Lecture 3

Tabular Parsing

Last modified 2016/05/18

Tabular parsing is an application of *dynamic programming* (see Cormen et al. 2009, Kleinberg and Tardos 2006) to the problems of recognition and parsing in order to achieve polynomial time complexity. In this lecture, we first see why polynomial-time recognition is possible in abstract terms, using the notion of an *alternating machine*. Next, we use *Datalog* to formalize tabular deductive parsing. To understand tabular context-free recognition/parsing, the generality of Datalog is an overkill, but it will pay dividends when we turn to recognition/parsing for grammar formalisms more powerful than context-free grammars. Finally, we look at tabular recognition/parsing based on pushdown automata.[1]

Complexity-theoretic considerations

We start with some notations and terminology we have not introduced so far. Let us write $\alpha \underset{\text{lm}}{\Rightarrow} \beta$ (resp. $\alpha \underset{\text{rm}}{\Rightarrow} \beta$) when β is the result of rewriting the leftmost (resp. rightmost) nonterminal in α according to some production in the given context-free grammar. A derivation

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$$

is a *leftmost* (resp. *rightmost*) derivation if $\alpha_i \underset{\text{lm}}{\Rightarrow} \alpha_{i+1}$ (resp. $\alpha_i \underset{\text{rm}}{\Rightarrow} \alpha_{i+1}$) for $i = 1, \dots, n-1$. A string $\alpha \in (N \cup \Sigma)^*$ is a *left* (resp. *right*) *sentential form* if $S \underset{\text{lm}}{\Rightarrow}^* \alpha$ (respectively, if $S \underset{\text{rm}}{\Rightarrow}^* \alpha$).

A configuration of a top-down recognizer $(\alpha, a_{i+1} \dots a_n)$ on input $w = a_1 \dots a_n$ (reachable from the initial configuration) corresponds to a left sentential form $a_1 \dots a_i \alpha$. The accepting configuration is reachable from such a configuration if and only if $\alpha \Rightarrow^* a_{i+1} \dots a_n$. Thus, we can think of a configuration

[1] I'm grateful to Robert Glück for helpful comments on an earlier version of this lecture.

$(\alpha, a_{i+1} \dots a_n)$ as representing the problem "$\alpha \Rightarrow^* a_{i+1} \dots a_n$?" When in configuration $(A\beta, a_{i+1} \dots a_n)$, what the top-down recognizer tries to do next is in effect to reduce the problem

$$A\beta \Rightarrow^* a_{i+1} \dots a_n?$$

to the "subproblems"

$$\begin{gathered} \alpha_1\beta \Rightarrow^* a_{i+1} \dots a_n? \\ \vdots \\ \alpha_k\beta \Rightarrow^* a_{i+1} \dots a_n? \end{gathered}$$

where $A \to \alpha_1, \dots, A \to \alpha_k$ is the list of productions with A on the left-hand side. The first problem is equivalent to the disjunction of these subproblems. The top-down recognizer nondeterministically picks one of these subproblems and proceeds to the next step.

The input part of a configuration $(\alpha, a_{i+1} \dots a_n)$ can be represented by the integer i, which only takes $O(\log n)$ space (assuming that the entire input is stored externally), but the stack depth $|\alpha|$ in an accepting computation can be as large as $n - i$, even when the grammar has no ε-productions,[2] so the amount of space required to represent a configuration $(\alpha, a_{i+1} \dots a_n)$ is $\Theta(n)$ (i.e., roughly proportional to n) in the worst case.[3] So we have a nondeterministic algorithm that can be implemented on a Turing machine operating in space $O(n)$. From this information alone, all we can say is that the deterministic simulation can be done in exponential time. In fact, the number of left sentential forms consistent with some prefix of the input is exponential in the worst case (even for grammars with no ε- or unit productions), so the naive deterministic simulation must take exponential time (Aho and Ullman 1972). The situation with the bottom-up recognizer is completely analogous.

Exercise 3.1. Show that the backtracking top-down recognizer (with lookahead) takes exponential time in the worst case on the following CFG:

$$\begin{gathered} S \to a\,S\,A \mid a\,S\,B \mid \# \\ A \to a \qquad B \to b \end{gathered}$$

Exercise 3.2. Show that the backtracking bottom-up recognizer takes exponential time in the worst case on the following CFG:

$$\begin{gathered} S \to A\,S\,a \mid B\,S\,b \mid \# \\ A \to a \qquad B \to a \end{gathered}$$

[2]In the presence of ε-productions, the stack depth can become arbitrarily large in general.

[3]Here we are assuming that the length n of the input is available to the algorithm from the beginning, so that a configuration $(\alpha, a_{i+1} \dots a_n)$ with $|\alpha| > n - i$ need not be considered by the algorithm.

There is a very different way of dealing with the problem "$\alpha \Rightarrow^* \beta$?", however, which makes use of an important property of the derivation relation of context-free grammars:

$$X_1 \ldots X_n \Rightarrow^* \beta \quad \text{iff} \quad X_1 \Rightarrow^* \beta_1 \wedge \cdots \wedge X_n \Rightarrow^* \beta_n$$
$$\text{for some } \beta_1, \ldots, \beta_n \text{ such that } \beta = \beta_1 \ldots \beta_n.$$

Using this property, we can reduce the problem "$S \Rightarrow^* a_1 \ldots a_n$?" to a set of independent problems of the form

$$X \Rightarrow^* a_{i+1} \ldots a_j?,$$

where $0 \leq i \leq j \leq n$. For example, suppose that the grammar has two rules for expanding S:

$$S \to AB \qquad S \to CDE$$

Take $n = 2$. Then we can reduce the problem "$S \Rightarrow^* a_1 a_2$" to the following set of problems:

$$A \Rightarrow^* \varepsilon \wedge B \Rightarrow^* a_1 a_2?$$
$$A \Rightarrow^* a_1 \wedge B \Rightarrow^* a_2?$$
$$A \Rightarrow^* a_1 a_2 \wedge B \Rightarrow^* \varepsilon?$$
$$C \Rightarrow^* \varepsilon \wedge D \Rightarrow^* \varepsilon \wedge E \Rightarrow^* a_1 a_2?$$
$$C \Rightarrow^* \varepsilon \wedge D \Rightarrow^* a_1 \wedge E \Rightarrow^* a_2?$$
$$C \Rightarrow^* \varepsilon \wedge D \Rightarrow^* a_1 a_2 \wedge E \Rightarrow^* \varepsilon?$$
$$C \Rightarrow^* a_1 \wedge D \Rightarrow^* \varepsilon \wedge E \Rightarrow^* a_2?$$
$$C \Rightarrow^* a_1 \wedge D \Rightarrow^* a_2 \wedge E \Rightarrow^* \varepsilon?$$
$$C \Rightarrow^* a_1 a_2 \wedge D \Rightarrow^* \varepsilon \wedge E \Rightarrow^* \varepsilon?$$

The disjunction of these conjunctions is equivalent to "$S \Rightarrow^* a_1 a_2$?" Each conjunction in turn reduces to its conjuncts in the obvious way.

In general, a problem "$X \Rightarrow^* a_{i+1} \ldots a_j$?" reduces to a set of subproblems of the same form; the original problem is equivalent to a disjuntion of conjunctions made up of these subproblems. Thus, the dependence among different problems can be depicted in the form of an *AND/OR tree* (see Nilsson 1982), as in Figure 3.1. In a tree like this, nodes that have an arc through their outgoing edges are *AND nodes*; other nodes are *OR nodes*.[4] An AND node indicates that all its successor nodes (i.e., children) must be established, whereas an OR node only requires one of its successor nodes to be established.

[4]This usage of the terms "AND node" and "OR node" differs from Nilsson's (1982), but seems to be fairly standard (Simon and Lee 1971, Hall 1973). If we view an AND/OR graph as a Boolean circuit, AND nodes and OR nodes according to this usage correspond to *AND gates* and *OR gates*.

