Handout 7: Computation & Hierarchical parsing I; Earley's algorithm

1 Representations for dominance/precedence structure

We know that natural languages contain units larger than single words. For example, the sentence,

> I know that this President enjoys the exercise of military power.

plainly consists of two smaller sentences: *I know something* and what it is that I know, namely, that *this President enjoys the exercise of military power*. This second word group functions as a single unit in several senses. First of all, it intuitively stands for what it is that I know, and so is a *meaningful* unit. Second, one can move it around as one chunk—as in, *That this President enjoys the exercise of military power is something I know*. This group of words is a *syntactic* unit. Finally, echoing a common theme, there are other word sequences—not just single words—that can be substituted for it, while retaining grammaticality: *I know that the guy with his finger on the button is the President*. To put it bluntly, there's no way for us to say that *the guy with his finger on the button* and *the President* can both play the same syntactic roles. As we have seen, this leads to an unwelcome network duplication. It does not allow us to say that sentences are built out of hierarchical parts.

We've already seen that there are several reasons to use hierarchical descriptions for natural languages. Let's summarize these here.

- Larger units than single words. Natural languages have word sequences that act as if they were single units.

- Obvious hierarchy and recursion. Natural language sentences themselves may contain embedded sentences, and these sentences in turn may contain sentences.

- Nonadjacent constraints or grammatical relations. Natural languages exhibit constraints over nonadjacent words, as in Subject–Verb agreement.

- Succinctness. Natural languages are built by combining a few types of phrases in different combinations, like Noun Phrases and Verb Phrases. Important: the phrase names themselves, even their existence, is in a sense purely taxonomic, just as word classes are. (Phrases don't exist except as our theoretical apparatus requires them to.)

- Compositional meaning. Natural languages have sentences whose meaning intuitively follows the hierarchical structure of phrases.

A **phrase** or **nonterminal** is a collection of words that behaves alike (= acts identically under some set of operations, like movement). Example:

(1) (i)   John kissed the baby.

 (ii)   The baby was kissed by John.

 (iii) John kissed the baby and the politician.

 (iv)   The baby and the politician were kissed by John.

A **phrase category** (nonterminal), by analogy with a word category, is determined by identity under substitution contexts. For instance, what is called a **noun phrase** is simply an equivalence class of some string of tokens that can be substituted for one another anywhere. Grammars defined by such equivalence classes are therefore called **context-free**.
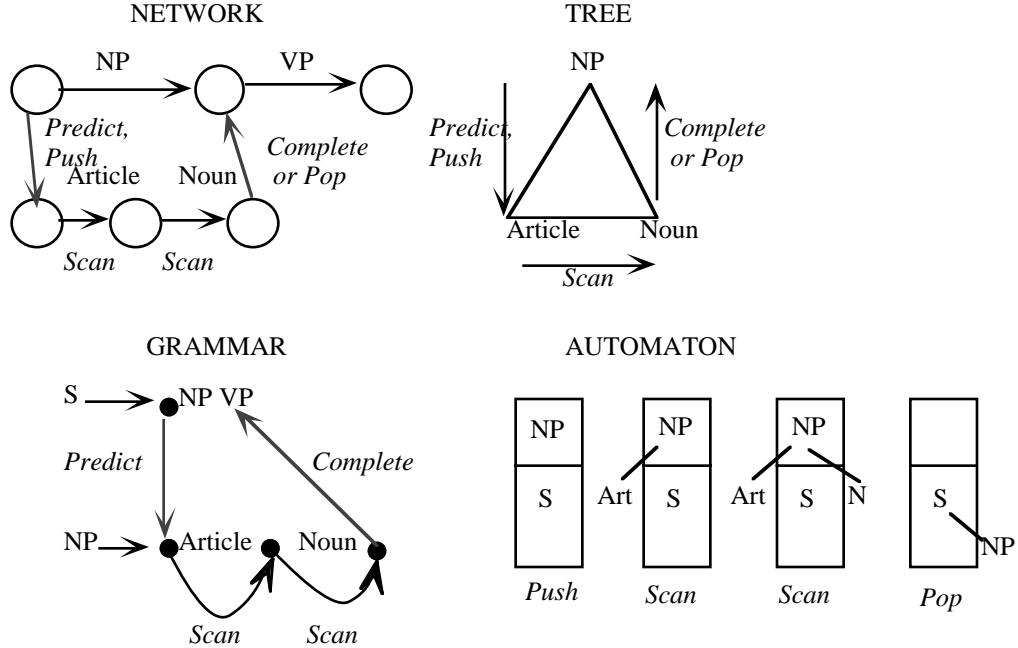
Another way to look at the same situation is to consider what minimal augmentation we need to make to the pure precedence structure of finite transition networks or, equivalently, right- or left-linear grammars in order to accommodate phrases. The only information in a precedence structure is in the binary predicate *precedes*. What we need to add is a new binary predicate *dominates*. Intuitively, it gives us the "vertical" arrangement of phrases while precedence gives us the "horizontal" arrangement. Given a sentence it is traditional to say that *all* phrases are related either by precedence or dominance (but not both); this yields a **tree** structure. It is important to remember that this structure itself is derivative; it is the relations that are of central importance and what are recovered during parsing.

