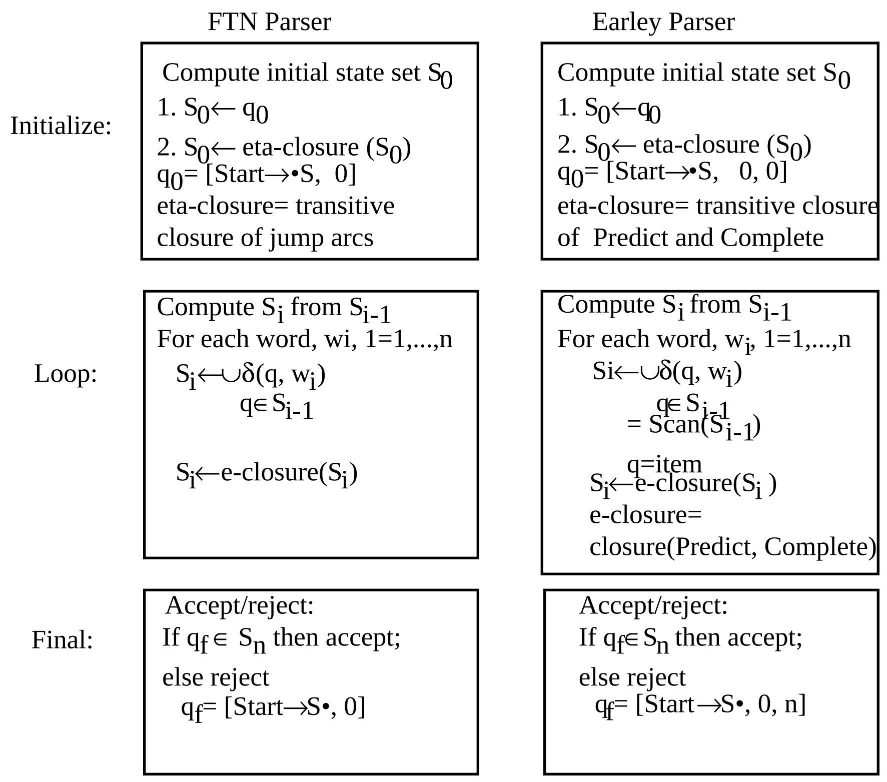
Massachusetts Institute of Technology
6.863J/9.611J, Natural Language Processing, Spring, 2001
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences

Handout 8: Computation & Hierarchical parsing II

1 FTN parsing redux



	FTN Parser	Earley Parser
Initialize:	Compute initial state set S_0 1. $S_0 \leftarrow q_0$ 2. $S_0 \leftarrow$ eta-closure (S_0) q_0= [Start→•S, 0] eta-closure= transitive closure of jump arcs	Compute initial state set S_0 1. $S_0 \leftarrow q_0$ 2. $S_0 \leftarrow$ eta-closure (S_0) q_0= [Start→•S, 0, 0] eta-closure= transitive closure of Predict and Complete
Loop:	Compute S_i from S_{i-1} For each word, wi, 1=1,...,n $S_i \leftarrow \cup_{q \in S_{i-1}} \delta(q, w_i)$ $S_i \leftarrow$ e-closure(S_i)	Compute S_i from S_{i-1} For each word, w_i, 1=1,...,n $Si \leftarrow \cup_{q \in S_{i-1}} \delta(q, w_i)$ = $\mathrm{Scan}_{q=\text{item}}(S_{i-1})$ $S_i \leftarrow$ e-closure(S_i) e-closure= closure(Predict, Complete)

Final:	Accept/reject:
	If $q_f \in S_n$ then accept;
	else reject
	q_f= [Start→S•, 0]
		Accept/reject:
		If $q_f \in S_n$ then accept;
		else reject
		q_f= [Start→S•, 0, n]

2 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). Note that with simple *linear* concatenation this information is implicitly encoded via the word position we are at. The *stopping* or *end* point of a phrase will be encoded by the word position. To proceed, 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 1: *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.