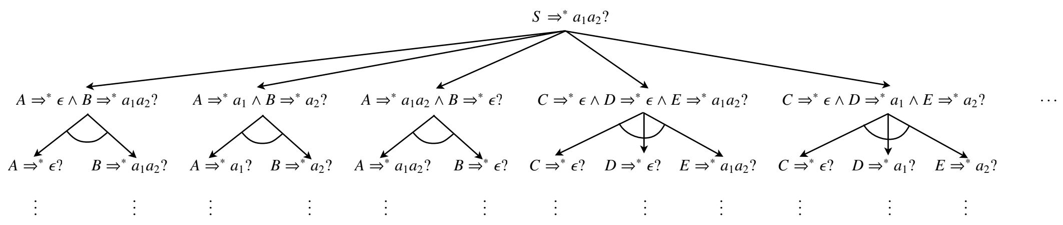

$S \Rightarrow^* a_1a_2?$
$A \Rightarrow^* \epsilon \wedge B \Rightarrow^* a_1a_2?$
$A \Rightarrow^* a_1 \wedge B \Rightarrow^* a_2?$
$A \Rightarrow^* a_1a_2 \wedge B \Rightarrow^* \epsilon?$
$C \Rightarrow^* \epsilon \wedge D \Rightarrow^* \epsilon \wedge E \Rightarrow^* a_1a_2?$
$C \Rightarrow^* \epsilon \wedge D \Rightarrow^* a_1 \wedge E \Rightarrow^* a_2?$
$\cdots$
$A \Rightarrow^* \epsilon?$
$B \Rightarrow^* a_1a_2?$
$A \Rightarrow^* a_1?$
$B \Rightarrow^* a_2?$
$A \Rightarrow^* a_1a_2?$
$B \Rightarrow^* \epsilon?$
$C \Rightarrow^* \epsilon?$
$D \Rightarrow^* \epsilon?$
$E \Rightarrow^* a_1a_2?$
$C \Rightarrow^* \epsilon?$
$D \Rightarrow^* a_1?$
$E \Rightarrow^* a_2?$

Figure 3.1: An AND/OR tree.

These considerations lead to an *alternating* machine approach to solving the problem "$S \Rightarrow^* a_1 \dots a_n$?" Alternating machines are a generalization of nondeterministic machines. Recall that in a nondeterministic machine (e.g., finite automaton, pushdown automaton, or Turing machine), if an accepting configuration is to be reachable from a configuration having more than one successor configuration (i.e., configuration that can be reached in a single step), it must be reachable from at least one of those successor configurations (unless the given configuration is already accepting). An alternating machine has two types of states: *existential* states and *universal* states. Existential states are just like states of a nondeterministic machine; if a machine's configuration is in an existential state, it is like the disjunction of its successor configurations. If a machine's configuration is in a universal state, in contrast, it is like the *conjunction* of its successor configurations: it requires that *all* its successor configurations lead to acceptance.

You can think of an alternating machine as a kind of two-person game. In an existential state, you get to make the next move; in a universal state, it is your opponent's turn to make a move. The input is accepted if you have a winning strategy in the sense that no matter how your opponent plays, you can eventually reach an accepting configuration by following your strategy.

It is not hard to see that the problem "$S \Rightarrow^* a_1 \dots a_n$?" is solvable by an alternating Turing machine operating in space $O(\log n)$. Since a substring $a_{i+1} \dots a_j$ of the input string can be represented by a pair (i, j) of integers in binary, representing a problem instance $X \Rightarrow^* a_{i+1} \dots a_j$ requires only $O(\log n)$ space. Now it is a fundamental result in computational complexity theory that the problems solvable by alternating Turing machines operating in logarithmic space are precisely those solvable by *deterministic* Turing machines operating in *polynomial time* (Chandra et al. 1980; see Sipser 2012).[5] Therefore, there is a polynomial time algorithm for solving the problem.

[5]In general, for $f(n) \geq \log n$, it holds that $\mathrm{ASPACE}(f(n)) = \mathrm{TIME}(2^{O(f(n))})$. (Here, $\mathrm{ASPACE}(f(n))$ refers to the class of problems solvable by an alternating Turing machine operating in space $O(f(n))$, and $\mathrm{TIME}(g(n))$ refers to the class of problems solvable by a deterministic Turing machine operating in time $O(g(n))$.)

We will make use of the analysis in terms of logspace-bounded alternating Turing machines in later lectures when we look at recognition/parsing for more powerful grammar formalisms.

Deduction systems for parsing

Let us approach the problem of recognition/parsing on a more concrete level. One of the keys to gaining efficiency is to represent a problem instance "$X \Rightarrow^* a_{i+1} \dots a_j$?" by a triple (X, i, j). Let us now view grammar symbols X (nonterminals or terminals) as binary predicates over input positions. Thus, an atomic formula $X(i, j)$ holds if and only if $X \Rightarrow^* a_{i+1} \dots a_j$, where a_k is the k-th symbol of the input. If we do this, a production of a context-free grammar like $S \to AB$ can be expressed as a *definite clause*:[6]

$$S(\boldsymbol{i}, \boldsymbol{k}) \leftarrow A(\boldsymbol{i}, \boldsymbol{j}),\ B(\boldsymbol{j}, \boldsymbol{k}).$$

This is a logic programming way of representing the first-order formula

$$\forall i \forall j \forall k((A(i, j) \wedge B(j, k)) \to S(i, k)),$$

or equivalently,

$$\forall i \forall k(\exists j(A(i, j) \wedge B(j, k)) \to S(i, k)).$$

The context-free grammar in Figure 3.2 can be represented as a *logic program* (i.e., a set of definite clauses), as in Figure 3.3. Definite clauses in a program are also called *rules*. In a rule $P \leftarrow Q_1, \dots, Q_n$, the atomic formula P is called the *head* of the rule, and $Q_1, \dots, Q_n$ are called the *body*. Since the program in Figure 3.3 does not contain any function symbols, it belongs to the subset of logic programming known as *Datalog* (see Ullman 1989b or Abiteboul et al. 1995).[7] Every context-free grammar can be expressed as a Datalog program in this way.[8]

[6]A *clause* is a disjunction of *literals*, i.e., atomic formulas (*positive literals*) or their negations (*negative literals*). A *Horn clause* is a clause which has at most one positive literal. A *definite clause* is a clause which has exactly one positive literal. A definite clause $P \vee \neg Q_1 \vee \dots \vee \neg Q_n$ is equivalent to an implication $(Q_1 \wedge \dots \wedge Q_n) \to P$. A definite clause with no negative literal is called a *fact*. A clause is *ground* if it contains no variables.

Since we treat definite clauses as syntactic objects, we use boldface italic letters for variables in definite clauses, to distinguish them from metavariables (i.e., variables used in the metalanguage).

[7]In Datalog, it is usually required that the variables in the head of a rule all appear in the body. This is not an essential restriction.

[8]A production $A \to X_1 \dots X_n$ of a CFG $G = (N, \Sigma, P, S)$, where $n \geq 1$ and $X_i \in N \cup \Sigma$, translates into the Datalog rule $A(\boldsymbol{i}_0, \boldsymbol{i}_n) \leftarrow X_1(\boldsymbol{i}_0, \boldsymbol{i}_1), \dots, X_n(\boldsymbol{i}_{n-1}, \boldsymbol{i}_n)$. An ε-production $A \to \varepsilon$ can be represented as $A(\boldsymbol{i}, \boldsymbol{j}) \leftarrow \boldsymbol{i} = \boldsymbol{j}$. For the sake of simplicity, we avoid using equality for the moment, assuming that the grammar contains no ε-production.

S → NP VP
S → Aux NP VP
S → VP
NP → Det N1
NP → Name
NP → Pronoun
VP → V
VP → V NP
VP → V NP PP
VP → VP PP
N1 → N
N1 → Name N1
N1 → N1 PP
PP → P NP

Aux → does
Det → that | this | a | the
Name → Houston | TWA
Pronoun → I | she | me
V → book | include | prefer
N → book | flight | meal | money
P → from | to | on

Figure 3.2: A CFG.

(As we shall see, *deduction systems* (Shieber et al. 1995, Sikkel 1997) for parsing can also be expressed as Datalog programs.)