Fact: Hierarchical structure (containing dominance and precedence predicates) is *not* associative. It therefore *can* represent at least *two* types of ambiguity: lexical (or category ambiguity), which it inherits from precedence structure; and **structural** or hierarchical ambiguity. Example: *the dark blue sky*, or *undoable*.

Fact: (Joshi and Levy, 1977). If there are n different phrase categories, then the collection of substitutable equivalence classes of phrases is completely determined by all possible **tree structures** of depth less than or equal to $2n - 1$. (Compare this result to that for pure precedence structures, where the substitution classes for a grammar with n word categories is fixed by all *linear* sequences of length $\leq 2n - 1$.)

Just as with precedence structure, there are four basic ways of representing precedence and hierarchical structure: as **recursive transition networks** (Conway, 1963, *Design of a separable transition diagram compiler*, first used for COBOL!); as **context-free grammars**; as **pushdown automata**; and (graphically) as singly-rooted, acyclic directed graphs, or **trees**.

The figure illustrates how the "abstract operations" in each representation encodes the binary relations of precedence and dominance (which are necessary and sufficient to get us the precedence and dominance relations that we want). **Scan** gives us linear precedence (as before); **push** (**predict**) and **pop** (**complete**) give us the dominance relation.

NETWORK

TREE

GRAMMAR

AUTOMATON

Note how this method introduces a way to say where each phrase begins, ends, and attaches. How can we do this using our FTN network descriptions? We already have defined *one* kind of phrase using our "flat" networks—namely, a Sentence phrase, represented by the whole network. Why not just extend this to all phrases and represent each by a network of its own. It's clear enough what the networks for Noun Phrases and Verb Phrases should be in this case— just our usual finite-transition diagrams will do. What about the network for Sentences? It no longer consists of adjacent *words*, but of adjacent *phrases*: namely, a Noun Phrase followed by a Verb Phrase.

We have now answered the *begin* and *end* questions, but not the *attach* question. We must say how the subnetworks are linked together, and for this we'll introduce two new arc types to encode the dominance relation. We now do this by extending our categorization relation from words to groups of words. To get from state S-0 to state S-1 of the main Sentence network, we must determine that there is a Noun Phrase at that point in the input.

To check whether there is a Noun Phrase in the input, we must refer to the subnetwork labeled Noun Phrase. We can encode this reference to the Noun Phrase network in several ways. One way is just to add *jump* arcs from the Start state of the Sentence to the first state of the Noun Phrase subnetwork. This is also very much like a subroutine call: the subnetwork is the name of a procedure that we can invoke. In this case, we need to know the starting address of the procedure (the subnetwork) so that we can go and find it. Whatever the means, we have defined the *beginning* of a phrase. We now move through the Noun Phrase subnet, checking that all *is-a* relations for single words are satisfied. Reaching the final state for that network, we now have defined the *end* of a Noun Phrase. Now what? We should not just stop, but return to the main network that we came from—to the state on the other end of the Noun Phrase arc, since we have now seen that there is a Noun Phrase at this position in the sentence. In short, each subnetwork checks the *is-a* relation for each distinct phrase, while the main sentence network checks the *is-a* relation for the sentence as a whole. (Note that in

Sentence:
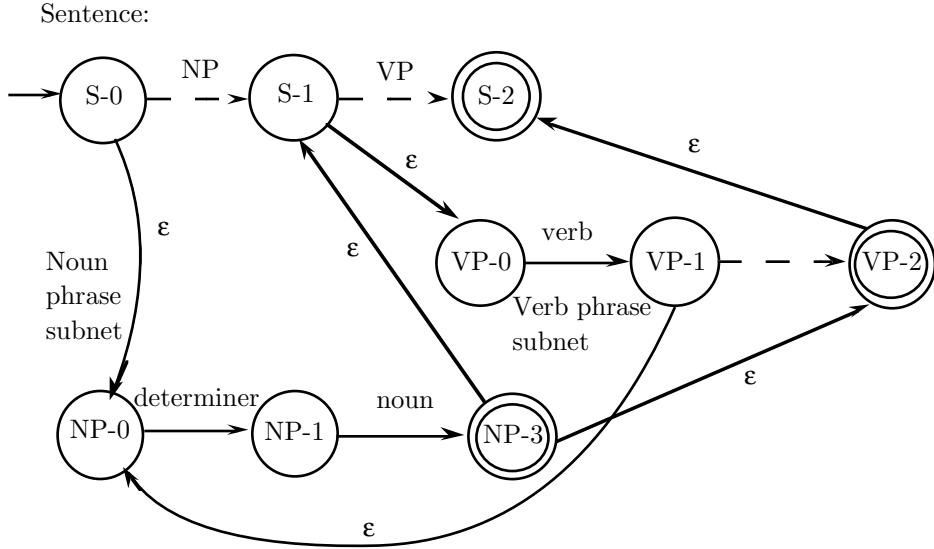
Figure 1: Using *jump* arcs for network calls. In general, a stack must be used to keep track of (arbitrary) return addresses.

general we must keep track of the proper return addresses, and we can use a stack for that.)

This arrangement also answers the *attachment* question: the subunit Noun Phrase fits into the larger picture of things depending on how we get back to a main network from a subnetwork. Similarly, we now have to refer to the Verb Phrase subnetwork to check the *is-a* relation for a Verb Phrase, and then, returning from there, come back to state S-2, and finish. Note that the beginning and end of the main network are defined as before for simple finite-state transition systems, by the start state and end state of the network itself.

