical form, and $2^{\mathbb{E}}$ to indicate the set of all sets of derivations in canonical form. There are three derivation semirings: the derivation forest semiring, the Viterbi-derivation semiring, and the Viterbi-n-best semiring. The operators used in the derivation semirings ($\cdot, \max_{Vit}, \times_{Vit}, \max_{Vit\text{-}n}$, and $\times_{Vit\text{-}n}$) will be described later, in Section 2.5.

The inside semiring includes all nonnegative real numbers, to be closed under addition, and includes infinity to be closed under infinite sums, while the Viterbi semiring contains only numbers up to 1, since under max this still leads to closure.

The three derivation forest semirings can be used to find especially important values: the derivation forest semiring computes all derivations of a sentence; the Viterbi-derivation semiring computes the most probable derivation; and the Viterbi-n-best semiring computes the n most probable derivations. A derivation is simply a list of rules from the grammar. From a derivation, a parse tree can be derived, so the derivation forest semiring is analogous to conventional parse forests. Unlike the other semirings, all three of these semirings are noncommutative. The additive operation of these semirings is essentially union or maximum, while the multiplicative operation is essentially concatenation. These semirings are described in more detail in Section 2.5.

2.2 Item-based Description

A semiring parser requires an item-based description of the parsing algorithm, in the form given earlier. So far, we have skipped one important detail of semiring parsing. In a simple recognition system, as used in deduction systems, all that matters is whether an item can be deduced or not. Thus, in these simple systems, the order of processing items is relatively unimportant, as long as some simple constraints are met. On the other hand, for a semiring such as the inside semiring, there are important ordering constraints: we cannot compute the inside value of an item until the inside values of

3 To be more precise, all semirings we discuss here are **naturally ordered**, meaning that we can define a partial ordering, $\sqsubseteq$, such that $x \sqsubseteq y$ if and only if there exists z such that $x \oplus z = y$. We call a naturally ordered complete semiring ω-continuous (Kuich 1997, 612) if for any sequence $x_1, x_2, \ldots$ and for any constant y, if for all n, $\bigoplus_{0 \leq i \leq n} x_i \sqsubseteq y$, then $\bigoplus_i x_i \sqsubseteq y$.

all of its children have been computed.

Thus, we need to impose an ordering on the items, in such a way that no item precedes any item on which it depends. We will assign each item x to a "bucket" B, writing $bucket(x) = B$ and saying that item x is **associated** with B. We order the buckets in such a way that if item y depends on item x, then $bucket(x) \leq bucket(y)$. For some pairs of items, it may be that both depend, directly or indirectly, on each other; we associate these items with special "looping" buckets, whose values may require infinite sums to compute. We will also call a bucket looping if an item associated with it depends on itself.

One way to achieve a bucketing with the required ordering constraints (suggested by Fernando Pereira) is to create a graph of the dependencies, with a node for each item, and an edge from each item x to each item b that depends on it. We then separate the graph into its strongly connected components (maximal sets of nodes all reachable from each other), and perform a topological sort. Items forming singleton strongly connected components are associated with their own buckets; items forming nonsingleton strongly connected components are associated with the same looping bucket. See also Section 5.

Later, when we discuss algorithms for interpreting an item-based description, we will need another concept. Of all the items associated with a bucket B, we will be able to find derivations for only a subset. If we can derive an item x associated with bucket B, we write $x \in B$, and say that item x is **in** bucket B. For example, the goal item of a parser will almost always be *associated* with the last bucket; if the sentence is grammatical, the goal item will be *in* the last bucket, and if it is not grammatical, it will not be.

It will be useful to assume that there is a single, variable-free goal item, and that this goal item does not occur as a condition for any rules. We could always add a new goal item [*goal*] and a rule $\frac{[old\text{-}goal]}{[goal]}$ where [*old-goal*] is the goal in the original description.

2.3 The Grammar

A semiring parser also requires a grammar as input. We will need a list of rules in the grammar, and a function, $R(rule)$, that gives the value for each rule in the grammar. This latter function will be semiring-specific. For instance, for computing the inside and Viterbi probabilities, the value of a grammar rule is just the conditional probability of that rule, or 0 if it is not in the grammar. For the Boolean semiring, the value is *TRUE* if the rule is in the grammar, *FALSE* otherwise. $R(rule)$ replaces the set of rules R of a conventional grammar description; a rule is in the grammar if $R(rule) \neq 0$.

2.4 Conditions for Correct Processing

We will say that a semiring parser works correctly if, for any grammar, input, and semiring, the value of the input according to the grammar equals the value of the input using the parser. In this subsection, we will define the value of an input according to the grammar, define the value of an input using the parser, and give a sufficient condition for a semiring parser to work correctly. From this point onwards, unless we specifically mention otherwise, we will assume that some fixed semiring, item-based description, and grammar have been given, without specifically mentioning which ones.

2.4.1 Value According to Grammar. Consider a derivation E, consisting of grammar rules $e_1, e_2, \ldots, e_m$. We define the value of the derivation according to the grammar to

be simply the product (in the semiring) of the values of the rules used in E:

$$V_G(E) = \bigotimes_{i=1}^{m} R(e_i)$$

Then we can define the value of a sentence that can be derived using grammar derivations $E^1, E^2, \ldots, E^k$ to be:

$$V_G = \bigoplus_{j=1}^{k} V_G(E^j)$$

where k is potentially infinite. In other words, the value of the sentence according to the grammar is the sum of the values of all derivations. We will assume that in each grammar formalism there is some way to define derivations uniquely; for instance, in CFGs, one way would be using left-most derivations. For simplicity, we will simply refer to derivations, rather than, for example, left-most derivations, since we are never interested in nonunique derivations.

A short example will help clarify. We consider the following grammar:

$$\begin{array}{lcll} S & \rightarrow & AA & R(S \rightarrow AA) \\ A & \rightarrow & AA & R(A \rightarrow AA) \\ A & \rightarrow & a & R(A \rightarrow a) \end{array} \tag{2}$$

and the input string *aaa*. There are two grammar derivations, the first of which is $S \overset{S \rightarrow AA}{\Rightarrow} AA \overset{A \rightarrow AA}{\Rightarrow} AAA \overset{A \rightarrow a}{\Rightarrow} aAA \overset{A \rightarrow a}{\Rightarrow} aaA \overset{A \rightarrow a}{\Rightarrow} aaa$, which has value $R(S \rightarrow AA) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)$. Notice that the rules in the value are the same rules in the same order as in the derivation. The other grammar derivation is $S \overset{S \rightarrow AA}{\Rightarrow} AA \overset{A \rightarrow a}{\Rightarrow} aA \overset{A \rightarrow AA}{\Rightarrow} aAA \overset{A \rightarrow a}{\Rightarrow} aaA \overset{A \rightarrow a}{\Rightarrow} aaa$, which has value $R(S \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)$. The value of the sentence is the sum of the values of the two derivations,

$$\begin{array}{l} \big[R(S \rightarrow AA) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)\big] \quad \oplus \\ \big[R(S \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)\big] \end{array}$$

2.4.2 Item Derivations. Next, we define item derivations, i.e., derivations using the item-based description of the parser. We define item derivation in such a way that for a correct parser description, there is exactly one item derivation for each grammar derivation. The value of a sentence using the parser is the sum of the value of all item derivations of the goal item. Just as with grammar derivations, individual item derivations are finite, but there may be infinitely many item or grammar derivations of a sentence.

We say that $\frac{a_1 \ldots a_k}{b} c_1 \ldots c_j$ is an **instantiation** of deduction rule $\frac{A_1 \ldots A_k}{B} C_1 \ldots C_j$ whenever the first expression is a variable-free instance of the second; that is, the first expression is the result of consistently substituting constant terms for each variable in the second. Now, we can define an **item derivation tree**. Intuitively, an item derivation

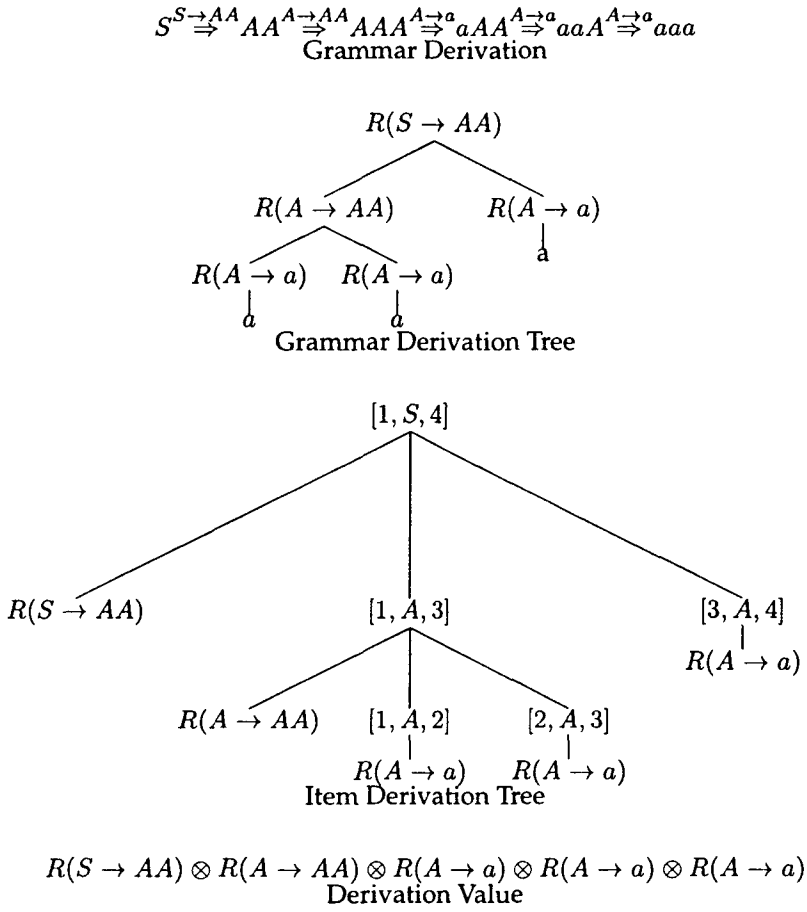

Figure 6
Grammar derivation, grammar derivation tree, item derivation tree, and derivation value.