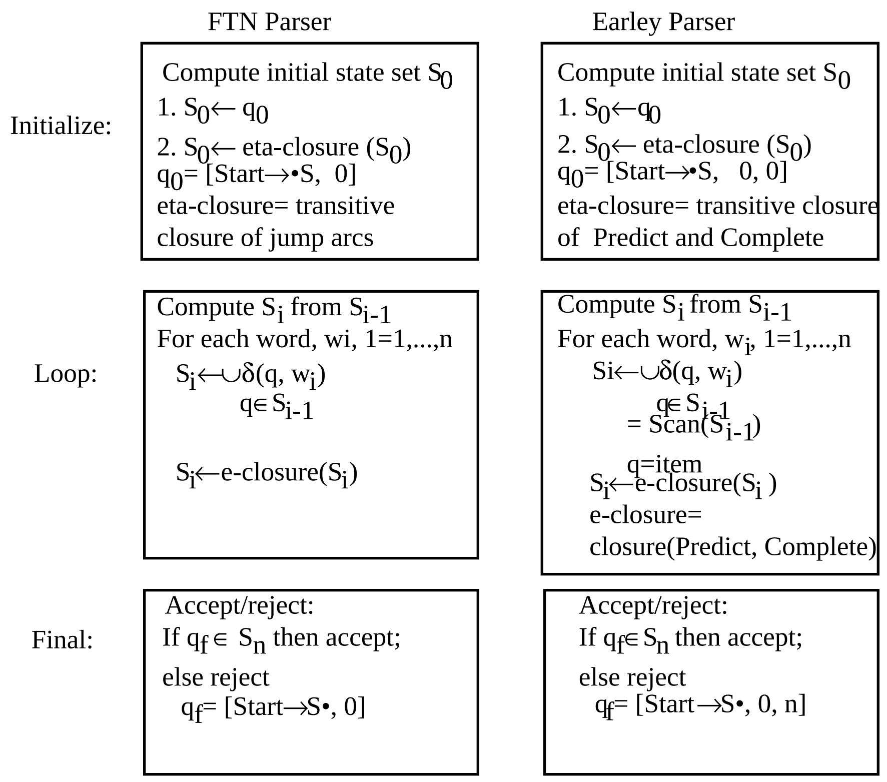
Definition 2: *Predict* (*Push*): Given a state $(A \rightarrow \alpha \bullet B\beta, k, i)$ in state set S_i, then add all states of the form $(B \rightarrow \bullet\gamma, i, i)$ to state set S_i.

Definition 3: *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.)

3 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 S_0 1. $S_0 \leftarrow q_0$ 2. $S_0 \leftarrow$ eta-closure (S_0) q_0= [Start→•S, 0] eta-closure= transitive closure of jump arcs	Compute initial state set S_0 1. $S_0 \leftarrow q_0$ 2. $S_0 \leftarrow$ eta-closure (S_0) q_0= [Start→•S, 0, 0] eta-closure= transitive closure of Predict and Complete
Loop:	Compute S_i from S_{i-1} For each word, wi, 1=1,...,n $S_i \leftarrow \cup_{q \in S_{i-1}} \delta(q, w_i)$ $S_i \leftarrow$ e-closure(S_i)	Compute S_i from S_{i-1} For each word, w_i, 1=1,...,n $Si \leftarrow \cup_{q \in S_{i-1}} \delta(q, w_i)$ = $\text{Scan}_{q=\text{item}}(S_{i-1})$ $S_i \leftarrow$ e-closure(S_i) e-closure= closure(Predict, Complete)

Final:	Accept/reject:
	If $q_f \in S_n$ then accept;
	else reject
	q_f= [Start→S•, 0]
		Accept/reject:
		If $q_f \in S_n$ then accept;
		else reject
		q_f= [Start→S•, 0, n]

4 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*).

Start→S	S→NP VP
NP→Name	NP→Det Noun
NP→Name PP	PP→ Prep NP
VP→V NP	VP→V NP PP
V→ate	Noun→ice-cream
Name→John	Name→ice-cream
Noun→table	Det→the
Prep→on	