The revised network now answers the key questions of hierarchical analysis:

- A phrase of a certain type *begins* by referring to its network description, either as the initial state of the main network (S-0 in our example), or by a *call* from one network to another. The *name* of a phrase comes from the name of the subnetwork.

- A phrase *ends* when we reach the final state of the network describing it. This means that we have *completed* the construction of a phrase of a particular type.

- A phrase is *attached* to the phrase that is the name of the network that referred to (called) it.

To look at these answers from a slightly different perspective, note that each basic network is itself a finite-state automaton that gives the basic *linear* order of elements in a phrase. The same linear order of states and arcs imposes a linear order on phrases. This establishes a *precedes* relation between every pair of elements (words or phrases). Hierarchical *domination* is fixed by the pattern of subnetwork jumps and returns. But because the sentence patterns described by network and subnetwork traversal must lead all the way from the start state to
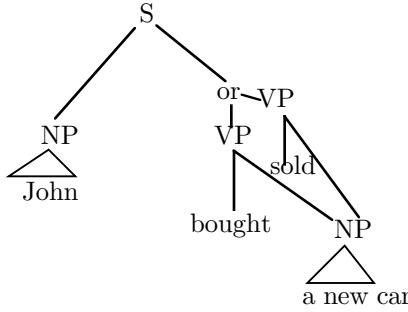
Figure 2: A multidimensional syntactic structure.

a final state without interruption, we can see that such a network *must* relate every pair of elements (=arc labels) by either *dominates* or *precedes*. You can check informally that this seems to be so. If a phrase, like a Noun Phrase, does not dominate another phrase, like a Verb Phrase, then either the Noun Phrase precedes the Verb Phrase or the Verb Phrase precedes the Noun Phrase.

In summary, the hierarchical network system we have described can model languages where *every* pair of phrases or words satisfies either the *dominates* or *precedes* relation. It is interesting to ask whether this is a necessary property of natural languages. Could we have a natural language where two words or phrases were not related by either *dominates* or *precedes*? This seems hard to imagine because words and phrases are spoken linearly, and so seem to automatically satisfy the *precedes* relation if they don't satisfy *dominates*. However, remember that we are interested not just in the external representation of word strings, but also in their internal (mental and computer) representation. There is nothing that bars us or a computer from storing the representation of a linear string of words in a non-linear way.

In fact, there are constructions in English and other languages that suggest just this possibility. If we look at the following sentence,

> *John bought or sold a new car*

we note that *sold a new car* forms a Verb Phrase—as we would expect since *sell* subcategorizes for a Noun Phrase. But what about *bought*? It too demands a Noun Phrase Object. By the restrictions on subcategorization described in the last chapter, we know that this Noun Phrase must appear in the same Verb Phrase that *bought* forms. That is, the Verb Phrase *bought* ... must dominate *a new car*. This is a problem, because the only way we can do this using our hierarchical networks is to have *bought* dominate the Verb Phrase *sold a new car* as well. Suppose, though, that we relax the condition that all elements of a sentence must be in either a *dominates* or *precedes* relation. Then we could have the Verb Phrases *bought* and *sold* bearing no dominance or precedence relation to each other. This would be compatible with the picture in figure **??**. Here we have *two* Verb Phrases dominating *a new car*. Neither dominates or precedes the other. Note how the subcategorization constraint is met by both phrases.

This is an *internal* representation of a sentence. When it comes time to *speak* it must of course be converted into a physically realizable external form. Then, and only then, are the two verbs projected onto a single line, the speech stream.

It's obvious that such examples would cause trouble with the network model we've described so far, because it forces exhaustive *precedes* and *dominates* relations. Most network models resort to some additional machinery, external to the basic operation of the network itself, in order to accommodate such sentences.

More generally, this problem may be attributed to an implicit bias in a linear or hierarchical language description. By saying that a sentence is just a sequence of words or phrases, we are in effect saying that the important relationships among elements in a sentence are limited to those that can be stated in terms of what can be placed next to what. These are properties based on string concatenation, and the network model directly encodes this.

2   How many kinds of phrases are there?

In fact, the possible kinds of phrases in a given natural language are much more restricted than the preceding suggests. For instance, a noun is never the "base" of a verb phrase, or a verb the base of a noun phrase. Rather, every phrase seems to be a "projection" of some bottom lexical category from which it inherits its basic categorial properties. (Traditionally, this was dubbed the "endocentric" character of phrases.)

The base of every phrase of type X is a word of type X: a noun phrase is based on a noun; an adjective phrase (as in *green with envy*), is based on an adjective, and so forth. Thus the basic phrase types in a language are formed from the major word classes. The phrases themselves are derivative. This idea is called $\overline{\text{X}}$ *theory*. (Historical note: these observations were perhaps first noted by Harris (1951), *Methods in Structural Linguistics*; restudied by Lyons (1968), and then Chomsky (1970).) The idea is based on the noting the following kind of generalization (parentheses indicate optionality; * indefinite repetition):

Verb Phrase (VP)→Verb (NP)* (PP)* (S)
Prepositional Phrase (PP)→Preposition (NP)* (PP)* (S)
Adjective Phrase(AP) →Adjective (NP)* (PP)* (S)
Noun Phrase (NP)→Determiner Noun (PP)* (S)

Generalizing: X Phrase (XP)→Verb (NP)* (PP)* (S)
for X=Noun, Verb, Adjective, etc.