tree for x just gives a way of deducing x from the grammar rules. We define an item derivation tree recursively. The base case is rules of the grammar: $\langle r \rangle$ is an item derivation tree, where r is a rule of the grammar. Also, if $D_{a_1}, \ldots, D_{a_k}, D_{c_1}, \ldots, D_{c_j}$ are derivation trees headed by $a_1 \ldots a_k, c_1 \ldots c_j$ respectively, and if $\frac{a_1 \ldots a_k}{b} c_1 \ldots c_j$ is the instantiation of a deduction rule, then $\langle b\colon D_{a_1}, \ldots, D_{a_k} \rangle$ is also a derivation tree. Notice that the $D_{c_1} \ldots D_{c_j}$ do not occur in this tree: they are side conditions, and although their existence is required to prove that $c_1 \ldots c_j$ could be derived, they do not contribute to the value of the tree. We will write $\frac{a_1 \ldots a_k}{b}$ to indicate that there is an item derivation tree of the form $\langle b\colon D_{a_1}, \ldots, D_{a_k} \rangle$. As mentioned in Section 2.2, we will write $x \in B$ if $bucket(x) = B$ and there is an item derivation tree for x.

We can continue the example of parsing *aaa*, now using the item-based CKY parser of Figure 3. There are two item derivation trees for the goal item; in Figure 6, we give the first as an example, displaying it as a tree, rather than with angle bracket notation, for simplicity.

Notice that an item derivation is a tree, not a directed graph. Thus, an item subderivation could occur multiple times in a given item derivation. This means that

we can have a one-to-one correspondence between item derivations and grammar derivations; loops in the grammar lead to an infinite number of grammar derivations, and an infinite number of corresponding item derivations.

A grammar including rules such as

$$\begin{aligned} S &\rightarrow AAA \\ A &\rightarrow B \\ A &\rightarrow a \\ B &\rightarrow A \\ B &\rightarrow \epsilon \end{aligned}$$

would allow derivations such as $S \Rightarrow AAA \Rightarrow BAA \Rightarrow AA \Rightarrow BA \Rightarrow A \Rightarrow B \Rightarrow \epsilon$. We would include the exact same item derivation showing $A \Rightarrow B \Rightarrow \epsilon$ three times. Similarly, for a derivation such as $A \Rightarrow B \Rightarrow A \Rightarrow B \Rightarrow A \Rightarrow a$, we would have a corresponding item derivation tree that included multiple uses of the $A \rightarrow B$ and $B \rightarrow A$ rules.

2.4.3 Value of Item Derivation. The value of an item derivation D, $V(D)$, is the product of the value of its rules, $R(r)$, in the same order that they appear in the item derivation tree. Since rules occur only in the leaves of item derivation trees, the order is precisely determined. For an item derivation tree D with rule values $d_1, d_2, \ldots, d_j$ as its leaves,

$$V(D) = \bigotimes_{i=1}^{j} R(d_i) \tag{3}$$

Alternatively, we can write this equation recursively as

$$V(D) = \begin{cases} R(D) & \text{if } D \text{ is a rule} \\ \bigotimes_{i=1}^{k} V(D_i) & \text{if } D = \langle b\text{: } D_1, \ldots, D_k \rangle \end{cases} \tag{4}$$

Continuing our example, the value of the item derivation tree of Figure 6 is

$$R(S \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)$$

the same as the value of the first grammar derivation.

Let $inner(x)$ represent the set of all item derivation trees headed by an item x. Then the value of x is the sum of all the values of all item derivation trees headed by x. Formally,

$$V(x) = \bigoplus_{D \in inner(x)} V(D)$$

The value of a sentence is just the value of the goal item, $V(goal)$.

2.4.4 Iso-valued Derivations. In certain cases, a particular grammar derivation and a particular item derivation will have the same value for any semiring and any rule value function R. In this case, we say that the two derivations are **iso-valued**. In particular, if and only if the same rules occur in the same order in both derivations, then their values will always be the same, and they are iso-valued. In Figure 6, the grammar derivation and item derivation meet this condition. In some cases, a grammar derivation and an

item derivation will have the same value for any commutative semiring and any rule value function. In this case, we say that the derivations are **commutatively iso-valued**.

Finishing our example, the value of the goal item given our example sentence is just the sum of the values of the two item-based derivations,

$$\begin{aligned}&\left[R(S \to AA) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a) \otimes R(A \to a)\right] \quad \oplus \\ &\left[R(S \to AA) \otimes R(A \to a) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a)\right]\end{aligned}$$

This value is the same as the value of the sentence according to the grammar.

2.4.5 Conditions for Correctness. We can now specify the conditions for an item-based description to be correct.

Theorem 1
Given an item-based description I, if for every grammar G, there exists a one-to-one correspondence between the item derivations using I and the grammar derivations, and the corresponding derivations are iso-valued, then for every complete semiring, the value of a given input $w_1 \ldots w_n$ is the same according to the grammar as the value of the goal item. (If the semiring is commutative, then the corresponding derivations need only be commutatively iso-valued.)

Proof
The proof is very simple; essentially, each term in each sum occurs in the other. By hypothesis, for a given input, there are grammar derivations $E_1 \ldots E_k$ (for $0 \leq k \leq \infty$) and corresponding item derivation trees $D_1 \ldots D_k$ of the goal item. Since corresponding items are iso-valued, for all i, $V(E_i) = V(D_i)$. (If the semiring is commutative, then since the items are commutatively iso-valued, it is still the case that for all i, $V(E_i) = V(D_i)$.) Now, since the value of the string according to the grammar is just $\bigoplus_i V(E_i) = \bigoplus_i V(D_i)$, and the value of the goal item is $\bigoplus_i V(D_i)$, the value of the string according to the grammar equals the value of the goal item. □

There is one additional condition for an item-based description to be usable in practice, which is that there be only a finite number of derivable items for a given input sentence; there may, however, be an infinite number of derivations of any item.

2.5 The Derivation Semirings
All of the semirings we use should be familiar, except for the derivation semirings, which we now describe. These semirings, unlike the other semirings described in Figure 5, are not commutative under their multiplicative operator, concatenation.

In many parsers, it is conventional to compute parse forests: compact representations of the set of trees consistent with the input. We will use a related concept, derivation forests, a compact representation of the set of derivations consistent with the input, which corresponds to the parse forest for CFGs, but is easily extended to other formalisms.

Often, we will not be interested in the set of all derivations, but only in the most probable derivation. The Viterbi-derivation semiring computes this value. Alternatively, we might want the n best derivations, which would be useful if the output of the parser were passed to another stage, such as semantic disambiguation; this value is computed by the Viterbi-n-best derivation semiring.

Notice that each of the derivation semirings can also be used to create transducers. That is, we simply associate strings rather than grammar rules with each

rule value. Instead of grammar rule concatenation, we perform string concatenation. The derivation semiring then corresponds to nondeterministic transductions; the Viterbi semiring corresponds to a weighted or probabilistic transducer; and the Viterbi-n-best semiring could be used to get n-best lists from probabilistic transducers.

2.5.1 Derivation Forest. The derivation forest semiring consists of sets of derivations, where a derivation is a list of rules of the grammar.[4] Sets containing one rule, such as $\{\langle X \rightarrow YZ \rangle\}$ for a CFG, constitute the primitive elements of the semiring. The additive operator $\cup$ produces a union of derivations, and the multiplicative operator $\cdot$ produces the concatenation, one derivation concatenated with the next. The concatenation operation $(\cdot)$ is defined on both derivations and sets of derivations; when applied to a set of derivations, it produces the set of pairwise concatenations. The additive identity is simply the empty set, $\emptyset$: union with the empty set is an identity operation. The multiplicative identity is the set containing the empty derivation, $\{\langle\rangle\}$: concatenation with the empty derivation is an identity operation. Derivations need not be complete. For instance, for CFGs, $\{\langle X \rightarrow YZ,\ Y \rightarrow y \rangle\}$ is a valid element, as is $\{\langle Y \rightarrow y,\ X \rightarrow x \rangle\}$. In fact, $\{\langle X \rightarrow A,\ B \rightarrow b \rangle\}$ is a valid element, although it could not occur in a valid grammar derivation, or in a correctly functioning parser. An example of concatenation of sets is $\{\langle A \rightarrow a \rangle, \langle B \rightarrow b \rangle\} \cdot \{\langle C \rightarrow c \rangle, \langle D \rightarrow d \rangle\} = \{\langle A \rightarrow a, C \rightarrow c \rangle, \langle A \rightarrow a, D \rightarrow d \rangle, \langle B \rightarrow b, C \rightarrow c \rangle, \langle B \rightarrow b, D \rightarrow d \rangle\}$.