The program in Figure 3.3 is equivalent to the grammar in Figure 3.2 in the following sense: a string $a_1 \dots a_n$ is generated by the grammar in Figure 3.2 if and only if $S(0, \overline{n})$ is *derivable* from the program in Figure 3.3 together with the set of facts:

$$\{a_1(0, 1), \dots, a_n(\overline{n-1}, \overline{n})\},$$

where derivability is understood in the usual sense of first-order logic. Here, "0", "1", etc., are constant symbols. We assume that for each natural number, there is a distinct constant symbol representing it. If k is a natural number, we let "$\overline{k}$" stand for the constant symbol representing k. For example, since the string does this flight include a meal is generated by the grammar in Figure 3.2, $S(0, 6)$ is derivable from the program in Figure 3.3 together with

$$\{\mathsf{does}(0, 1), \mathsf{this}(1, 2), \mathsf{flight}(2, 3), \mathsf{include}(3, 4), \mathsf{a}(4, 5), \mathsf{meal}(5, 6)\}.$$

In the Datalog parlance, predicates that are defined in the program are called *intensional predicates*, and predicates whose extension is fixed by a set of facts external to the program (known as an *extensional database*) are called *extensional predicates*. When a context-free grammar is expressed as a Datalog program, nonterminals of the grammar correspond to intensional predicates, and terminals to extensional predicates. The extensional database is determined by the input string.

```
S(i,k) ← NP(i,j), VP(j,k).
S(i,l) ← Aux(i,j), NP(j,k), VP(k,l).
S(i,j) ← VP(i,j).
NP(i,k) ← Det(i,j), N1(j,k).
NP(i,j) ← Name(i,j).
NP(i,j) ← Pronoun(i,j).
VP(i,j) ← V(i,j).
VP(i,k) ← V(i,j), NP(j,k).
VP(i,l) ← V(i,j), NP(j,k), PP(k,l).
VP(i,k) ← VP(i,j), PP(j,k).
N1(i,j) ← N(i,j).
N1(i,k) ← N1(i,j), N(j,k).
N1(i,k) ← N1(i,j), PP(j,k).
PP(i,k) ← P(i,j), NP(j,k).

Aux(i,j) ← does(i,j).
Det(i,j) ← that(i,j).
Det(i,j) ← this(i,j).
Det(i,j) ← a(i,j).
Det(i,j) ← the(i,j).
Name(i,j) ← Houston(i,j).
Name(i,j) ← TWA(i,j).
Pronoun(i,j) ← I(i,j).
Pronoun(i,j) ← she(i,j).
Pronoun(i,j) ← me(i,j).
V(i,j) ← book(i,j).
V(i,j) ← include(i,j).
V(i,j) ← prefer(i,j).
N(i,j) ← book(i,j).
N(i,j) ← flight(i,j).
N(i,j) ← meal(i,j).
N(i,j) ← money(i,j).
P(i,j) ← from(i,j).
P(i,j) ← to(i,j).
P(i,j) ← on(i,j).
```

Figure 3.3: The Datalog program representing a CFG.

Naive bottom-up evaluation There is a simple bottom-up method for computing all ground facts derivable from a Datalog program **P** and an extensional database D. This method is based on the following inference rule:

$$\frac{P \leftarrow Q_1, \ldots, Q_n \quad Q_1\theta \quad \ldots \quad Q_n\theta}{P\theta} \tag{3.1}$$

where θ is a *ground substitution* mapping all the variables in $Q_1, \ldots, Q_n$ to constants. (This rule is a restricted form of *resolution*.) Assuming the Datalog convention that the variables in P all appear in $Q_1, \ldots, Q_n$, $P\theta$ is a ground fact. Note that an instance of this rule is uniquely determined by a ground instance $(P \leftarrow Q_1, \ldots, Q_n)\theta$ of a clause in **P**.

Given a Datalog program **P** and an extensional database D, we define *derivation trees* (from **P**, D) inductively as follows:

1. If P is a fact in D, the tree with just one node labeled by P is a derivation tree for P.

2. If $P \leftarrow Q_1, \ldots, Q_n$ is a rule in **P** and $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are derivation trees for

$Q_1\theta, \ldots, Q_n\theta$, respectively, then

$$\begin{array}{c} P\theta \\ \diagup \quad \cdots \quad \diagdown \\ \mathcal{T}_1 \qquad \mathcal{T}_n \end{array}$$

is a derivation tree for $P\theta$.

A ground fact is derivable from $\mathbf{P} \cup D$ if and only if it has a derivation tree from $\mathbf{P}, D$. If $\mathbf{P}$ represents a CFG G and D represents a string $a_1 \ldots a_n$, then the derivation trees from $\mathbf{P} \cup D$ correspond one-to-one to the (partial) parse trees for substrings of $a_1 \ldots a_n$.

The following algorithm, called *naive bottom-up evaluation*, finds all ground facts derivable from $\mathbf{P} \cup D$.[9]

```
NAIVE($\mathbf{P}, D$)
1  $D^1 \leftarrow D$
2  $\Delta^1 \leftarrow D$
3  $i \leftarrow 1$
4  while $\Delta^i \neq \emptyset$
5       do $F^{i+1} \leftarrow \{ P\theta \mid P \leftarrow Q_1, \ldots, Q_n$ is in $\mathbf{P}$ and $Q_1\theta, \ldots, Q_n\theta \in D^i \}$
6          $\Delta^{i+1} \leftarrow F^{i+1} - D^i$
7          $D^{i+1} \leftarrow D^i \cup \Delta^{i+1}$
8          $i \leftarrow i + 1$
9  return $D^i$
```

In the algorithm, D^i is the set of facts derivable from $\mathbf{P} \cup D$ with derivation trees of height $< i$ (assuming that a tree with just one node is of height 0). A ground fact is derivable from $\mathbf{P} \cup D$ if and only if it is in the output of NAIVE($\mathbf{P}, D$). The algorithm runs in polynomial time in the size of D. To see this, note that the number of iterations of the while loop is bounded by the number of ground facts, which is $O(qn^r)$, where q is the number of predicates in $\mathbf{P}$, n is the number of constants in D, and r is the maximal arity of the predicates. Each invocation of the operation in line 5 can be performed in time $O(pn^l)$, where p is the size of $\mathbf{P}$ and l is the maximal number of variables in rules in $\mathbf{P}$. (Note that there are at most n^l ground instances of each clause in $\mathbf{P}$.) This algorithm is not particularly efficient, however, because the same facts are generated over and over again in line 5. (Since $D^i \supseteq D^{i-1}$, it is always the case that $F^{i+1} \supseteq F^i$ for $i \geq 2$.)

For example, consider the program (3.3) and input database

$$D = \{\mathsf{book}(0, 1), \mathsf{the}(1, 2), \mathsf{flight}(2, 3), \mathsf{from}(3, 4), \mathsf{Houston}(4, 5)\}.$$

[9]The pseudocode in this lecture follows the style of Cormen et al. (2001). The block structure is indicated by indentation.

We have

$$\begin{aligned}
D^1 &= D\\
D^2 &= \{\mathsf{N}(0,1), \mathsf{V}(0,1), \mathsf{Det}(1,2), \mathsf{N}(2,3), \mathsf{P}(3,4), \mathsf{Name}(4,5)\} \cup D^1\\
D^3 &= \{\mathsf{N1}(0,1), \mathsf{VP}(0,1), \mathsf{N1}(2,3), \mathsf{NP}(4,5)\} \cup D^2\\
D^4 &= \{\mathsf{NP}(1,3), \mathsf{PP}(3,5)\} \cup D^3\\
D^5 &= \{\mathsf{VP}(0,3), \mathsf{VP}(0,5), \mathsf{N1}(2,5)\} \cup D^4\\
D^6 &= \{\mathsf{S}(0,3), \mathsf{S}(0,5), \mathsf{NP}(1,5)\} \cup D^5\\
D^7 &= D^6
\end{aligned}$$