In other words, hierarchical grammars for natural languages are much more restricted than one would think at first glance. They might all have the same skeleton templates within a language, so-called **X-bar theory**. From this point of view, in fact, the hierarchical rules are derivative, "projected" from the lexicon. (More on this below.) Indeed, we shall adopt an even more restrictive view that the structure is essentially **binary branching**.

For a more formal account, we first give the definition of a context-free grammar.

Definition 1: A *context-free grammar*, G, is a 4-tuple, $(N, T, P, Start)$, where N is a finite, nonempty set of phrase symbols or *nonterminals*; T is a finite, nonempty set of words or *terminals*; P is a finite set of productions of the form $X \rightarrow \alpha$, where X is a single nonterminal and α is a string of terminals and nonterminals; and $Start$ is a designated *start* symbol.

To define the notion of a language, we define a derivation relation as with FTNs: two strings of nonterminals and terminals α, β are related via a derivation step, $\Rightarrow$, according to grammar G if there is a rule of G, $X \rightarrow \varphi$ such that $\alpha = \psi X \gamma$ and $\beta = \psi \varphi \gamma$. That is, we can obtain the string β from α by replacing X by φ. The string of nonterminals and terminals produced at each step of a derivation from S is called a *sentential form*. $\overset{*}{\Rightarrow}$ is the application of zero or more derivation steps. The *language generated by a context-free grammar*, L(G), is the set of all the terminal strings that can be derived from S using the rules of G:

$$L(G) = \{w | w \in T^* \text{ and } S \overset{*}{\Rightarrow} w\}$$

Notation: Let $L(a)$ denote the label of a node a in a (phrase structure) tree. Let the set of node labels (with respect to a grammar) be denoted V_N. $V = V_N \cup V_T$ (the set of node labels plus terminal elements).

Definition 2: The categories of a phrase structure tree immediately dominating the actual words of the sentence correspond to lexical categories and are called **preterminals** or X^0 **categories**. These are written in the format, e.g., $N^0 \rightarrow dog, cat, \ldots$.

Definition 3: A network (grammar) is *cyclic* if there exists a nonterminal N such that the nonterminal can be reduced to itself (derive itself). For example, the following FTN is cyclic:

$$N \overset{*}{\Rightarrow} N1 \overset{*}{\Rightarrow} N$$

Ambiguity in a context-free grammar is exactly analogous to the notion of different paths through a finite-state transition network. If a context-free grammar G has at least one sentence with two or more derivations, then it is *ambiguous*; otherwise it is *unambiguous*.

Definition 4: A grammar is *infinitely ambiguous* if for some sentences there are an infinite number of derivations; otherwise, it is *finitely* ambiguous.

Example. The following grammar is infinitely ambiguous and cyclic.

    Start→ S
    S→ S
    S→ a

Neither of these cases seem to arise in natural language. Consider the second case. Such rules are generally not meaningful linguistically, because a nonbranching chain introduces no new description in terms of *is-a* relationships. For example, consider the grammar fragment,

    S→ NP VP   NP→NP
    NP→Det Noun

This grammar has a derivation tree with NP–NP—NP dominating whatever string of terminals forms a Noun Phrase (such as *an ice-cream*). Worse still, there could be an arbitrary number of NPs dominating *an ice-cream*. This causes computational nightmares; suppose a parser had to build all of these possibilities—it could go on forever. Fortunately, this extra layer of description is superfluous. If we know that *an ice-cream* is the Subject Noun Phrase, occupying the first two positions in the sentence, then that is all we need to know. The extra

NPs do not add anything new in the way of description, since all they can do is simply repeat the statement that there is an NP occupying the first two positions. To put the same point another way, we would expect no grammatical process to operate on the lowest NP without also affecting NP in the same way.

We can contrast this nonbranching NP with a branching one:

S→ NP VP   NP→NP PP
PP→ Prep NP   NP→ Det Noun

Here we can have an NP dominating the branching structure NP–PP. We need both NPs, because they describe different phrases in the input. For example, the lowest NP might dominate *an ice-cream* and the higher NP might dominate *an ice-cream with raspberry toppings*—plainly different things, to an ice-cream aficionado.

Note that like the FTN case, ambiguity is a property of a grammar. However, there are *two* ways that ambiguity can arise in a context-free system. First, just as with finite-state systems, we can have *lexical* or *category ambiguity*: one and the same word can be analyzed in two different ways, for example, as either a Noun or a Verb. For example, we could have Noun→*time* or Verb→*time*. Second, because we now have phrases, not just single words, one can sometimes analyze the same sequence of word categories in more than one way. For example, *the guy on the hill with the telescope* can be analyzed as a single Noun Phrase as either,

[the guy [on the hill [with the telescope]]]
(the hill has a telescope on it)

[the guy [on the hill]][with the telescope]
(the guy has the telescope)

The word categories are not any different in these two analyses. The only thing that changes is *how* the categories are stitched together. In simple cases, one can easily spot this kind of *structural ambiguity* in a context-free system. It often shows up when two different rules share a common phrase boundary. Here's an example:

VP→   Verb   NP   PP
NP→   Det   Noun   PP

The NP and VP share a PP in common: the rightmost part of an NP can be a Prepositional Phrase, which can also appear just after an NP in a VP. We might not know whether the PP belongs to the NP or the VP—an ambiguity.