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
```

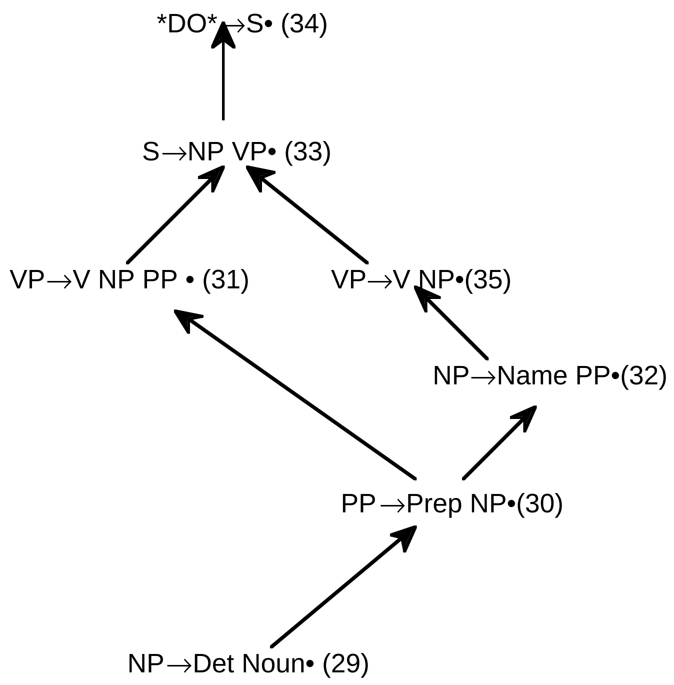


Figure 1: Distinct parses lead to distinct state triple paths in the Earley algorithm

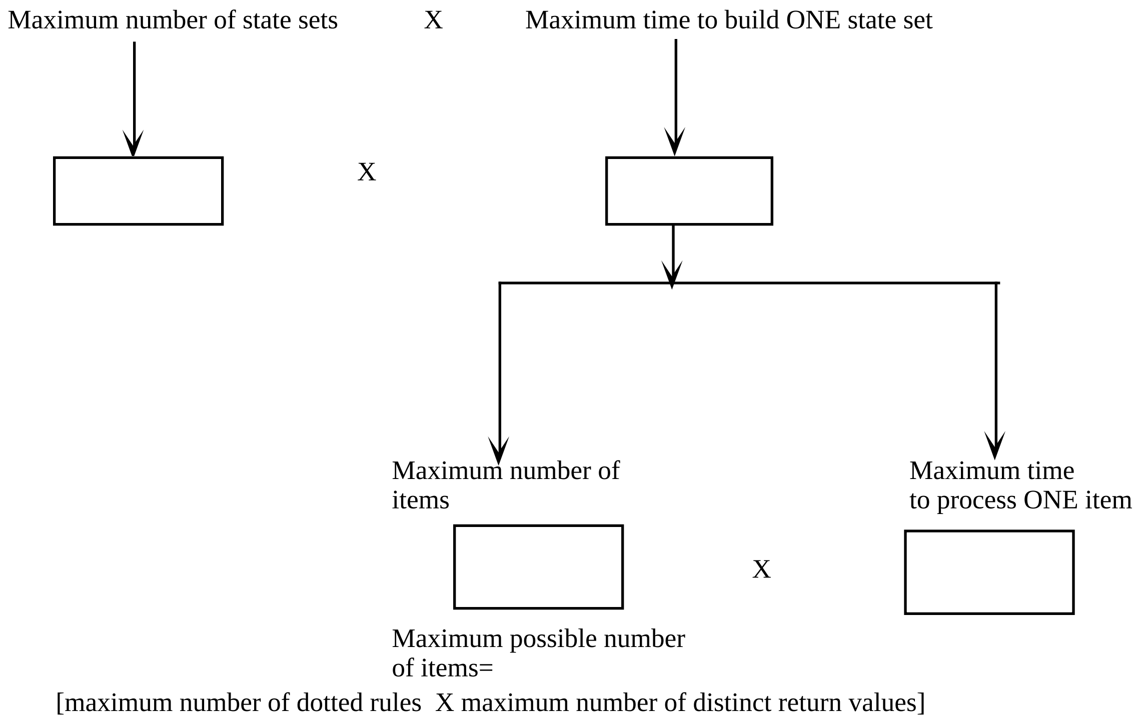
```
6           0            *DO* ==> S .          (37)  (complete 1) = 34 [parse 2]
6           1            VP ==> V NP . PP      (38)  (complete 18 to 13)
6           6            PP ==> . PREP NP      (39)  (predict from 38)

((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))))))))
```

5 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]