Potentially, derivation forests are sets of infinitely many items. However, it is still possible to store them using finite-sized representations. Elsewhere (Goodman 1998), we show how to implement derivation forests efficiently, using pointers, in a manner analogous to the typical implementation of parse forests, and also similar to the work of Billot and Lang (1989). Using these techniques, both union and concatenation can be implemented in constant time, and even infinite unions will be reasonably efficient.

2.5.2 Viterbi-derivation Semiring. The Viterbi-derivation semiring computes the most probable derivation of the sentence, given a probabilistic grammar. Elements of this semiring are a pair, a real number v and a derivation forest E, i.e., the set of derivations with score v. We define $\max_{Vit}$, the additive operator, as

$$\max_{Vit}(\langle v, E \rangle, \langle w, D \rangle) = \begin{cases} \langle v, E \rangle & \text{if } v > w \\ \langle w, D \rangle & \text{if } v < w \\ \langle v, E \cup D \rangle & \text{if } v = w \end{cases}$$

In typical practical Viterbi parsers, when two derivations have the same value, one of the derivations is arbitrarily chosen. In practice, this is usually a fine solution, and one that could be used in a real-world implementation of the ideas in this paper, but from a theoretical viewpoint, the arbitrary choice destroys the associative property of the additive operator, $\max_{Vit}$. To preserve associativity, we keep derivation forests of all elements that tie for best.

The definition for $\max_{Vit}$ is only defined for two elements. Since the operator is associative, it is clear how to define $\max_{Vit}$ for any finite number of elements, but we also need infinite summations to be defined. We use the **supremum**, sup: the supremum of a set is the smallest value at least as large as all elements of the set; that is, it is a

4 This semiring is equivalent to one well known to mathematicians, the polynomials over noncommuting variables.

maximum that is defined in the infinite case. We can now define $\max_{Vit}$ for the case of infinite sums. Let

$$w = \sup_{\langle v,E\rangle \in X} v$$

$$D = \{E | \langle w, E\rangle \in X\}$$

Then $\max_{Vit} X = \langle w, D\rangle$. D is potentially empty, but this causes us no problems in theory, and will not occur in practice. We define $\times_{Vit}$ as

$$\langle v, E\rangle \times_{Vit} \langle w, D\rangle = \langle v \times w, E \cdot D\rangle$$

where $E \cdot D$ represents the concatenation of the two derivation forests.

2.5.3 Viterbi-n-best Semiring. The last kind of derivation semiring is the Viterbi-n-best semiring, which is used for constructing n-best lists. Intuitively, the value of a string using this semiring will be the n most likely derivations of that string (unless there are fewer than n total derivations.) In practice, this is actually how a Viterbi-n-best semiring would typically be implemented. From a theoretical viewpoint, however, this implementation is inadequate, since we must also define infinite sums and be sure that the distributive property holds. Elsewhere (Goodman 1998), we give a mathematically precise definition of the semiring that handles these cases.

3. Efficient Computation of Item Values

Recall that the value of an item x is just $V(x) = \bigoplus_{D \in inner(x)} V(D)$, the sum of the values of all derivation trees headed by x. This definition may require summing over exponentially many or even infinitely many terms. In this section, we give relatively efficient formulas for computing the values of items. There are three cases that must be handled. First is the base case, when x is a rule. In this case, $inner(x)$ is trivially $\{\langle x\rangle\}$, the set containing the single derivation tree x. Thus, $V(x) = \bigoplus_{D \in inner(x)} V(D) = \bigoplus_{D \in \{\langle x\rangle\}} V(D) = V(\langle x\rangle) = R(x)$

The second and third cases occur when x is an item. Recall that each item is associated with a bucket, and that the buckets are ordered. Each item x is either associated with a nonlooping bucket, in which case its value depends only on the values of items in earlier buckets; or with a looping bucket, in which case its value depends potentially on the values of other items in the same bucket. In the case when the item is associated with a nonlooping bucket, if we compute items in the same order as their buckets, we can assume that the values of items $a_1 \ldots a_k$ contributing to the value of item b are known. We give a formula for computing the value of item b that depends only on the values of items in earlier buckets.

For the final case, in which x is associated with a looping bucket, infinite loops may occur, when the value of two items in the same bucket are mutually dependent, or an item depends on its own value. These infinite loops may require computation of infinite sums. Still, we can express these infinite sums in a relatively simple form, allowing them to be efficiently computed or approximated.

3.1 Item Value Formula

Theorem 2

If an item x is not in a looping bucket, then

$$V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i) \tag{5}$$

Proof

Let us expand our notion of inner to include deduction rules: $inner(\frac{a_1 \ldots a_k}{b})$ is the set of all derivation trees of the form $\langle b\text{: } \langle a_1 \ldots \rangle \langle a_2 \ldots \rangle \ldots \langle a_k \ldots \rangle \rangle$. For any item derivation tree that is not a simple rule, there is some $a_1 \ldots a_k, b$ such that $D \in inner(\frac{a_1 \ldots a_k}{b})$. Thus, for any item x,

$$\begin{aligned} V(x) &= \bigoplus_{D \in inner(x)} V(D) \\ &= \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \quad \bigoplus_{D \in inner(\frac{a_1 \ldots a_k}{x})} V(D) \end{aligned} \tag{6}$$

Consider item derivation trees $D_{a_1} \ldots D_{a_k}$ headed by items $a_1 \ldots a_k$ such that $\frac{a_1 \ldots a_k}{x}$. Recall that $\langle x\text{: } D_{a_1}, \ldots, D_{a_k} \rangle$ is the item derivation tree formed by combining each of these trees into a full tree, and notice that $\bigcup_{\substack{D_{a_1} \in inner(a_1), \ldots, \\ D_{a_k} \in inner(a_k)}} \langle x\text{: } D_{a_1}, \ldots, D_{a_k} \rangle = inner(\frac{a_1 \ldots a_k}{x})$.

Therefore

$$\begin{aligned} \bigoplus_{D \in inner(\frac{a_1 \ldots a_k}{x})} V(D) &= \bigoplus_{\substack{D_{a_1} \in inner(a_1), \ldots, \\ D_{a_k} \in inner(a_k)}} V(\langle x\text{: } D_{a_1}, \ldots, D_{a_k} \rangle) \\ &= \bigoplus_{\substack{D_{a_1} \in inner(a_1), \ldots, \\ D_{a_k} \in inner(a_k)}} \bigotimes_{i=1}^{k} V(D_{a_i}) \\ &= \bigotimes_{i=1}^{k} \bigoplus_{D_{a_i} \in inner(a_i)} V(D_{a_i}) \\ &= \bigotimes_{i=1}^{k} V(a_i) \end{aligned}$$

Substituting this back into Equation 6, we get

$$V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i)$$

completing the proof. □

Now, we address the case in which x is an item in a looping bucket. This case requires computation of an infinite sum. We will write out this infinite sum, and discuss how to compute it exactly in all cases, except for one, where we approximate it.

Consider the derivable items $x_1 \ldots x_m$ in some looping bucket B. If we build up derivation trees incrementally, when we begin processing bucket B, only those trees with no items from bucket B will be available, what we will call zeroth generation derivation trees. We can put these zeroth generation trees together to form first generation trees, headed by elements in B. We can combine these first generation trees with each other and with zeroth generation trees to form second generation trees, and so on. Formally, we define the generation of a derivation tree headed by x in bucket B to be the largest number of items in B we can encounter on a path from the root to a leaf.

Consider the set of all trees of generation at most g headed by x. Call this set $inner_{\leq g}(x, B)$. We can define the **$\leq g$ generation value** of an item x in bucket B, $V_{\leq g}(x, B)$:

$$V_{\leq g}(x, B) = \bigoplus_{D \in inner_{\leq g}(x,B)} V(D)$$

Intuitively, as g increases, for $x \in B$, $inner_{\leq g}(x, B)$ becomes closer and closer to $inner(x)$. That is, the finite sum of values in the former approaches the infinite sum of values in the latter. For ω-continuous semirings (which includes all of the semirings considered in this paper), an infinite sum is equal to the supremum of the partial sums (Kuich 1997, 613). Thus,

$$V(x) = \bigoplus_{D \in inner(x,B)} V(D) = \sup_g V_{\leq g}(x, B)$$

It will be easier to compute the supremum if we find a simple formula for $V_{\leq g}(x, B)$.

Notice that for items $x \in B$, there will be no generation 0 derivations, so $V_{\leq 0}(x, B) = 0$. Thus, generation 0 makes a trivial base for a recursive formula. Now, we can consider the general case:

Theorem 3
For x an item in a looping bucket B, and for $g \geq 1$,

$$V_{\leq g}(x, B) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} \begin{cases} V(a_i) & \text{if } a_i \notin B \\ V_{\leq g-1}(a_i, B) & \text{if } a_i \in B \end{cases} \tag{7}$$

The proof parallels that of Theorem 2 (Goodman 1998).