We can now define $\overline{\mathrm{X}}$ context-free grammars.

Definition 5: (Kornai, 1983) An $\overline{\mathrm{X}}^n$ *grammar* (an X *n-bar* grammar) is a context free grammar $G = (N, T, P, Start)$ satisfying in addition the following 3 constraints:

$$\begin{aligned}
&\text{(lexicality)} && N = \{X^i | 1 \le i \le n, X \in T\} \\
&\text{(centrality)} && Start = X^n, X \in T \\
&\text{(succession)} && \text{rules in } P \text{ have the form } X^i \to \alpha X^{i-1}\beta \\
&\text{(uniformity)} && \text{where } \alpha \text{ and } \beta \text{ are possibly empty strings over} \\
&\text{(maximality)} && \text{the set of "maximal categories (projections)." } N_M = \{X^n | X \in T\}
\end{aligned}$$

Notation: The superscripts are called *bar levels* and are sometimes notated $\overline{X}$, $\overline{\overline{X}}$, etc. The items α (*not* a single category, note) are called the *specifiers* of a phrase, while β comprise the *complements* of a phrase. The most natural example is that of a verb phrase: the objects of the verb phrase are its COMPlements while its Specifier might be the Subject Noun Phrase.

Example. Suppose $n = 2$. Then the $\overline{X}^2$ grammar for noun expansions could include the rules:

$$N^2 \to \text{Determiner} N^1 P^2$$
$$N^1 \to N^0 (= \text{noun})$$

The X are lexical items (word categories). The X^i are called *projections* of X, and X is the *head* of X^i. Note that the $\overline{X}$ definition enforces the constraint that all maximal projections, or full phrases like noun phrases (NPs), verb phrases (VPs), etc., are maximal projection phrases, uniformly (this restriction is relaxed in some accounts).

It is not hard to show that the number of bar levels doesn't really matter to the *language* that is generated, in that given, say, an X^3 grammar, we can always find an X^1 grammar that generates the same language by simply substituting for X^3 and X^2. Conversely, given any X^n grammar, we can find an X^{n+1} grammar generating the same language—for every rule $\alpha \to \beta$ in the old grammar, the the new grammar just adds rules that has the same rules with the bar level incremented by 1, plus a new rule $X^1 \to X^0$, for every lexical item X.

Unfortunately, Kornai's definition itself is flawed given current views. Let us see what sort of $\overline{X}$ theory people actually use nowadays. We can restrict the number of bar levels to 2, make the rules binary branching, and add rules of the form $X^i \to \alpha X^i$ or $X^i \to X^i\beta$. The binary branching requirement is quite natural for some systems of semantic interpretation, a matter to which we will return. On this most recent view, the central assumption of the $\overline{X}$ model is that a word category X can function as the head of a phrase and be projected to a corresponding phrasal category XP by (optionally) adding one of three kinds of modifiers: X can be projected into an $\overline{X}$ by adding a **complement**; the $\overline{X}$ can be recursively turned into another $\overline{X}$ by the addition of a modifying adjunct, and this $\overline{X}$ can be projected into an XP ($\overline{\overline{X}}$) by adding a **specifier**. In other words, the basic skeleton structure for phrases in English looks like this:

(2) $[_{\overline{\overline{X}}}$ specifier $[_{\overline{X}}$ adjunct $[_{\overline{X}} [_X$ head] complement]]]

Moreover, all word categories, not just A, N, P, V, are projectable in this way. The so-called *functional* categories D(eterminer), I(nflection), and C(omplementizer) (*that*, *for*, etc.) are similarly projectable. Taking an example from Radford (1990), consider this example and see how it is formed:

(3) They will each bitterly criticize the other

The Verb *criticize* is projected in a $\overline{\text{V}}$ by adding the noun complement *the other*; the $\overline{\text{V}}$ *criticize the other* is projected into another $\overline{\text{V}}$ by adding the adverbial adjunct *bitterly*; the new $\overline{\text{V}}$ is projected into a full VP by adding the specifier *each*. Continuing, consider the modal auxiliary verb *will*. Since it is **I**nflected for tense, we can assign it the category **I**. I is projected into $\bar{\text{I}}$ or **IP** by the addition of the verb phrase complement [*each bitterly criticize the other*]. This is projected into another $\bar{\text{I}}$ by adding the adverbial adjunct *probably*; this finally forms an IP by adding the specifier (subject) *they*, so we get a full sentence. Thus IP=S(entence) phrase, in this case.

Similarly, a phrase like *that you should be tired* is composed of the Complementizer word *that*, which heads up what is now called a **Complement Phrase** or **CP** (and was sometimes called an $\overline{\text{S}}$ phrase; *don't* get this confused with the *Complement* of a phrase), and the Complement IP (=S) *you should be tired*; here the specifier and adjunct are missing.

The basic configuration of Specifier-Head and Head-Complement relations may exhaust what we need to get out of local syntactic relations, so it appears. Thus the apparent tree structure is really derivative. (Recent linguistic work may show that the binary branching structure is also derivative from more basic principles.)