Seminaive bottom-up evaluation The following modification of naive bottom-up evaluation avoids considering the same instance of the same clause more than once.

```
SEMINAIVE($\mathbf{P}$, $D$)
 1  $D^0 \leftarrow \emptyset$
 2  $D^1 \leftarrow D$
 3  $\Delta^1 \leftarrow D$
 4  $i \leftarrow 1$
 5  while $\Delta^i \neq \emptyset$
 6      do $F^{i+1} \leftarrow \left\{ P\theta \,\middle|\, \begin{array}{l} P \leftarrow Q_1, \ldots, Q_n \text{ is in } \mathbf{P},\ 1 \leq j \leq n,\ Q_j\theta \in \Delta^i, \\ Q_1\theta, \ldots, Q_{j-1}\theta \in D^i, \text{ and } Q_{j+1}\theta, \ldots, Q_n\theta \in D^{i-1} \end{array} \right\}$
 7         $\Delta^{i+1} \leftarrow F^{i+1} - D^i$
 8         $D^{i+1} \leftarrow D^i \cup \Delta^{i+1}$
 9         $i \leftarrow i + 1$
10  return $D^i$
```

In line 6, at least one of the atoms $Q_1\theta, \ldots, Q_n\theta$ belongs to the set Δ^i of the most recently derived facts (j is the greatest index such that $Q_i\theta$ belongs to Δ^i). This guarantees that the ground instance $(P \leftarrow Q_1, \ldots, Q_n)\theta$ of $P \leftarrow Q_1, \ldots, Q_n$ has never been used at earlier stages. On the same input as before, we have

$$\begin{aligned}
F^2 &= \{\mathsf{N}(0,1), \mathsf{V}(0,1), \mathsf{Det}(1,2), \mathsf{N}(2,3), \mathsf{P}(3,4), \mathsf{Name}(4,5)\}\\
F^3 &= \{\mathsf{N1}(0,1), \mathsf{VP}(0,1), \mathsf{N1}(2,3), \mathsf{NP}(4,5)\}\\
F^4 &= \{\mathsf{NP}(1,3), \mathsf{PP}(3,5)\}\\
F^5 &= \{\mathsf{VP}(0,3), \mathsf{VP}(0,5), \mathsf{N1}(2,5)\}\\
F^6 &= \{\mathsf{S}(0,3), \mathsf{S}(0,5), \mathsf{VP}(0,5), \mathsf{NP}(1,5)\}\\
F^7 &= \{\mathsf{VP}(0,5)\}
\end{aligned}$$

The values of D^i and Δ^i are the same as before. Since $\{\mathsf{VP}(0,5)\} \subseteq D^6$, we have $\Delta^7 = \varnothing$ and $D^7 = D^6$.

We can turn this algorithm into one that outputs a representation of the set of all derivation trees from **P**, D:

```
SEMINAIVE-PARSE($\mathbf{P}$, $D$)
 1  $D^0 \leftarrow \varnothing$
 2  $D^1 \leftarrow D$
 3  $C^1 \leftarrow D$
 4  $\Delta^1 \leftarrow D$
 5  $i \leftarrow 1$
 6  while $\Delta^i \neq \varnothing$
 7      do $F^{i+1} \leftarrow \left\{ P\theta \;\middle|\; \begin{array}{l} P \leftarrow Q_1,\dots,Q_n \text{ is in } \mathbf{P},\ 1 \le j \le n,\ Q_j\theta \in \Delta^i, \\ Q_1\theta,\dots,Q_{j-1}\theta \in D^i, \text{ and } Q_{j+1}\theta,\dots,Q_n\theta \in D^{i-1} \end{array} \right\}$
 8         $C^{i+1} \leftarrow \left\{ (P \leftarrow Q_1,\dots,Q_n)\theta \;\middle|\; \begin{array}{l} P \leftarrow Q_1,\dots,Q_n \text{ is in } \mathbf{P},\ 1 \le j \le n, \\ Q_j\theta \in \Delta^i,\ Q_1\theta,\dots,Q_{j-1}\theta \in D^i, \text{ and} \\ Q_{j+1}\theta,\dots,Q_n\theta \in D^{i-1} \end{array} \right\} \cup C^i$
 9         $\Delta^{i+1} \leftarrow F^{i+1} - D^i$
10         $D^{i+1} \leftarrow D^i \cup \Delta^{i+1}$
11         $i \leftarrow i + 1$
12  return $C^i$
```

In the implementation, the operations in lines 7 and 8 should be performed simultaneously. In this algorithm, the final value of C^i records all rule instances $(P \leftarrow Q_1,\dots,Q_n)\theta$ such that $Q_1\theta,\dots,Q_n\theta$ are derivable facts.

On the same input as before, we have

$C^1 = \{\mathsf{book}(0,1),\ \ \mathsf{the}(1,2),\ \ \mathsf{flight}(2,3),\ \ \mathsf{from}(3,4),\ \ \mathsf{Houston}(4,5)\}$

$C^2 = \{\mathsf{N}(0,1) \leftarrow \mathsf{book}(0,1),\ \ \mathsf{V}(0,1) \leftarrow \mathsf{book}(0,1),\ \ \mathsf{Det}(1,2) \leftarrow \mathsf{the}(1,2),$
$\quad \mathsf{N}(2,3) \leftarrow \mathsf{flight}(2,3),\ \ \mathsf{P}(3,4) \leftarrow \mathsf{from}(3,4),$
$\quad \mathsf{Name}(4,5) \leftarrow \mathsf{Houston}(4,5)\} \cup C^1$

$C^3 = \{\mathsf{N1}(0,1) \leftarrow \mathsf{N}(0,1),\ \ \mathsf{VP}(0,1) \leftarrow \mathsf{V}(0,1),\ \ \mathsf{N1}(2,3) \leftarrow \mathsf{N}(2,3),$
$\quad \mathsf{NP}(4,5) \leftarrow \mathsf{Name}(4,5)\} \cup C^2$

$C^4 = \{\mathsf{NP}(1,3) \leftarrow \mathsf{Det}(1,2), \mathsf{N1}(2,3),\ \ \mathsf{PP}(3,5) \leftarrow \mathsf{P}(3,4), \mathsf{NP}(4,5)\} \cup C^3$

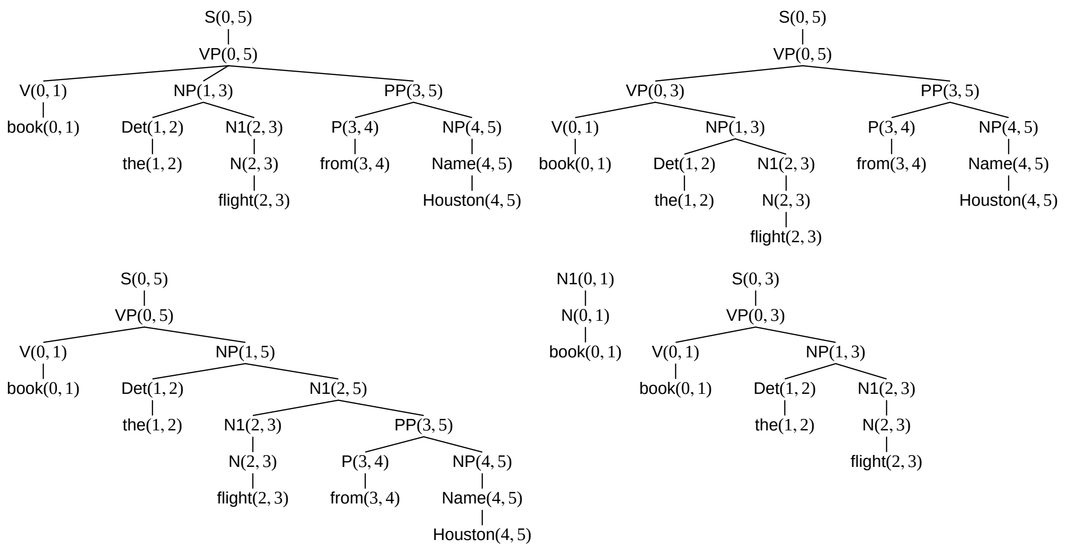
$C^5 = \{\mathsf{VP}(0,3) \leftarrow \mathsf{V}(0,1), \mathsf{NP}(1,3),\ \ \mathsf{VP}(0,5) \leftarrow \mathsf{V}(0,1), \mathsf{NP}(1,3), \mathsf{PP}(3,5),$
$\quad \mathsf{N1}(2,5) \leftarrow \mathsf{N1}(2,3), \mathsf{PP}(3,5)\} \cup C^4$

$C^6 = \{\mathsf{S}(0,3) \leftarrow \mathsf{VP}(0,3),\ \ \mathsf{S}(0,5) \leftarrow \mathsf{VP}(0,5),\ \ \mathsf{VP}(0,5) \leftarrow \mathsf{VP}(0,3), \mathsf{PP}(3,5),$
$\quad \mathsf{NP}(1,5) \leftarrow \mathsf{Det}(1,2), \mathsf{N1}(2,5)\} \cup C^5$

$C^7 = \{\mathsf{VP}(0,5) \leftarrow \mathsf{V}(0,1), \mathsf{NP}(1,5)\} \cup C^6$

Since the clauses in C^i do not contain any variables, the output of this algorithm can be regarded as a *propositional Horn clause program*. (Alternatively, it can be viewed as an AND/OR graph.) The set of derivation trees from **P**, D is the set of all derivation trees that can be formed from the clauses in the final value of C^i. Given a fact P in the final value of D^i, the derivation trees for P can be easily extracted from C^i. Note that the set of derivation trees may be infinite,[10] yet C^i is of polynomial size. In the special case of context-free grammars, this is known as a *shared parse forest*.