3.2 Solving the Infinite Summation

A formula for $V_{\leq g}(x, B)$ is useful, but what we really need is specific techniques for computing the supremum, $V(x) = \sup_g V_{\leq g}(x, B)$. For all ω-continuous semirings, the supremum of iteratively approximating the value of a set of polynomial equations, as we are essentially doing in Equation 7, is equal to the smallest solution to the equations (Kuich 1997, 622). In particular, consider the equations:

$$V_{\leq \infty}(x, B) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} \begin{cases} V(a_i) & \text{if } a_i \notin B \\ V_{\leq \infty}(a_i, B) & \text{if } a_i \in B \end{cases} \tag{8}$$

where $V_{\leq\infty}(x, B)$ can be thought of as indicating $|B|$ different variables, one for each item x in the looping bucket B. Equation 7 represents the iterative approximation of Equation 8, and therefore the smallest solution to Equation 8 represents the supremum of Equation 7.

One fact will be useful for several semirings: whenever the values of all items $x \in B$ at generation $g+1$ are the same as the values of all items in the preceding generation, g, they will be the same at all succeeding generations, as well. Thus, the value at generation g will be the value of the supremum. Elsewhere (Goodman 1998), we give a trivial proof of this fact.

Now, we can consider various semiring-specific algorithms for computing the supremum. Most of these algorithms are well known, and we have simply extended them from specific parsers (described in Section 7) to the general case, or from one semiring to another.

Notice in this section the wide variety of different algorithms, one for each semiring, and some of them fairly complicated. In a conventional system, these algorithms are interweaved with the parsing algorithm, conflating computation of infinite sums with parsing. The result is algorithms that are both harder to understand, and less portable to other semirings.

We first examine the simplest case, the Boolean semiring. Notice that whenever a particular item has value *TRUE* at generation g, it must also have value *TRUE* at generation $g+1$, since if the item can be derived in at most g generations then it can certainly be derived in at most $g+1$ generations. Thus, since the number of *TRUE* valued items is nondecreasing, and is at most $|B|$, eventually the values of all items must not change from one generation to the next. Therefore, for the Boolean semiring, a simple algorithm suffices: keep computing successive generations, until no change is detected in some generation; the result is the supremum. We can perform this computation efficiently if we keep track of items that change value in generation g and only examine items that depend on them in generation $g+1$. This algorithm is then similar to the algorithm of Shieber, Schabes, and Pereira (1993).

For the counting semiring, the Viterbi semiring, and the derivation forest semiring, we need the concept of a **derivation subgraph**. In Section 2.2 we considered the strongly connected components of the dependency graph, consisting of items that for some sentence could possibly depend on each other, and we put these possibly interdependent items together in looping buckets. For a given sentence and grammar, not all items will have derivations. We will find the subgraph of the dependency graph of items with derivations, and compute the strongly connected components of this subgraph. The strongly connected components of this subgraph correspond to loops that actually occur given the sentence and the grammar, as opposed to loops that might occur for some sentence and grammar, given the parser alone. We call this subgraph the derivation subgraph, and we will say that items in a strongly connected component of the derivation subgraph are part of a loop.

Now, we can discuss the counting semiring (integers under + and $\times$). In the counting semiring, for each item, there are three cases: the item can be in a loop; the item can depend (directly or indirectly) on an item in a loop; or the item does not depend on loops. If the item is in a loop or depends on a loop, its value is infinite. If the item does not depend on a loop in the current bucket, then its value becomes fixed after some generation. We can now give the algorithm: first, compute successive generations until the set of items in B does not change from one generation to the next. Next, compute the derivation subgraph, and its strongly connected components. Items in a strongly connected component (a loop) have an infi-

nite number of derivations, and thus an infinite value. Compute items that depend directly or indirectly on items in loops: these items also have infinite value. Any other items can only be derived in finitely many ways using items in the current bucket, so compute successive generations until the values of these items do not change.

The method for solving the infinite summation for the derivation forest semiring depends on the implementation of derivation forests. Essentially, that representation will use pointers to efficiently represent derivation forests. Pointers, in various forms, allow one to efficiently represent infinite circular references, either directly (Goodman 1999), or indirectly (Goodman 1998). Roughly, the algorithm we will use is to compute the derivation subgraph, and then create pointers analogous to the directed edges in the derivation subgraph, including pointers in loops whenever there is a loop in the derivation subgraph (corresponding to an infinite number of derivations). Details are given elsewhere (Goodman 1998). As in the finite case, this representation is equivalent to that of Billot and Lang (1989).

For the Viterbi semiring, the algorithm is analogous to the Boolean case. Derivations using loops in these semirings will always have values no greater than derivations not using loops, since the value with the loop will be the same as some value without the loop, multiplied by some set of rule probabilities that are at most 1. Since the additive operation is **max**, these lower (or at most equal) looping derivations do not change the value of an item. Therefore, we can simply compute successive generations until values fail to change from one iteration to the next.

Now, consider implementations of the Viterbi-derivation semiring in practice, in which we keep only a representative derivation, rather than the whole derivation forest. In this case, loops do not change values, and we use the same algorithm as for the Viterbi semiring. In an implementation of the Viterbi-n-best semiring, in practice, loops can change values, but at most n times, so the same algorithm used for the Viterbi semiring still works. Elsewhere (Goodman 1998), we describe theoretically correct implementations for both the Viterbi-derivation and Viterbi-n-best semirings that keep all values in the event of ties, preserving addition's associativity.

The last semiring we consider is the inside semiring. This semiring is the most difficult. There are two cases of interest, one of which we can solve exactly, and the other of which requires approximations. In many cases involving looping buckets, all deduction rules will be of the form $\frac{a_1 x}{b}$, where a_1 and b are items in the looping bucket, and x is either a rule, or an item in a previously computed bucket. This case corresponds to the items used for deducing singleton productions, such as those Earley's algorithm uses for rules of the form $A \rightarrow B$ and $B \rightarrow A$. In this case, Equation 8 forms a set of linear equations that can be solved by matrix inversion. In the more general case, as is likely to happen with epsilon rules, we get a set of nonlinear equations, and must solve them by approximation techniques, such as simply computing successive generations for many iterations.[5] Stolcke (1993) provides an excellent discussion of these cases, including a discussion of sparse matrix inversion, useful for speeding up some computations.

5 Note that even in the case where we can only use approximation techniques, this algorithm is relatively efficient. By assumption, in this case, there is at least one deduction rule with two items in the current generation; thus, the number of deduction trees over which we are summing grows exponentially with the number of generations: a linear amount of computation yields the sum of the values of exponentially many trees.

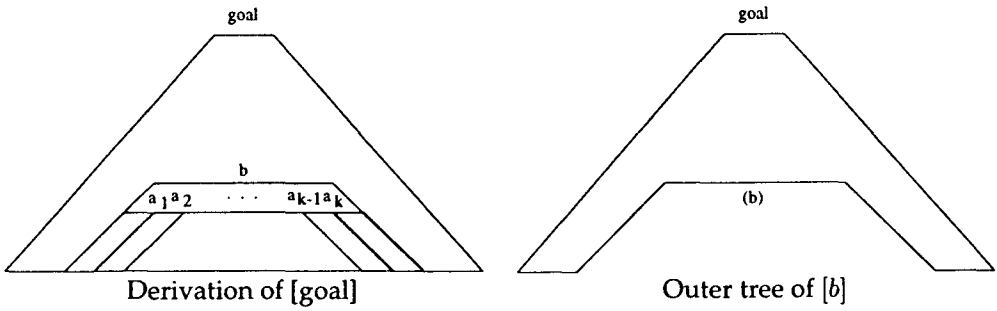

Figure 7
Outside algorithm.

4. Reverse Values

The previous section showed how to compute several of the most commonly used values for parsers, including Boolean, inside, Viterbi, counting, and derivation forest values, among others. Noticeably absent from the list are the outside probabilities, which we define below. In general, computing outside probabilities is significantly more complicated than computing inside probabilities.

In this section, we show how to compute outside probabilities from the same item-based descriptions used for computing inside values. Outside probabilities have many uses, including for reestimating grammar probabilities (Baker 1979), for improving parser performance on some criteria (Goodman 1996b), for speeding parsing in some formalisms, such as data-oriented parsing (Goodman 1996a), and for good thresholding algorithms (Goodman 1997).

We will show that by substituting other semirings, we can get values analogous to the outside probabilities for any commutative semiring; elsewhere (Goodman 1998) we have shown that we can get similar values for many noncommutative semirings as well. We will refer to these analogous quantities as **reverse** values. For instance, the quantity analogous to the outside value for the Viterbi semiring will be called the reverse Viterbi value. Notice that the inside semiring values of a hidden Markov model (HMM) correspond to the forward values of HMMs, and the reverse inside values of an HMM correspond to the backwards values.