3   Defining the representation for hierarchical parsing

Representing the input and trees

We define the *information* needed to parse hierarchical structure in a general way, so that we can see its relationship to what we did for precedence structure and so that we can modify this information to handle even more complex parsing tasks. What we do is augment the notion of a *state* and an element or *item* of that state.

As with precedence structure, we number the input stream of tokens with indices $0, 1, 2, \ldots, n$ that corresponds to interword positions, 0 being before the first word, $i-1$ between word $i-1$ and word i.

A **state** S_i represents all the possible (dominance, precedence) relations that could be hold of a sentence after processing i words.

An **item** represents the minimal information needed to describe a element of a state. In the case of precedence relations we could encode this information as a **dotted rule** that indicated what state we were in:

$$[X \Rightarrow \alpha \bullet \beta, i]$$

where α is the string of states seen so far and β is the (predicted) string of states to come, and i is the (redundant) state set number. Note that in a linear network all structures implicitly *start* at position 0 in the input.

A dominance/precedence structure requires this additional information for its state set items, since each item describes a partially-constructed tree. We can determine this by first seeing how a completed tree could be described:

1. A **name** for the root of the tree, e.g., S, NP, VP.

2. The **left edge** of the tree, i.e., its starting position in the input. This is given by a number indexing the interword position at which we began construction of the tree.

3. The **right edge** of the tree built so far (redundant with the state set number as before).

Example. The form [NP 0 2] describes an NP spanning positions 0 through 2 of the input. In terms of dotted rules then, we need to represent partially constructed trees by augmenting the dotted rules with a third number indicating the starting position of the tree plus the (redundant) state set number indicating how far along we have progressed so far in constructing that tree:

$$[X \Rightarrow \alpha \bullet \beta, i, j]$$

This is sometimes called a **state triple**.

Another way to think of this is in theorem proving terms (an idea first developed by Minker (1972) and then independently by Kowalski (1973)): when we have a triple in the form $[X \Rightarrow \alpha \bullet \beta, i, j]$ it means that we are attempting to prove that $\alpha\beta$ spans positions i through j in the input, and that we have already proved (or found) that α exists in the input, and that β is yet to be found (hence, it is "predicted").

Note that phrase names are atomic symbols. Complex nonterminal names are possible. Then the notion of symbol equality must follow the notions of feature-path equality described earlier. (If the feature complexes are simple attribute-value pairs with no recursive structure then this test is straightforward; if not, then the tests are not even guaranteed to terminate and some kind of restriction must be placed on the system, e.g., a limit on the depth of the test.)

4   Earley's algorithm

Earley's algorithm is like the state set simulation of a nondeterministic FTN presented earlier, with the addition of a *single* new integer representing the starting point of a hierarchical phrase (since now phrases can start at any point in the input). Given input n, a series of state sets S_0, S_1, ..., S_n is built, where S_i contains all the valid items after reading i words. The algorithm as presented is a simple recognizer; as usual, parsing involves more work.

In theorem-proving terms, the Earley algorithm selects the leftmost nonterminal (phrase) in a rule as the next candidate to see if one can find a "proof" for it in the input. (By varying which nonterminal is selected, one can come up with a different strategy for parsing.)

To recognize a sentence using a context-free grammar G and Earley's algorithm:

1 Compute the initial *state set*, S_0:

    1a Put the start state, $(Start \rightarrow \bullet S, 0, 0)$, in S_0.

    1b Execute the following steps until no new state triples are added.

        1b1 Apply **complete** to S_0.

        1b2 Apply **predict** to S_0.

2 For each word w_i, $i = 1, 2, \ldots, n$, build state set S_i given state set S_{i-1}.

    2a Apply **scan** to state set S_i.

    2b Execute the following steps until no new state triples are added to state set S_i.

        2b1 Apply **complete** to S_i

        2b2 Apply **predict** to S_i

    2c If state set S_i is empty, reject the sentence; else increment i.

    2d If $i < n$ then go to Step 2a; else go to Step 3.

3 If state set n includes the accept state $(Start \rightarrow S\bullet, 0, n)$, then accept; else reject.

Defining the basic operations on items

Definition 6: *Scan:* For all states $(A \rightarrow \alpha \bullet t\beta, k, i-1)$ in state set S_{i-1}, if $w_i = t$, then add $(A \rightarrow \alpha t \bullet \beta, k, i)$ to state set S_i.

Definition 7: *Predict* (*Push*): Given a state $(A \rightarrow \alpha \bullet B\beta, k, i)$ in state set S_i, then add all states of the form $(B \rightarrow \bullet \gamma, i, i)$ to state set S_i.

Definition 8: *Complete* (*Pop*): If state set S_i contains the triple $(B \rightarrow \gamma \bullet, k, i)$, then, for all rules in state set k of the form, $(A \rightarrow \alpha \bullet B\beta, l, k)$, add $(A \rightarrow \alpha B \bullet \beta, l, i)$ to state set S_i. (If the return value is empty, then do nothing.)