On the running example, the set of derivation trees from **P**, D consists of the following trees together with all their subtrees:



Notice that each "tree fragment" of depth 1 corresponds to a rule instance in C^7. This correspondence is many-one; the same rule instance may appear in more than one place.

In general Datalog query evaluation, ground facts with the same predicate are grouped together into a relation, and each group of facts in the set F^i is efficiently computed using relational algebra operations. In the application to recognition/parsing, however, the number of relevant facts is usually small, so it makes more sense to process one fact at a time. This leads to a version of the well-known *chart parsing* algorithm. (See Shieber et al. 1995 and Sikkel 1998, where a formulation of a chart parsing control algorithm is given for arbitrary deduction systems.)

[10]This will be so, for instance, when the Datalog program **P** represents a context-free grammar G that allows a cycle $A \Rightarrow_G^* A$.

```
SEMINAIVE-CHART($\mathbf{P}$, $D$)
1   $chart \leftarrow \emptyset$
2   $agenda \leftarrow D$
3   while $agenda \neq \emptyset$
4        do pop an element from $agenda$ and call it $trigger$
5           $new\_chart \leftarrow chart \cup \{trigger\}$
6           $F \leftarrow \left\{ P\theta \;\middle|\; \begin{array}{l} P \leftarrow Q_1, \dots, Q_n \text{ is in } \mathbf{P},\ 1 \le j \le n,\ trigger = Q_j\theta, \\ Q_1\theta, \dots, Q_{j-1}\theta \in new\_chart, \\ Q_{j+1}\theta, \dots, Q_n\theta \in chart \end{array} \right\}$
7           $chart \leftarrow new\_chart$
8           foreach $item \in F - (chart \cup agenda)$
9               do push $item$ onto $agenda$
10  return $chart$
```

Here, the agenda is treated as a stack.

Note that in the worst case, all ground instances of rules in **P** are considered in line 6, so that the running time of SEMINAIVE-CHART($\mathbf{P}$, D) may be at least proportional to pn^l, where p is the number of rules in **P**, l is the maximal number of variables in the rules of **P**, and n is the number of constants in D. In the special case of Datalog programs corresponding to context-free grammars in Chomsky normal form,[11] we can simplify the algorithm to the following, which is a variant of the *Cocke–Younger–Kasami* (*CYK*) algorithm (see Aho and Ullman 1972):[12]

[11]See Problem 2.2 of Lecture 2 for the definition of Chomsky normal form.

[12]In some publications, this algorithm is called *Cocke–Kasami–Younger* or *CKY*. Aho and Ullman (1972) describe Younger's (1967) algorithm and the earlier Cocke's (as described in Hays 1967) as versions of the same algorithm, while calling Kasami's (1965, 1966) "a similar algorithm". Younger (1967) was aware of Cocke's algorithm through Hays 1962, but wrote that "a parsing procedure of Kay (1963) . . . is most closely related to" his. Kay (2000) relates that Cocke came up with his algorithm during his visit to Hays's home in California, after he accompanied Hays to a machine translation conference; Kay saw the Fortran code Cocke wrote "less than a year later". According to Kay (2005), the time of Cocke's invention was in 1960. Kasami's (1965, 1966) work was independent of Cocke's or Younger's, and relied on (quadratic) Greibach normal form (see Problem 3.5) rather than Chomsky normal form.

CYK($\mathbf{P}, D$)

```
1   ▷ $\mathbf{P}$ is a Datalog program representing a CFG in CNF
2   ▷ $D = \{a_1(0,1), \ldots, a_n(\overline{n-1}, \overline{n})\}$
3   $chart \leftarrow \varnothing$
4   $agenda \leftarrow D$
5   while $agenda \neq \varnothing$
6       do pop an element from $agenda$ and call it $trigger$
7          if $trigger$ is of the form $a_k(\overline{k-1}, \overline{k})$
8             then $F \leftarrow \{\, A(\overline{k-1}, \overline{k}) \mid A(\boldsymbol{x}, \boldsymbol{y}) \leftarrow a_k(\boldsymbol{x}, \boldsymbol{y})$ is in $\mathbf{P} \,\}$
9          if $trigger$ is of the form $C(\bar{j}, \bar{k})$
10            then $F \leftarrow \left\{ A(\bar{i}, \bar{k}) \;\middle|\; \begin{array}{l} A(\boldsymbol{x}, \boldsymbol{z}) \leftarrow B(\boldsymbol{x}, \boldsymbol{y}),\ C(\boldsymbol{y}, \boldsymbol{z}) \text{ is in } \mathbf{P} \text{ and} \\ B(\bar{i}, \bar{j}) \in chart \end{array} \right\}$
11         $chart \leftarrow chart \cup \{trigger\}$
12         foreach $item \in F - (chart \cup agenda)$
13             do push $item$ onto $agenda$
14  return $chart$
```

This algorithm derives all derivable facts of the form $A(\bar{i}, \bar{j})$ after $a_j(\overline{j-1}, \bar{j})$ is popped from the agenda but before $a_{j+1}(\bar{j}, \overline{j+1})$ is, so it is not necessary to consider a rule of the form $A(\boldsymbol{x}, \boldsymbol{z}) \leftarrow C(\boldsymbol{x}, \boldsymbol{y}),\ B(\boldsymbol{y}, \boldsymbol{z})$ in line 10. The behavior of this algorithm is close to the standard formulation of the CYK algorithm, and, with appropriate data structures, it can be implemented to run in time $O(pn^3)$, where p is the size of $\mathbf{P}$ and n is the size of the input.

Exercise 3.3. Formulate versions of SEMINAIVE-CHART and of CYK that output a representation of the set of all derivation trees.

Chart parsing To obtain better asymptotic time complexity for general Datalog programs, we use new inference rules instead of (3.1).

$$\frac{P \leftarrow Q, Q_1, \ldots, Q_n \quad Q\theta}{(P \leftarrow Q_1, \ldots, Q_n)\theta} \quad (n \geq 1)$$

$$\frac{P \leftarrow Q \quad Q\theta}{P\theta}$$

We store ground facts as well as (possibly) non-ground definite clauses in the chart. In the special case of Datalog programs representing context-free grammars without ε-productions, these inference rules take the following form ($n \geq 2$ in the first two inference rules):

$$\frac{A(\boldsymbol{x}_0, \boldsymbol{x}_n) \leftarrow Y_1(\boldsymbol{x}_0, \boldsymbol{x}_1), \ldots, Y_n(\boldsymbol{x}_{n-1}, \boldsymbol{x}_n) \quad Y_1(\bar{i}, \bar{j})}{A(\bar{i}, \boldsymbol{x}_n) \leftarrow Y_2(\bar{j}, \boldsymbol{x}_2), \ldots, Y_n(\boldsymbol{x}_{n-1}, \boldsymbol{x}_n)} \tag{3.2}$$

$$\frac{A(\bar{i},\boldsymbol{x}_n) \leftarrow Y_1(\bar{j},\boldsymbol{x}_1), Y_2(\boldsymbol{x}_1,\boldsymbol{x}_2),\ldots,Y_n(\boldsymbol{x}_{n-1},\boldsymbol{x}_n) \quad Y_1(\bar{j},\bar{k})}{A(\bar{i},\boldsymbol{x}_n) \leftarrow Y_2(\bar{k},\boldsymbol{x}_2),\ldots,Y_n(\boldsymbol{x}_{n-1},\boldsymbol{x}_n)} \tag{3.3}$$

$$\frac{A(\boldsymbol{x}_0,\boldsymbol{x}_1) \leftarrow Y(\boldsymbol{x}_0,\boldsymbol{x}_1) \quad Y(\bar{i},\bar{j})}{A(\bar{i},\bar{j})} \tag{3.4}$$

$$\frac{A(\bar{i},\boldsymbol{x}) \leftarrow Y(\bar{j},\boldsymbol{x}) \quad Y(\bar{j},\bar{k})}{A(\bar{i},\bar{k})} \tag{3.5}$$