Compare the outside algorithm (Baker 1979; Lari and Young 1990), given in Figure 7, to the inside algorithm of Figure 2. Notice that while the inside and recognition algorithms are very similar, the outside algorithm is quite a bit different. In particular, while the inside and recognition algorithms looped over items from shortest to longest, the outside algorithm loops over items in the reverse order, from longest to shortest. Also, compare the inside algorithm's main loop formula to the outside algorithm's main loop formula. While there is clearly a relationship between the two equations, the exact pattern of the relationship is not obvious. Notice that the outside formula is about twice as complicated as the inside formula. This doubled complexity is typical of outside formulas, and partially explains why the item-based description format is so useful: descriptions for the simpler inside values can be developed with relative ease, and then automatically used to compute the twice-as-complicated outside values.[6]

6 Jumping ahead a bit, compare Equation 13 for reverse values to Equation 5 for forward values. Let k be the number of terms above the line. Notice that the reverse values equation sums over k times as many terms as the forward values equation. Parsers where all rules have $k = 1$ terms above the line can only

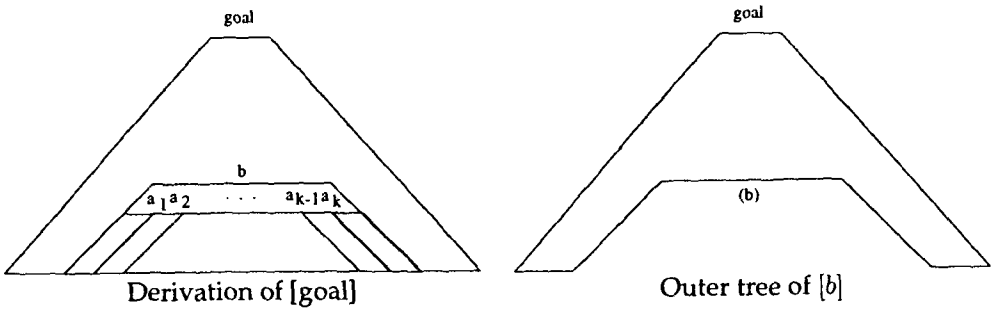

Figure 8
Item derivation tree of [goal] and outer tree of [*b*].

For a context-free grammar, using the CKY parser of Figure 3, recall that the inside probability for an item $[i, A, j]$ is $P(A \rightarrow w_i \ldots w_{j-1})$. The outside probability for the same item is $P(S \stackrel{*}{\Rightarrow} w_1 \ldots w_{i-1} A w_j \ldots w_n)$. Thus, the outside probability has the property that when multiplied by the inside probability, it gives the probability that the start symbol generates the sentence using the given item, $P(S \stackrel{*}{\Rightarrow} w_1 \ldots w_{i-1} A w_j \ldots w_n \stackrel{*}{\Rightarrow} w_1 \ldots w_n)$. This probability equals the sum of the probabilities of all derivations using the given item. Formally, letting $P(D)$ represent the probability of a particular derivation, and $C(D, [i, X, j])$ represent the number of occurrences of item $[i, X, j]$ in derivation D (which for some parsers could be more than one if X were part of a loop),

$$inside(i, X, j) \times outside(i, X, j) = \sum_{D \text{ a derivation}} P(D)\, C(D, [i, X, j])$$

The reverse values in general have an analogous meaning. Let $C(D, x)$ represent the number of occurrences (the count) of item x in item derivation tree D. Then, for an item x, the reverse value $Z(x)$ should have the property

$$V(x) \otimes Z(x) = \bigoplus_{D \text{ a derivation}} V(D) C(D, x) \tag{9}$$

Notice that we have multiplied an element of the semiring, $V(D)$, by an integer, $C(D, x)$. This multiplication is meant to indicate repeated addition, using the additive operator of the semiring. Thus, for instance, in the Viterbi semiring, multiplying by a count other than 0 has no effect, since $x \oplus x = \max(x, x) = x$, while in the inside semiring, it corresponds to actual multiplication. This value represents the sum of the values of all derivation trees that the item x occurs in; if an item x occurs more than once in a derivation tree D, then the value of D is counted more than once.

To formally define the reverse value of an item x, we must first define the **outer** trees $outer(x)$. Consider an item derivation tree of the goal item, containing one or more instances of item x. Remove one of these instances of x, and its children too, leaving a gap in its place. This tree is an outer tree of x. Figure 8 shows an item derivation tree of the goal item, including a subderivation of an item b, derived from terms $a_1, \ldots, a_k$. It also shows an outer tree of b, with b and its children removed; the spot b was removed from is labeled (b).

parse regular grammars, and tend to be less useful. Thus, in most parsers of interest, $k > 1$, and the complexity of (at least some) outside equations, when the sum is written out, is at least doubled.

For an outer tree $D \in outer(x)$, we define its value, $Z(D)$, to be the product of the values of all rules in D, $\bigotimes_{r \in D} R(r)$. Then, the reverse value of an item can be formally defined as

$$Z(x) = \bigoplus_{D \in outer(x)} Z(D) \tag{10}$$

That is, the reverse value of x is the sum of the values of each outer tree of x.

Now, we show that this definition of reverse values has the property described by Equation 9.[7]

Theorem 4

$$V(x) \otimes Z(x) = \bigoplus_{D \text{ a derivation}} V(D)C(D,x)$$

Proof
First, observe that

$$V(x) \otimes Z(x) = \left(\bigoplus_{I \in inner(x)} V(I) \right) \otimes \bigoplus_{O \in outer(x)} Z(O) = \bigoplus_{I \in inner(x)} \bigoplus_{O \in outer(x)} V(I) \otimes Z(O) \tag{11}$$

Next, we argue that this last expression equals the expression on the right-hand side of Equation 9, $\bigoplus_D V(D)C(D,x)$. For an item x, any outer part of an item derivation tree for x can be combined with any inner part to form a complete item derivation tree. That is, any $O \in outer(x)$ and any $I \in inner(x)$ can be combined to form an item derivation tree D containing x, and any item derivation tree D containing x can be decomposed into such outer and inner trees. Thus, the list of all combinations of outer and inner trees corresponds exactly to the list of all item derivation trees containing x. In fact, for an item derivation tree D containing $C(D,x)$ instances of x, there are $C(D,x)$ ways to form D from combinations of outer and inner trees. Also, notice that for D combined from O and I

$$V(I) \otimes Z(O) = \bigotimes_{r \in I} R(r) \otimes \bigotimes_{r \in O} R(r) = \bigotimes_{r \in D} R(r) = V(D)$$

Thus,

$$\bigoplus_{I \in inner(x)} \bigoplus_{O \in outer(x)} V(I) \otimes Z(O) = \bigoplus_{D} V(D)C(D,x) \tag{12}$$

Combining Equation 11 with Equation 12, we see that

$$V(x) \otimes Z(x) = \bigoplus_{D \text{ a derivation}} V(D)C(D,x)$$

completing the proof. □

7 We note that satisfying Equation 9 is a useful but not sufficient condition for using reverse inside values for grammar reestimation. While this definition will typically provide the necessary values for the E step of an E-M algorithm, additional work will typically be required to prove this fact; Equation 9 should be useful in such a proof.

There is a simple, recursive formula for efficiently computing reverse values. Recall that the basic equation for computing forward values not involved in loops was

$$V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i)$$

At this point, for conciseness, we introduce a nonstandard notation. We will soon be using many sequences of the form $1, 2, \ldots, j-2, j-1, j+1, j+2, \ldots, k-1, k$. We denote such sequences by $1, \ldots^{j}, k$. By extension, we will also write $f(1), \ldots^{j}, f(k)$ to indicate a sequence of the form $f(1), f(2), \ldots, f(j-2), f(j-1), f(j+1), f(j+2), \ldots, f(k-1), f(k)$.

Now, we can give a simple formula for computing reverse values $Z(x)$ not involved in loops:

Theorem 5
For items $x \in B$ where B is nonlooping,

$$Z(x) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \otimes \bigotimes_{i=1, \ldots^{j}, k} V(a_i) \tag{13}$$

unless x is the goal item, in which case $Z(x) = 1$, the multiplicative identity of the semiring.

Proof
The simple case is when x is the goal item. Since an outer tree of the goal item is a derivation of the goal item, with the goal item and its children removed, and since we assumed in Section 2.2 that the goal item can only appear in the root of a derivation tree, the outer trees of the goal item are all empty. Thus,

$$Z(goal) = \bigoplus_{D \in outer(goal)} Z(D) = Z(\{\langle\rangle\}) = \bigotimes_{r \in \{\langle\rangle\}} R(r) = 1$$

As mentioned in Section 2.1, the value of the empty product is the multiplicative identity.