5   Comparison of FTN and Earley state set parsing

The FTN and Earley parsers are almost identical in terms of representations and algorithmic structure. Both construct a sequence of state sets S_0, S_1, ..., S_n. Both algorithms consist of three parts: an initialization stage; a loop stage, and an acceptance stage. The only difference is that since the Earley parser must handle an expanded notion of an *item* (it is now a partial

tree rather than a partial linear sequence), one must add a single new integer index to mark the return address in hierarchical structure.

Note that *prediction* and *completion* both act like ϵ-transitions: they spark parser operations without consuming any input; hence, one must close each state set construction under these operations (= we must add *all* states we can reach after reading i words, including those reached under ϵ-transitions.)

Question: where is the stack in the Earley algorithm? (Since we need a stack for return pointers.)

	FTN Parser	Earley Parser
Initialize:	Compute initial state set S0 1. S0← q0 2. S0← eta-closure (S0) q0= [Start→•S, 0] eta-closure= transitive closure of jump arcs	Compute initial state set S0 1. S0←q0 2. S0← eta-closure (S0) q0= [Start→•S, 0, 0] eta-closure= trans. closure of Predict and Complete

Loop:	Compute Si from Si-1
	For each word, wi, 1=1,...,n
	  Si← ∪d(q, wi)
	    q∈Si-1
	
	  Si← e-closure(Si)
		Compute Si from Si-1
		For each word, wi, 1=1,...,n
		  Si← ∪d(q, wi)
		    q∈Si-1
		    = SCAN(Si-1)
		    q=item
		  Si← e-closure(Si)
		  e-closure=
		    closure(PREDICT,
		    COMPLETE)

Final:	Accept/reject:
	If qf ∈ Sn then accept;
	else reject
	
	  qf= [Start→S•, 0]
		Accept/reject:
		If qf∈ Sn then accept;
		else reject
		
		  qf= [Start→S•, 0, n]

6 A simple example of the algorithm in action

Let's now see how this works with a simple grammar and then examine how parses may be retrieved. There have been several schemes proposed for parse storage and retrieval.

Here is a simple grammar plus an example parse for *John ate ice-cream on the table* (ambiguous as to the placement of the Prepositional Phrase *on the table*).

$$
\begin{array}{ll}
\text{Start} \rightarrow \text{S} & \text{S} \rightarrow \text{NP VP} \\
\text{NP} \rightarrow \text{Name} & \text{NP} \rightarrow \text{Det Noun} \\
\text{NP} \rightarrow \text{Name PP} & \text{PP} \rightarrow \text{Prep NP} \\
\text{VP} \rightarrow \text{V NP} & \text{VP} \rightarrow \text{V NP PP} \\
\text{V} \rightarrow \text{ate} & \text{Noun} \rightarrow \text{ice-cream} \\
\text{Name} \rightarrow \text{John} & \text{Name} \rightarrow \text{ice-cream} \\
\text{Noun} \rightarrow \text{table} & \text{Det} \rightarrow \text{the} \\
\text{Prep} \rightarrow \text{on} &
\end{array}
$$