The following algorithm is sometimes called the *bottom-up left-corner parser* (Sikkel 1997, Ljunglöf and Wirén 2010)[13] and is close to what is known as the *bottom-up chart parser* (Kay 1986; see Gazdar and Mellish 1989 or Samuelsson and Wirén 2000). We call it the *bottom-up left-corner chart parser* to emphasize that it is a tabular recognition algorithm.

```
BOTTOM-UP-LC-CHART($\mathbf{P}$, $D$)
 1  ▷ $\mathbf{P}$ is a Datalog program representing a CFG without $\varepsilon$-productions
 2  ▷ $D = \{a_1(0,1),\ldots,a_n(\overline{n-1},\bar{n})\}$
 3  $chart \leftarrow \emptyset$
 4  $agenda \leftarrow D$
 5  while $agenda \neq \emptyset$
 6      do pop an element from $agenda$ and call it $trigger$
 7         if $trigger$ is a fact $Q$
 8            then $F \leftarrow \left\{ P \,\middle|\, \begin{array}{l} R \in chart \cup \mathbf{P} \text{ and } P \text{ follows from } R \text{ and } Q \\ \text{by one of (3.2)–(3.5)} \end{array} \right\}$
 9            else $F \leftarrow \emptyset$
10         $chart \leftarrow chart \cup \{trigger\}$
11         foreach $item \in F - (chart \cup agenda)$
12             do push $item$ onto $agenda$
13  return $chart$
```

With appropriate data structures, BOTTOM-UP-LC-CHART can be implemented to run in time $O(pn^3)$.

Exercise 3.4. List the facts and clauses generated by the procedure BOTTOM-UP-LC-CHART when run on the program in Figure 3.3 and the database $\{\mathsf{book}(0,1), \mathsf{the}(1,2), \mathsf{flight}(2,3), \mathsf{from}(3,4), \mathsf{Houston}(4,5)\}$, in the order they are popped from the agenda.

[13]The bottom-up left-corner chart parser was first proposed as an improvement over Earley's (1970) algorithm (see Leiss 1990).

If $A(\boldsymbol{x}_0, \boldsymbol{x}_n) \leftarrow Y_1(\boldsymbol{x}_0, \boldsymbol{x}_1), \dots, Y_n(\boldsymbol{x}_{n-1}, \boldsymbol{x}_n)$ is a clause representing a context-free production $A \to Y_1 \dots Y_n$, a derived clause

$$A(\bar{i}, \boldsymbol{x}_n) \leftarrow Y_l(\bar{j}, \boldsymbol{x}_l), \dots, Y_n(\boldsymbol{x}_{n-1}, \boldsymbol{x}_n)$$

corresponds to what is usually called an *item*

$$(A \to Y_1 \dots Y_{l-1} \bullet Y_l \dots Y_n, i, j)$$

where $A \to Y_1 \dots Y_{l-1} \bullet Y_l \dots Y_n$ is a *dotted production*, i.e., a production in the CFG with a dot inserted somewhere in the middle of the right-hand side. (Remember that the free variables in a derived clause are implicitly universally quantified.) We may treat dotted productions as binary predicates. If we do this, items are nothing but facts:

$$[A \to Y_1 \dots Y_{l-1} \bullet Y_l \dots Y_n](\bar{i}, \bar{j}).$$

Inference rules (3.2), (3.3), (3.5) can now be reformulated as definite clauses, i.e., Datalog rules ($n \geq 2$ and $2 \leq l \leq n - 1$):

(3.6) $\quad [A \to Y_1 \bullet Y_2 \dots Y_n](\boldsymbol{x}, \boldsymbol{y}) \leftarrow Y_1(\boldsymbol{x}, \boldsymbol{y}).$

(3.7)
$$[A \to Y_1 \dots Y_l \bullet Y_{l+1} \dots Y_n](\boldsymbol{x}, \boldsymbol{z}) \leftarrow [A \to Y_1 \dots Y_{l-1} \bullet Y_l \dots Y_n](\boldsymbol{x}, \boldsymbol{y}),\ Y_l(\boldsymbol{y}, \boldsymbol{z}).$$

(3.8) $\quad A(\boldsymbol{x}, \boldsymbol{z}) \leftarrow [A \to Y_1 \dots Y_{n-1} \bullet Y_n](\boldsymbol{x}, \boldsymbol{y}),\ Y_n(\boldsymbol{y}, \boldsymbol{z}).$

Note that the remaining inference rule (3.4) is just a special case of (3.1). Given a Datalog program representing a CFG without ε-productions, if we rewrite each rule $A(\boldsymbol{x}_0, \boldsymbol{x}_n) \leftarrow Y_1(\boldsymbol{x}_0, \boldsymbol{x}_1), \dots, Y_n(\boldsymbol{x}_{n-1}, \boldsymbol{x}_n)$ with $n \geq 2$ into n rules using (3.6)–(3.8), leaving rules of the form $A(\boldsymbol{x}_0, \boldsymbol{x}_1) \leftarrow Y(\boldsymbol{x}_0, \boldsymbol{x}_1)$ intact, the result is a program where each clause hast at most two atoms in the body. For example, the program in Figure 3.3 is transformed into the one in Figure 3.4 by this procedure. (Note that this program represents a CFG whose productions all have right-hand sides of length 1 or 2 and which is equivalent to the original CFG.) We can run (a simplified version of) the procedure Seminaive-chart[14] against such programs with the running time $O(pn^3)$.

In general, we can express a tabular recognition (parsing) algorithm by specifying a method of transforming a Datalog program directly representing a grammar (“original program”) into another Datalog program (“transformed program”), abstracting away the *control algorithm* for generating facts (e.g., a canonical chart parsing algorithm like Seminaive-chart) that is used in conjunction with the transformed program. When we do this, the transformed program is called a *deduction system* (Shieber et al. 1995), and the method of conversion from the CFG

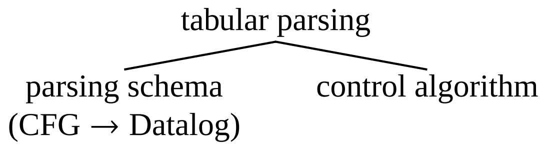
[14]In line 6, n is either 1 or 2, and j can be set to n due to the absence of the ε-productions.

```
[S → NP • VP](i, j) ← NP(i, j).
S(i, k) ← [S → NP • VP](i, j), VP(j, k).
[S → Aux • NP VP](i, j) ← Aux(i, j).
[S → Aux NP • VP](i, k) ←
      [S → Aux • NP VP](i, j), Aux(j, k).
S(i, l) ← [S → Aux NP • VP](i, k), VP(k, l).
S(i, j) ← VP(i, j).
[NP → Det • N1](i, j) ← Det(i, j).
NP(i, k) ← [NP → Det • N1](i, j), N1(j, k).
NP(i, j) ← Name(i, j).
NP(i, j) ← Pronoun(i, j).
VP(i, j) ← V(i, j).
[VP → V • NP](i, j) ← V(i, j).
VP(i, k) ← [VP → V • NP](i, j), NP(j, k).
[VP → V • NP PP](i, j) ← V(i, j).
[VP → V NP • PP](i, k) ←
      [VP → V • NP PP](i, j), NP(j, k).
VP(i, l) ← [VP → V NP • PP](i, k), PP(k, l).
[VP → VP • PP](i, j) ← VP(i, j).
VP(i, k) ← [VP → VP • PP](i, j), PP(j, k).
N1(i, j) ← N(i, j).
[N1 → N1 • N](i, j) ← N1(i, j).
N1(i, k) ← [N1 → N1 • N](i, j), N(j, k).
[N1 → N1 • PP](i, j) ← N1(i, j).
N1(i, k) ← [N1 → N1 • PP](i, j), PP(j, k).
[PP → P • NP](i, j) ← P(i, j).
PP(i, k) ← [PP → P • NP](i, j), NP(j, k).

Aux(i, j) ← does(i, j).
Det(i, j) ← that(i, j).
Det(i, j) ← this(i, j).
Det(i, j) ← a(i, j).
Name(i, j) ← Houston(i, j).
Name(i, j) ← TWA(i, j).
Pronoun(i, j) ← I(i, j).
Pronoun(i, j) ← she(i, j).
Pronoun(i, j) ← me(i, j).
V(i, j) ← book(i, j).
V(i, j) ← include(i, j).
V(i, j) ← prefer(i, j).
N(i, j) ← book(i, j).
N(i, j) ← flight(i, j).
N(i, j) ← meal(i, j).
N(i, j) ← money(i, j).
P(i, j) ← from(i, j).
P(i, j) ← to(i, j).
P(i, j) ← on(i, j).
```