Now, we consider the general case. We need to expand our concept of *outer* to include deduction rules, where $outer(j, \frac{a_1 \ldots a_k}{b})$ is an item derivation tree of the goal item with one subtree removed, a subtree headed by a_j whose parent is b and whose siblings are headed by $a_1, \ldots^{j}, a_k$. Notice that for every outer tree $D \in outer(x)$, there is exactly one $j, a_1, \ldots, a_k$, and b such that $x = a_j$ and $D \in outer(j, \frac{a_1 \ldots a_k}{b})$: this corresponds to the deduction rule used at the spot in the tree where the subtree headed by x was deleted. Figure 9 illustrates the idea of putting together an outer tree of b with inner trees for $a_1, \ldots^{j}, a_k$ to form an outer tree of $x = a_j$. Using this observation,

$$\begin{aligned} Z(x) &= \bigoplus_{D \in outer(x)} Z(D) \\ &= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{D \in outer\left(j, \frac{a_1 \ldots a_k}{b}\right)} Z(D) \end{aligned} \tag{14}$$

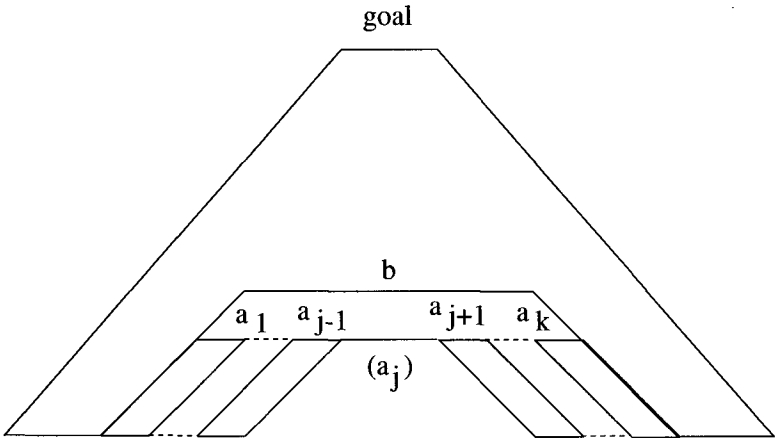

Figure 9
Combining an outer tree with inner trees to form an outer tree.

Now, consider all of the outer trees $outer(j, \frac{a_1 \ldots a_k}{b})$. For each item derivation tree $D_{a_1} \in inner(a_1), \stackrel{-j}{\ldots}, D_{a_k} \in inner(a_k)$ and for each outer tree $D_b \in outer(b)$, there will be one outer tree in the set $outer(j, \frac{a_1 \ldots a_k}{b})$. Similarly, each tree in $outer(j, \frac{a_1 \ldots a_k}{b})$ can be decomposed into an outer tree in $outer(b)$ and derivation trees for $a_1, \stackrel{-j}{\ldots}, a_k$. Then,

$$
\begin{aligned}
&\bigoplus_{D \in outer\left(j, \frac{a_1 \ldots a_k}{b}\right)} Z(D) \\
&= \bigoplus_{\substack{D_b \in outer(b), \\ D_{a_1} \in inner(a_1), \stackrel{-j}{\ldots}, \\ D_{a_k} \in inner(a_k)}} Z(D_b) \otimes V(D_{a_1}) \otimes \stackrel{-j}{\cdots} \otimes V(D_{a_k}) \\
&= \left(\bigoplus_{D_b \in outer(b)} Z(D_b) \right) \otimes \left(\bigoplus_{D_{a_1} \in inner(a_1)} V(D_{a_1}) \right) \otimes \stackrel{-j}{\cdots} \otimes \left(\bigoplus_{D_{a_k} \in inner(a_k)} V(D_{a_k}) \right) \\
&= Z(b) \otimes V(a_1) \otimes \stackrel{-j}{\cdots} \otimes V(a_k) \\
&= Z(b) \otimes \bigotimes_{i=1, \stackrel{-j}{\ldots}, k} V(a_i)
\end{aligned}
\tag{15}
$$

Substituting equation 15 into equation 14, we conclude that

$$
Z(x) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \otimes \bigotimes_{i=1, \stackrel{-j}{\ldots}, k} V(a_i)
$$

completing the general case.

Computing the reverse values for loops is somewhat more complicated, and as in the forward case, requires an infinite sum, and the use of the concept of generation.

We define the generation g of an outer tree D of item x in bucket B to be the number of items in bucket B on the path between the root and the removal point, inclusive. We can then let $Z_{\leq g}(x, B)$ represent the sum of the values of all trees headed by x of generation at most g. In the base case, $Z_{\leq 0}(x, B) = 0$. For ω-continuous semirings, $Z_{\leq g}(x, B)$ approaches $Z(x)$ as g approaches ∞. We can give a recursive equation for $Z_{\leq g}(x, B)$ as follows, using a proof similar to that of Theorem 5 (Goodman 1998):

Theorem 6
For items $x \in B$ and $g \geq 1$,

$$Z_{\leq g}(x, B) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigotimes_{i=1, \ldots, j, \ldots, k} V(a_i) \right) \otimes \begin{cases} Z_{\leq g-1}(b, B) & \text{if } b \in B \\ Z(b) & \text{if } b \notin B \end{cases} \quad (16)$$

5. Semiring Parser Execution

Executing a semiring parser is fairly simple. There is, however, one issue that must be dealt with before we can actually begin parsing. A semiring parser computes the values of items in the order of the buckets they fall into. Thus, before we can begin parsing, we need to know which items fall into which buckets, and the ordering of those buckets. There are three approaches to determining the buckets and ordering that we will discuss in this section. The first approach is a simple, brute-force enumeration of all items, derivable or not, followed by a topological sort. This approach will have suboptimal time and space complexity for some item-based descriptions. The second approach is to use an agenda parser in the Boolean semiring to determine the derivable items and their dependencies, and to then perform a topological sort. This approach has optimal time complexity, but typically suboptimal space complexity. The final approach is to use bucketing code specific to the item-based interpreter. This achieves optimal performance for additional programming effort.

The simplest way to determine the bucketing is to simply enumerate all possible items for the given item-based description, grammar, and input sentence. Then, we compute the strongly connected components and a partial ordering; both steps can be done in time proportional to the number of items plus the number of dependencies (Cormen, Leiserson, and Rivest 1990, Chap. 23). For some parsers, this technique has optimal time complexity, although poor space complexity. In particular, for the CKY algorithm, the time complexity is optimal, but since it requires computing and storing all possible $O(n^3)$ dependencies between the items, it takes significantly more space than the $O(n^2)$ space required in the best implementation. In general, the brute-force technique raises the space complexity to be the same as the time complexity. Furthermore, for some algorithms, such as Earley's algorithm, there could be a significant time complexity added as well. In particular, Earley's algorithm may not need to examine all possible items. For certain grammars, Earley's algorithm examines only a linear number of items and a linear number of dependencies, even though there are $O(n^2)$ possible items, and $O(n^3)$ possible dependencies. Thus the brute-force approach would require $O(n^3)$ time and space instead of $O(n)$ time and space, for these grammars.

The next approach to finding the bucketing solves the time complexity problem. In this approach, we first parse in the Boolean semiring, using the agenda parser described by Shieber, Schabes, and Pereira (1995), and then we perform a topological sort. The techniques that Shieber, Schabes, and Pereira use work well for the Boolean semiring, where items only have value *TRUE* or *FALSE*, but cannot be used directly for

for *current* := first bucket **to** last bucket
  if *current* is a looping bucket
    /* *replace with semiring-specific code* */
    for $x \in current$
      $V[x, 0] = 0;$
    for $g := 1$ **to** ∞
      for each $x \in current$, $a_1 \ldots a_k$ s.t. $\frac{a_1 \ldots a_k}{x}$

$$V[x,g] := V[x,g] \oplus \bigotimes_{i=1}^{k} \begin{cases} V[a_i] & a_i \notin current \\ V[a_i, g-1] & a_i \in current \end{cases}$$

    for each $x \in current$
      $V[x] := V[x, \infty];$
  else
    for each $x \in current$, $a_1 \ldots a_k$ s.t. $\frac{a_1 \ldots a_k}{x}$
      $V[x] := V[x] \oplus \bigotimes_{i=1}^{k} V[a_i];$
return $V[goal];$

Figure 10
Forward semiring parser interpreter.

other semirings. For other semirings, we need to make sure that the values of items are not computed until after the values of all items they depend on are computed. However, we can use the algorithm of Shieber, Schabes, and Pereira to compute all of the items that are derivable, and to store all of the dependencies between the items. Then we perform a topological sort on the items. The time complexity of both the agenda parser and the topological sort will be proportional to the number of dependencies, which will be proportional to the optimal time complexity. Unfortunately, we still have the space complexity problem, since again, the space used will be proportional to the number of dependencies, rather than to the number of items.

The third approach to bucketing is to create algorithm-specific bucketing code; this results in parsers with both optimal time and optimal space complexity. For instance, in a CKY-style parser, we can simply create one bucket for each length, and place each item into the bucket for its length. For some algorithms, such as Earley's algorithm, special-purpose code for bucketing might have to be combined with code to make sure all and only derivable items are considered (using triggering techniques described by Shieber, Schabes, and Pereira) in order to achieve optimal performance.

Once we have the bucketing, the parsing step is fairly simple. The basic algorithm appears in Figure 10. We simply loop over each item in each bucket. There are two types of buckets: looping buckets, and nonlooping buckets. If the current bucket is a looping bucket, we compute the infinite sum needed to determine the bucket's values; in a working system, we substitute semiring-specific code for this section, as described in Section 3.2. If the bucket is not a looping bucket, we simply compute all of the possible instantiations that could contribute to the values of items in that bucket. Finally, we return the value of the goal item.

The reverse semiring parser interpreter is very similar to the forward semiring parser interpreter. The differences are that in the reverse semiring parser interpreter, we traverse the buckets in reverse order, and we use the formulas for the reverse values, rather than the forward values. Elsewhere (Goodman 1998), we give a simple inductive proof to show that both interpreters compute the correct values.