Let's follow how this parse works using Earley's algorithm and the parser used in laboratory 2. (The headings and running count of state numbers aren't supplied by the parser. Also note that `Start` is replaced by `*DO*`. Some additional duplicated states that are printed during tracing have been removed for clarity, and comments added.)

```
(in-package 'gpsg)
(remove-rule-set 'testrules)
(remove-rule-set 'testdict)
(add-rule-set 'testrules 'CFG)
(add-rule-list 'testrules
               '((S ==> NP VP)
                 (NP ==> name)
                 (NP ==> Name PP)
                 (VP ==> V NP)
                 (NP ==> Det Noun)
                 (PP ==> Prep NP)
                 (VP ==> V NP PP)))


(add-rule-set 'testdict 'DICTIONARY)
(add-rule-list 'testdict
               '((ate V)
                 (John Name)
                 (table Noun)
                 (ice-cream Noun)
                 (ice-cream Name)
                 (on Prep)
                 (the Det)))

(create-cfg-table 'testg 'testrules 's 0)


? (pprint (p "john ate ice-cream on the table"
          :grammar 'testg :dictionary 'testdict :print-states t))
```

```
State set     Return ptr     Dotted rule
(nothing)
0             0              *DO* ==> . S $          (1)  (start state)
0             0              S ==> . NP VP           (2)  (predict from 1)
0             0              NP ==> . NAME           (3)  (predict from 2)
0             0              NP ==> . NAME PP        (4)  (predict from 2)
0             0              NP ==> . DET NOUN       (5)  (predict from 2)


John [Name]
1             0              NP ==> NAME .           (6)  (scan over 3)
1             0              NP ==> NAME . PP        (7)  (scan over 4)
1             0              S ==> NP . VP           (8)  (complete 6 to 2)
1             1              PP ==> . PREP NP        (9)  (predict from 7)
1             1              VP ==> . V NP           (10) (predict from 8)
1             1              VP ==> . V NP PP        (11) (predict from 8)


ate [V]
2             1              VP ==> V . NP           (12) (scan over 10)
2             1              VP ==> V . NP PP        (13) (scan over 11)
2             2              NP ==> . NAME           (14) (predict from 12/13)
2             2              NP ==> . NAME PP        (15) (predict from 12/13)
2             2              NP ==> . DET NOUN       (16) (predict from 12/13)


ice-cream [Name, Noun]
3             2              NP ==> NAME .           (17) (scan over 14)
3             2              NP ==> NAME . PP        (18) (scan over 15)
3             1              VP ==> V NP . PP        (19) (complete 17 to 13)
3             1              VP ==> V NP .           (20) (complete 17 to 12)
3             3              PP ==> . PREP NP        (21) (predict from 18/19)
3             0              S ==> NP VP .           (22) (complete 20 to 8)
3             0              *DO* ==> S . $          (23) (complete 8 to 1)


on [Prep]
4             3              PP ==> PREP . NP        (24) (scan over 21)
4             4              NP ==> . NAME           (25) (predict from 24)
4             4              NP ==> . NAME PP        (26) (predict from 24)
4             4              NP ==> . DET NOUN       (27) (predict from 24)


the [Det]
5             4              NP ==> DET . NOUN       (28) (scan over 27)


table [Noun]
6             4              NP ==> DET NOUN .       (29) (scan over 28)
6             3              PP ==> PREP NP .        (30) (complete 29 to 24)
6             1              VP ==> V NP PP .        (31) (complete 24 to 19)
6             2              NP ==> NAME PP .         (32) (complete 24 to 18)
6             0              S ==> NP VP .           (33) (complete 8 to 1)
6             0              *DO*  ==> S .           (34) (complete 1) [parse 1]
6             1              VP ==> V NP .           (35) (complete 18 to 12)
6             0              S ==> NP VP .           (36) (complete 12 to 1) = 33
```
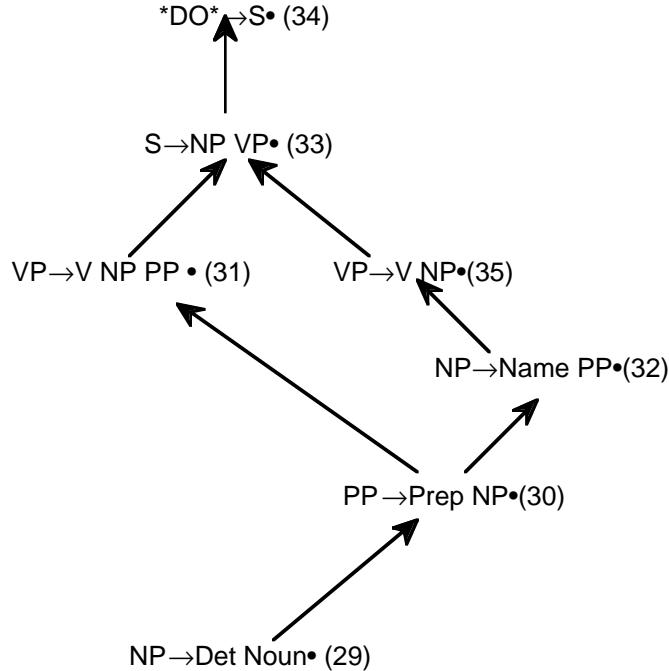
Figure 3: Distinct parses lead to distinct state triple paths in the Earley algorithm

```
6              0            *DO* ==> S .        (37)  (complete 1) = 34 [parse 2]
6              1            VP ==> V NP . PP    (38)  (complete 18 to 13)
6              6            PP ==> . PREP NP    (39)  (predict from 38)

((START
  (S (NP (NAME JOHN))
   (VP (V ATE) (NP (NAME ICE-CREAM))
    (PP (PREP ON) (NP (DET THE) (NOUN TABLE))))))
 (START
  (S (NP (NAME JOHN))
   (VP (V ATE)
    (NP (NAME ICE-CREAM) (PP (PREP ON) (NP (DET THE) (NOUN TABLE)))))))))
```

7   Time complexity of the Earley algorithm

The worst case time complexity of the Earley algorithm is dominated by the time to construct the state sets. This in turn is decomposed into the time to process a single item in a state set times the maximum number of items in a state set (assuming no duplicates; thus, we are assuming some implementation that allows us to quickly check for duplicate states in a state

set). In the worst case, the maximum number of distinct items is the maximum number of dotted rules times the maximum number of distinct return values, or $|G| \cdot n$. The time to process a single item can be found by considering separately the `scan`, `predict` and `complete` actions. `Scan` and `predict` are effectively constant time (we can build in advance all the possible single next-state transitions, given a possible category). The `complete` action could force the algorithm to advance the dot in all the items in a state set, which from the previous calculation, could be $|G| \cdot n$ items, hence proportional to that much time. We can combine the values as shown below to get an upper bound on the execution time, assuming that the primitive operations of our computer allow us to maintain lists without duplicates without any additional overhead (say, by using bit-vectors; if this is not done, then searching through or ordering the list of states could add in another $|G|$ factor.). Note as before that grammar size (measure by the total number of symbols in the rule system) is an important component to this bound; more so than the input sentence length, as you will see in Laboratory 2.

If there is no ambiguity, then this worst case does not arise (why?). The parse is then linear time (why?). If there is only a *finite* ambiguity in the grammar (at each step, there are only a finite, bounded in advance number of ambiguous attachment possibilities) then the worst case time is proportional to n^2.

Maximum number of state sets   X   Maximum time to build ONE state set

X

Maximum number of
items

Maximum time
to process ONE item

X

Maximum possible number
of items=

[maximum number of dotted rules X maximum number of distinct return values]