Figure 3.4: A transformed Datalog program for bottom-up chart parsing.

to the transformed Datalog program is called a *parsing schema* (Sikkel 1997). In other words, a tabular recognition/parsing algorithm can be presented as the combination of a parsing schema and a control algorithm:



Earley's algorithm Earley's (1970) algorithm (or, more precisely, a variant thereof) can be viewed as a refinement of the bottom-up left-corner chart parser that allows partial reduction of parse forest by “top-down filtering”. With a slight modification of the control algorithm, the algorithm can be made to satisfy the *correct prefix property*.

The way to obtain (a variant of) Earley’s algorithm from bottom-up chart parsing is to add predicates that express *top-down prediction*. For an intensional predicate $A(\boldsymbol{i}, \boldsymbol{j})$ (corresponding to a nonterminal), we use $\sim A(\boldsymbol{i})$ as a unary predicate expressing the prediction that nonterminal A may start at position i. The fact $\sim A(\bar{i})$ will be derivable if and only if there is a leftmost derivation $S \Rightarrow_G^* a_1 \dots a_i A\alpha$ where $a_1 \dots a_i$ is the prefix of the input string of length i, or equivalently, if and only if the top-down stack-based recognizer can reach the configuration $(A\alpha, a_{i+1} \dots a_n)$ starting with $(S, a_1 \dots a_n)$, for some $\alpha \in (N \cup \Sigma)^*$.

Let **P** be the deduction system for bottom-up chart parsing obtained from a program representing a CFG. The deduction system **P′** for Earley parsing is obtained from **P** in the following way:

1. Each rule of the form
$$A(\boldsymbol{i}, \boldsymbol{j}) \leftarrow X(\boldsymbol{i}, \boldsymbol{j})$$
in **P** where A is a nonterminal will be replaced by
$$A(\boldsymbol{i}, \boldsymbol{j}) \leftarrow \sim A(\boldsymbol{i}),\ X(\boldsymbol{i}, \boldsymbol{j})$$
in **P′**.
2. Each rule of the form
$$[A \to Y_1 \bullet Y_2 \dots Y_n](\boldsymbol{i}, \boldsymbol{j}) \leftarrow Y_1(\boldsymbol{i}, \boldsymbol{j})$$
in **P** will be replaced by
$$[A \to Y_1 \bullet Y_2 \dots Y_n](\boldsymbol{i}, \boldsymbol{j}) \leftarrow \sim A(\boldsymbol{i}),\ Y_1(\boldsymbol{i}, \boldsymbol{j})$$
in **P′**.
3. For each rule
$$A(\boldsymbol{i}, \boldsymbol{j}) \leftarrow B(\boldsymbol{i}, \boldsymbol{j})$$
in **P** such that A and B are nonterminals, there will be a new rule
$$\sim B(\boldsymbol{i}) \leftarrow \sim A(\boldsymbol{i})$$
in **P′**.
4. For each rule
$$[A \to Y_1 \bullet Y_2 \dots Y_n](\boldsymbol{i}, \boldsymbol{j}) \leftarrow Y_1(\boldsymbol{i}, \boldsymbol{j})$$
in **P** such that Y_1 is a nonterminal, there will be a new rule
$$\sim Y_1(\boldsymbol{i}) \leftarrow \sim A(\boldsymbol{i})$$
in **P′**.

5. For each rule of the form

$$A(\boldsymbol{i},\boldsymbol{k}) \leftarrow B(\boldsymbol{i},\boldsymbol{j}),\ C(\boldsymbol{j},\boldsymbol{k})$$

in **P**, where C is a nonterminal, **P′** will have a new rule

$$\sim C(\boldsymbol{j}) \leftarrow B(\boldsymbol{i},\boldsymbol{j}).$$

For example, from the deduction system in Figure 3.4, we obtain the one in Figure 3.5. In addition to these rules, we need the "seed fact":

$$\sim \mathsf{S}(0).$$

We include this fact in the input database.

It is easy to see that whenever $A(\bar{i},\bar{j})$ is derivable from the Earley deduction system (together with the input database), we have the following:

- $\sim A(\bar{i})$ is derivable from the Earley deduction system (together with the input database), and
- $A(\bar{i},\bar{j})$ is derivable from the deduction system for bottom-up chart parsing (together with the input database).

The *correct prefix property* refers to the following condition:[15]

- On input $a_1 \dots a_n$, the algorithm reports failure before scanning the $(i+1)$-st symbol of the input whenever $a_1 \dots a_i$ is not a prefix of any element of the language.

In the case of tabular parsing algorithms of the kind we are considering, "scanning the $(i + 1)$-st symbol of the input" means popping $a_{i+1}(\bar{i},\overline{i+1})$ from the agenda.

On the assumption that the grammar is "reduced" in the sense that every nonterminal derives some string, the correct prefix property can be achieved by coupling the Earley deduction system with the following modification of the SEMINAIVE-CHART control algorithm:[16]

[15]The term was originally used to refer to the following property of stack-based parsers (Sippu and Soisalon-Soininen 1990): if a configuration (α,ε) is reachable from (γ,x) where γ is the initial stack content, then x is a prefix of some element of the language.

[16]See footnote 14.

```
SEMINAIVE-CHART($\mathbf{P}$, $D$)
chart ← ∅
agenda ← D
while agenda ≠ ∅
    do pop an element from agenda and call it trigger
       new_chart ← chart ∪ {trigger}
       $F \leftarrow \left\{ P\theta_1 \dots \theta_n \;\middle|\; \begin{array}{l} P \leftarrow Q_1, \dots, Q_n \text{ is in } \mathbf{P},\ 1 \le j \le n, \\ \mathit{trigger} = Q_j\theta_j, \\ Q_1\theta_1, \dots, Q_{j-1}\theta_{j-1} \in \mathit{new\_chart}, \\ Q_{j+1}\theta_{j+1}, \dots, Q_n\theta_n \in \mathit{chart}, \text{ and} \\ \theta_1, \dots, \theta_n \text{ are compatible} \end{array} \right\}$
       if trigger is of the form $a_{i+1}(\overline{i}, \overline{i+1})$ and $F = \emptyset$
          then reject
       chart ← new_chart
       foreach item ∈ F − (chart ∪ agenda)
           do push item onto agenda
return chart
```

The composition of the parsing schema for bottom-up left-corner chart parsing and the above transformation is the parsing schema for Earley parsing. It is an instance of a technique in Datalog known as *generalized supplementary magic-sets rewriting* (Beeri and Ramakrishnan 1991; see Ullman 1989a,b or Abiteboul et al. 1995).

Exercise 3.5. List the facts generated by the procedure SEMINAIVE-CHART when run on the program in Figure 3.5 and the database $\{\sim\mathsf{S}(0), \mathsf{book}(0,1), \mathsf{the}(1,2), \mathsf{flight}(2,3), \mathsf{from}(3,4), \mathsf{Houston}(4,5)\}$, in the order they are popped from the agenda.