There are two other implementation issues. First, for some parsers, it will be possible to discard some items. That is, some items serve the role of temporary variables, and can be discarded after they are no longer needed, especially if only the forward values are going to be computed. Also, some items do not depend on the input string, but only on the rule value function of the grammar. The values of these items can be precomputed.

6. Examples

In this section, we survey other results that are described in more detail elsewhere (Goodman 1998), including examples of formalisms that can be parsed using item-based descriptions, and other uses for the technique of semiring parsing.

6.1 Finite State Automata and Hidden Markov Models

Nondeterministic finite-state automata (NFAs) and HMMs turn out to be examples of the same underlying formalism, whose values are simply computed in different semirings. Other semirings lead to other interesting values. For HMMs, notice that the forward values are simply the forward inside values; the backward values are the reverse values of the inside semiring; and Viterbi values are the forward values of the Viterbi semiring. For NFAs, we can use the Boolean semiring to determine whether a string is in the language of an NFA; we can use the counting semiring to determine how many state sequences there are in the NFA for a given string; and we can use the derivation forest semiring to get a compact representation of all state sequences in an NFA for an input string. A single item-based description can be used to find all of these values.

6.2 Prefix Values

For language modeling, it may be useful to compute the prefix probability of a string. That is, given a string $w_1 \ldots w_n$, we may wish to know the total probability of all sentences beginning with that string,

$$\sum_{k \geq 0, v_1, \ldots, v_k} P(S \rightarrow w_1 \ldots w_n v_1 \ldots v_k)$$

where $v_1 \ldots v_k$ represent words that could possibly follow $w_1 \ldots w_n$. Jelinek and Lafferty (1991) and Stolcke (1993) both give algorithms for computing these prefix probabilities. Elsewhere (Goodman 1998), we show how to produce an item-based description of a prefix parser. There are two main advantages to using an item-based description: ease of derivation, and reusability.

First, the conventional derivations are somewhat complex, requiring a fair amount of inside-semiring-specific mathematics. In contrast, using item-based descriptions, we only need to derive a parser that has the property that there is one item derivation for each (complete) grammar derivation that would produce the prefix. The value of any prefix given the parser will then automatically be the sum of all grammar derivations that include that prefix.

The other advantage is that the same description can be used to compute many values, not just the prefix probability. For instance, we can use this description with the Viterbi-derivation semiring to find the most likely derivation that includes this prefix. With this most likely derivation, we could begin interpretation of a sentence even before the sentence was finished being spoken to a speech recognition system. We could even use the Viterbi-n-best semiring to find the n most likely derivations that include this prefix, if we wanted to take into account ambiguities present in parses of the prefix.

6.3 Beyond Context-Free

There has been quite a bit of previous work on the intersection of formal language theory and algebra, as described by Kuich (1997), among others. This previous work has made heavy use of the fact that there is a strong correspondence between algebraic equations in certain noncommutative semirings, and CFGs. This correspondence has made it possible to manipulate algebraic systems, rather than grammar systems, simplifying many operations.

On the other hand, there is an inherent limit to such an approach, namely a limit to context-free systems. It is then perhaps slightly surprising that we can avoid these limitations, and create item-based descriptions of parsers for weakly context-sensitive grammars, such as tree adjoining grammars (TAGs). We avoid the limitations of previous approaches using two techniques. One technique is to compute derivation trees, rather than parse trees, for TAGs. Computing derivation trees for TAGs is significantly easier than computing parse trees, since the derivation trees are context-free. The other trick we use is to create a set of equations for each grammar and string length rather than creating a set of equations for each grammar, as earlier formulations did. Because the number of equations grows with the string length with our technique, we can recognize strings in weakly context-sensitive languages. Goodman (1998) gives a further explication of this subject, including an item-based description for a simple TAG parser.

6.4 Tomita Parsing

Our goal in this section has been to show that item-based descriptions can be used to simply describe almost all parsers of interest. One parsing algorithm that would seem particularly difficult to describe is Tomita's graph-structured-stack LR parsing algorithm. This algorithm at first glance bears little resemblance to other parsing algorithms. Despite this lack of similarity, Sikkel (1993) gives an item-based description for a Tomita-style parser for the Boolean semiring, which is also more efficient than Tomita's algorithm. Sikkel's parser can be easily converted to our format, where it can be used for ω-continuous semirings in general.

6.5 Graham Harrison Ruzzo (GHR) Parsing

Graham, Harrison, and Ruzzo (1980) describe a parser similar to Earley's, but with several speedups that lead to significant improvements. Essentially, there are three improvements in the GHR parser. First, epsilon productions are precomputed; second, unary productions are precomputed; and, finally, completion is separated into two steps, allowing better dynamic programming.

Goodman (1998) gives a full item-based description of a GHR parser. The forward values of many of the items in our parser related to unary and epsilon productions can be computed off-line, once per grammar, which is an idea due to Stolcke (1993). Since reverse values require entire strings, the reverse values of these items cannot be computed until the input string is known. Because we use a single item-based description for precomputed items and nonprecomputed items, and for forward and reverse values, this combination of off-line and on-line computation is easily and compactly specified.

6.6 Grammar Transformations

We can apply the same techniques to grammar transformations that we have so far applied to parsing. Consider a grammar transformation, such as the Chomsky normal form (CNF) grammar transformation, which takes a grammar with epsilon, unary, and n-ary branching productions, and converts it into one in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$. For any sentence $w_1 \ldots w_n$ its value under the

original grammar in the Boolean semiring (*TRUE* if the sentence can be generated by the grammar, *FALSE* otherwise) is the same as its value under a transformed grammar. Therefore, we say that this grammar transformation is **value preserving** under the Boolean semiring. We can generalize this concept of value preserving to other semirings.

Elsewhere (Goodman 1998), we show that using essentially the same item-based descriptions we have used for parsing, we can specify grammar transformations. The concept of value preserving grammar transformation is already known in the intersection of formal language theory and algebra (Kuich 1997; Kuich and Salomaa 1986; Teitelbaum 1973). Our contribution is to show that these value preserving transformations can be written as simple item-based descriptions, allowing the same computational machinery to be used for grammar transformations as is used for parsing, and to some extent showing the relationship between certain grammar transformations and certain parsers, such as that of Graham, Harrison, and Ruzzo (1980). This uniform method of specifying grammar transformations is similar to, but clearer than, similar techniques used with covering grammars (Nijholt 1980; Leermakers 1989).

7. Previous Work

7.1 Historical Work

The previous work in this area is extensive, including work in deductive parsing, work in statistical parsing, and work in the combination of formal language theory and algebra. This paper can be thought of as synthetic, combining the work in all three areas, although in the course of synthesis, several general formulas have been found, most notably the general formula for reverse values. A comprehensive examination of all three areas is beyond the scope of this paper, but we can touch on a few significant areas of each.

First, there is the work in deductive parsing. This work in some sense dates back to Earley (1970), in which the use of items in parsers is introduced. More recent work (Pereira and Warren 1983; Pereira and Shieber 1987) demonstrates how to use deduction engines for parsing. Finally, both Shieber, Schabes, and Pereira (1995) and Sikkel (1993) have shown how to specify parsers in a simple, interpretable, item-based format. This format is roughly the format we have used here, although there are differences due to the fact that their work was strictly in the Boolean semiring.

Work in statistical parsing has also greatly influenced this work. We can trace this work back to research in HMMs by Baum and his colleagues (Baum and Eagon 1967; Baum 1972). In particular, the work of Baum developed the concept of backward probabilities (in the inside semiring), as well as many of the techniques for computing in the inside semiring. Viterbi (1967) developed corresponding algorithms for computing in the Viterbi semiring. Baker (1979) extended the work of Baum and his colleagues to PCFGs, including to computation of the outside values (or reverse inside values in our terminology). Baker's work is described by Lari and Young (1990). Baker's work was only for PCFGs in CNF, avoiding the need to compute infinite summations. Jelinek and Lafferty (1991) showed how to compute some of the infinite summations in the inside semiring, those needed to compute the prefix probabilities of PCFGs in CNF. Stolcke (1993) showed how to use the same techniques to compute inside probabilities for Earley parsing, dealing with the difficult problems of unary transitions, and the more difficult problems of epsilon transitions. He thus solved all of the important problems encountered in using an item-based parser to compute the inside and outside values (forward and reverse inside values); he also showed how to compute the forward Viterbi values.

The final area of work is in formal language theory and algebra. Although it is not widely known, there has been quite a bit of work showing how to use formal power series to elegantly derive results in formal language theory, dating back to Chomsky and Schützenberger (1963). The major classic results can be derived in this framework, but with the added benefit that they apply to all commutative ω-continuous semirings. The most accessible introduction to this literature we have found is by Kuich (1997). There are also books by Salomaa and Soittola (1978) and Kuich and Salomaa (1986).

One piece of work deserves special mention. Teitelbaum (1973) showed that any semiring could be used in the CKY algorithm, laying the foundation for much of the work that followed.

In summary, this paper synthesizes work from several different related fields, including deductive parsing, statistical parsing, and formal language theory; we emulate and expand on the earlier synthesis of Teitelbaum. The synthesis here is powerful: by generalizing and integrating many results, we make the computation of a much wider variety of values possible.

7.2 Recent Similar Work

There has also been recent similar work by Tendeau (1997b, 1997a). Tendeau (1997b) gives an Earley-like algorithm that can be adapted to work with complete semirings satisfying certain conditions. Unlike our version of Earley's algorithm, Tendeau's version requires time $O(n^{L+1})$ where L is the length of the longest right-hand side, as opposed to $O(n^3)$ for the classic version, and for our description. While one could split right-hand sides of rules to make them binary branching, speeding Tendeau's version up, this would then change values in the derivation semirings. Tendeau (1997b, 1997a) introduces a parse forest semiring, similar to our derivation forest semiring, in that it encodes a parse forest succinctly. To implement this semiring, Tendeau's version of rule value functions take as their input not only a nonterminal, but also the span that it covers; this is somewhat less elegant than our version. Tendeau (1997a) gives a generic description for dynamic programming algorithms. His description is very similar to our item-based descriptions, except that it does not include side conditions. Thus, algorithms such as Earley's algorithm cannot be described in Tendeau's formalism in a way that captures their efficiency.

There are some similarities between our work and the work of Koller, McAllester, and Pfeffer (1997), who create a general formalism for handling stochastic programs that makes it easy to compute inside and outside probabilities. While their formalism is more general than item-based descriptions, in that it is a good way to express any stochastic program, it is also less compact than ours for expressing most dynamic programming algorithms. Our formalism also has advantages for approximating infinite sums, which we can do efficiently, and in some cases exactly. It would be interesting to try to extend item-based descriptions to capture some of the formalisms covered by Koller, McAllester, and Pfeffer, including Bayes' nets.

8. Conclusion

In this paper, we have given a simple item-based description format that can be used to describe a very wide variety of parsers. These parsers include the CKY algorithm, Earley's algorithm, prefix probability computation, a TAG parsing algorithm, Graham, Harrison, Ruzzo (GHR) parsing, and HMM computations. We have shown that this description format makes it easy to find parsers that compute values in any ω-continuous semiring. The same description can be used to find reverse values in commutative ω-

continuous semirings, and in many noncommutative ones as well. This description format can also be used to describe grammar transformations, including transformations to CNF and GNF, which preserve values in any commutative ω-continuous semiring.

While theoretical in nature, this paper is of some practical value. There are three reasons the results of this paper would be used in practice: first, these techniques make computation of the outside values simple and mechanical; second, these techniques make it easy to show that a parser will work in any ω-continuous semiring; and third, these techniques isolate computation of infinite sums in a given semiring from the parser specification process.

Perhaps the most useful application of these results is in finding formulas for outside values. For parsers such as CKY parsers, finding outside formulas is not particularly burdensome, but for complicated parsers such as TAG parsers, GHR parsers, and others, it can require a fair amount of thought to find these equations through conventional reasoning. With these techniques, the formulas can be found in a simple mechanical way.

The second advantage comes from clarifying the conditions under which a parser can be converted from computing values in the Boolean semiring (a recognizer) to computing values in any ω-continuous semiring. We should note that because in the Boolean semiring, infinite summations can be computed trivially and because repeatedly adding a term does not change results, it is not uncommon for parsers that work in the Boolean semiring to require significant modification for other semirings. For parsers like CKY parsers, verifying that the parser will work in any semiring is trivial, but for other parsers the conditions are more complex. With the techniques in this paper, all that is necessary is to show that there is a one-to-one correspondence between item derivations and grammar derivations. Once that has been shown, any ω-continuous semiring can be used.

The third use of this paper is to separate the computation of infinite sums from the main parsing process. Infinite sums can come from several different phenomena, such as loops from productions of the form $A \rightarrow A$; productions involving ϵ; and left recursion. In traditional procedural specifications, the solution to these difficult problems is intermixed with the parser specification, and makes the parser specific to semirings using the same techniques for solving the summations.

It is important to notice that the algorithms for solving these infinite summations vary fairly widely, depending on the semiring. On the one hand, Boolean infinite summations are nearly trivial to compute. For other semirings, such as the counting semiring, or derivation forest semiring, more complicated computations are required, including the detection of loops. Finally, for the inside semiring, in most cases only approximate techniques can be used, although in some cases, matrix inversion can be used. Thus, the actual parsing algorithm, if specified procedurally, can vary quite a bit depending on the semiring.

On the other hand, using our techniques makes infinite sums easier to deal with in two ways. First, these difficult problems are separated out, relegated conceptually to the parser interpreter, where they can be ignored by the constructor of the parsing algorithm. Second, because they are separated out, they can be solved once, rather than again and again. Both of these advantages make it significantly easier to construct parsers. Even in the case where, for efficiency, loops are precomputed offline, as in GHR parsing, the same item-based representation and interpreter can be used.

In summary, the techniques of this paper will make it easier to compute outside values, easier to construct parsers that work for any ω-continuous semiring, and easier

to compute infinite sums in those semirings. In 1973, Teitelbaum wrote:

> We have pointed out the relevance of the theory of algebraic power series in noncommuting variables in order to minimize further piecemeal rediscovery (page 199).

Many of the techniques needed to parse in specific semirings continue to be rediscovered, and outside formulas are derived without observation of the basic formulas given here. We hope this paper will bring about Teitelbaum's wish.

Acknowledgments

I would like to thank Stan Chen, Barbara Grosz, Luke Hunsberger, Fernando Pereira, and Stuart Shieber, for helpful comments and discussions, as well as the anonymous reviewers for their comments on earlier drafts. This work was funded in part by the National Science Foundation through Grant IRI-9350192, Grant IRI-9712068, and an NSF Graduate Student Fellowship.

References

Baker, James K. 1979. Trainable grammars for speech recognition. In *Proceedings of the Spring Conference of the Acoustical Society of America*, pages 547–550, Boston, MA, June.

Baum, Leonard E. 1972. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, 3:1–8.

Baum, Leonard E. and J. A. Eagon. 1967. An inequality with application to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bulletin of the American Mathematicians Society*, 73:360–363.

Billot, Sylvie and Bernard Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting*, pages 143–151, Vancouver. Association for Computational Linguistics.

Chomsky, Noam and Marcel-Paul Schützenberger. 1963. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holland, pages 118–161.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102.

Goodman, Joshua. 1996a. Efficient algorithms for parsing the DOP model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 143–152, May. Available as cmp-lg/9604008.

Goodman, Joshua. 1996b. Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting*, pages 177–183, Santa Cruz, CA, June. Association for Computational Linguistics. Available as cmp-lg/9605036.

Goodman, Joshua. 1997. Global thresholding and multiple-pass parsing. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 11–25.

Goodman, Joshua. 1998. *Parsing Inside-Out*. Ph.D. thesis, Harvard University. Available as cmp-lg/9805007 and from http://www.eecs.harvard.edu/~goodman/thesis.ps.

Goodman, Joshua. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Graham, Susan L., Michael A. Harrison, and Walter L. Ruzzo. 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July.

Jelinek, Frederick and John D. Lafferty. 1991. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, pages 315–323.

Koller, Daphne, David McAllester, and Avi Pfeffer. 1997. Effective bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 740–747, Providence, RI, August.

Kuich, Werner. 1997. Semirings and formal power series: Their relevance to formal languages and automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*. Springer-Verlag, Berlin, pages 609–677.

Kuich, Werner and Arto Salomaa. 1986. *Semirings, Automata, Languages*. Number 5 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany.

Lari, K. and S. J. Young. 1990. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56.

Leermakers, René. 1989. How to cover a grammar. In *Proceedings of the 27th Annual Meeting*, pages 135–142, Vancouver. Association for Computational Linguistics.

Nijholt, Anton. 1980. *Context-Free Grammars: Covers, Normal Forms, and Parsing*. Number 93 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany.

Pereira, Fernando and Stuart Shieber. 1987. *Prolog and Natural Language Analysis*. Number 10 of CSU Lecture Notes. Center for the Study of Language and Information, Stanford, CA.

Pereira, Fernando and David Warren. 1983. Parsing as deduction. In *Proceedings of the 21st Annual Meeting*, pages 137–44, Cambridge, MA. Association for Computational Linguistics.

Salomaa, Arto and Matti Soittola. 1978. *Automata-Theoretic Aspects of Formal Power Series*. Springer-Verlag, Berlin, Germany.

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

Sikkel, Klaas. 1993. *Parsing Schemata*. Ph.D. thesis, University of Twente, Enschede, The Netherlands.

Stolcke, Andreas. 1993. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Technical Report TR-93-065, International Computer Science Institute, Berkeley, CA. Available as cmp-lg/9411029.

Teitelbaum, Ray. 1973. Context-free error analysis by evaluation of algebraic power series. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 196–199, Austin, TX.

Tendeau, Frédéric. 1997a. Computing abstract decorations of parse forests using dynamic programming and algebraic power series. *Theoretical Computer Science*. To appear.

Tendeau, Frédéric. 1997b. An Earley algorithm for generic attribute augmented grammars and applications. In *Proceedings of the International Workshop on Parsing Technologies 1997*, pages 199–209.

Viterbi, Andrew J. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–267.