From pushdown automata to context-free grammars

There are tabular, dynamic programming algorithms for determining whether an input string is accepted by a pushdown automaton (and if so, computing the set of accepting transition sequences on the input). Therefore, it is always possible to turn a nondeterministic stack-based recognition/parsing algorithm for context-free grammars into a polynomial-time deterministic one, after all. Since these tabular algorithms are closely related to methods of converting pushdown automata into equivalent context-free grammars, let us first look at how this conversion can be done.

```
~NP(i) ← ~S(i).
~VP(j) ← [S → NP • VP](i, j).
~Aux(i) ← ~S(i).
~NP(j) ← [S → Aux • NP VP](i, j).
~VP(k) ← [S → Aux NP • VP](i, k).
~VP(i) ← ~S(i).
~Det(i) ← ~NP(i).
~N1(j) ← [NP → Det • N1](i, j).
~Name(i) ← ~NP(i).
~Pronoun(i) ← ~NP(i).

~V(i) ← ~VP(i).
~NP(j) ← [VP → V • NP](i, j).
~NP(j) ← [VP → V • NP PP](i, j).
~PP(k) ← [VP → V NP • PP](i, k).
~PP(k) ← [VP → VP • PP](i, k).
~N(i) ← ~N1(i).
~N(j) ← [N1 → N1 • N](i, j).
~PP(j) ← [N1 → N1 • PP](i, j).
~P(i) ← ~PP(i).
~NP(j) ← [PP → P • NP](i, j).

[S → NP • VP](i, j) ← ~S(i), NP(i, j).
S(i, k) ← [S → NP • VP](i, j), VP(j, k).
[S → Aux • NP VP](i, j) ← ~S(i), Aux(i, j).
[S → Aux NP • VP](i, k) ←
      [S → Aux • NP VP](i, j), NP(j, k).
S(i, l) ← [S → Aux NP • VP](i, k), VP(k, l).
S(i, j) ← ~S(i), VP(i, j).
[NP → Det • N1](i, j) ← ~NP(i), Det(i, j).
NP(i, k) ← [NP → Det • N1](i, j), N1(j, k).
NP(i, j) ← ~NP(i), Name(i, j).
NP(i, j) ← ~NP(i), Pronoun(i, j).
VP(i, j) ← ~VP(i), V(i, j).
[VP → V • NP](i, j) ← ~VP(i), V(i, j).
VP(i, k) ← [VP → V • NP](i, j), NP(j, k).
[VP → V • NP PP](i, j) ← ~VP(i), V(i, j).
[VP → V NP • PP](i, k) ←
      [VP → V • NP PP](i, j), NP(j, k).
VP(i, l) ← [VP → V NP • PP](i, k), PP(k, l).
[VP → VP • PP](i, j) ← ~VP(i), VP(i, j).
VP(i, k) ← [VP → VP • PP](i, j), PP(j, k).
N1(i, j) ← ~N1(i), N(i, j).
[N1 → N1 • N](i, j) ← ~N1(i), N1(i, j).
N1(i, k) ← [N1 → N1 • N](i, j), N(j, k).
[N1 → N1 • PP](i, j) ← ~N1(i), N1(i, j).
N1(i, k) ← [N1 → N1 • PP](i, j), PP(j, k).
[PP → P • NP](i, j) ← ~PP(i), P(i, j).
PP(i, k) ← [PP → P • NP](i, j), NP(j, k).

Aux(i, j) ← ~Aux(i), does(i, j).
Det(i, j) ← ~Det(i), that(i, j).
Det(i, j) ← ~Det(i), this(i, j).
Det(i, j) ← ~Det(i), a(i, j).
Name(i, j) ← ~Name(i), Houston(i, j).
Name(i, j) ← ~Name(i), TWA(i, j).
Pronoun(i, j) ← ~Pronoun(i), I(i, j).
Pronoun(i, j) ← ~Pronoun(i), she(i, j).
Pronoun(i, j) ← ~Pronoun(i), me(i, j).
V(i, j) ← ~V(i), book(i, j).
V(i, j) ← ~V(i), include(i, j).
V(i, j) ← ~V(i), prefer(i, j).
N(i, j) ← ~N(i), book(i, j).
N(i, j) ← ~N(i), flight(i, j).
N(i, j) ← ~N(i), meal(i, j).
N(i, j) ← ~N(i), money(i, j).
P(i, j) ← ~P(i), from(i, j).
P(i, j) ← ~P(i), to(i, j).
P(i, j) ← ~P(i), on(i, j).
```

Figure 3.5: A transformed Datalog program for Earley parsing.

We begin with formal definitions of pushdown automata. A *pushdown automaton* (PDA) is an 8-tuple $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$, where[17]

1. Q is a finite set of *states*,
2. Σ is a finite set called the *input alphabet*,
3. Γ is a finite set called the *stack alphabet*,
4. δ is a finite subset of $Q \times (\Gamma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$, called the set of *transitions*,
5. $q_I \in Q$ is the *initial state*,
6. $Z_I \in \Gamma \cup \{\varepsilon\}$ is the *initial stack content*, and
7. $Q_F \subseteq Q$ is the set of *final states*,
8. $Z_F \in \Gamma \cup \{\varepsilon\}$ is the *final stack content*.

We denote an element (q, Z, a, r, γ) of δ by

$$(q, Z) \overset{a}{\mapsto} (r, \gamma).$$

A *configuration* of a PDA M is a triple $(q, \alpha, x) \in Q \times \Gamma^* \times \Sigma^*$. The *initial configuration* on an input $x \in \Sigma^*$ is (q_I, Z_I, x), and an *accepting configuration* is (q, Z_F, ε), where $q \in Q_F$. Given a PDA M, we define the binary relation $\vdash_M$ between configurations as follows:

$$\vdash_M = \{\, ((q, Z\alpha, ax), (q', \gamma\alpha, x)) \mid (q, Z) \overset{a}{\mapsto} (q', \gamma) \in \delta \,\}$$

A PDA M *accepts* an input string $x \in \Sigma^*$ if and only if $(q_I, Z_I, x) \vdash_M^* (q, Z_F, \varepsilon)$ for some $q \in Q_F$. We say that M *accepts* a language $L \subseteq \Sigma^*$ if $L = \{\, x \in \Sigma^* \mid M \text{ accepts } x \,\}$. We write $L(M)$ for the language M accepts.

A PDA $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$ is said to be *stateless* if $|Q| = 1$. (This implies that $Q = Q_F = \{q_I\}$, ignoring the uninteresting case of $Q_F = \emptyset$.) A stateless PDA can be thought of as a 5-tuple $(\Sigma, \Gamma, \delta, Z_I, Z_F)$, where δ is a finite subset of $(\Gamma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$. In this case, an element of δ can be written in the form $Z \overset{a}{\mapsto} \gamma$, and configurations can be represented by pairs $(\alpha, x) \in \Gamma^* \times \Sigma^*$.

An *extended PDA* $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$ (Aho and Ullman 1972) is like a PDA except that δ is a finite subset of $Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$. The definition of $\vdash_M$ and of acceptance is the same as with PDAs.

[17]The exact definition of PDA varies. Transitions of the form $(q, \varepsilon, a, r, \gamma)$ are often not allowed (Aho and Ullman 1972, Hopcroft and Ullman 1979, Kozen 1997). The last component Z_F is usually not part of the definition of PDAs but rather of the definition of acceptance. It is actually too restrictive to require specification of a single final stack content; when we consider deterministic PDAs, we must allow acceptance by arbitrary final stack.

The top-down recognizer we looked at in Lecture 2 constructed from a CFG is an example of a stateless PDA. The bottom-up stack-based recognizer is an example of a stateless extended PDA.

Theorem 3.1. *Given any extended PDA M, one can construct a PDA M' such that $L(M) = L(M')$.*

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$ be an extended PDA. We construct a PDA $M' = (Q', \Sigma, \Gamma, \delta', q_I, Z_I, Q_F, Z_F)$ as follows. Let Q' contain all states of Q, and let δ' contain all transitions in δ of the form $(q, \gamma) \overset{a}{\mapsto} (r, \gamma')$, where $|\gamma| \leq 1$. For each transition in δ of the form $(q, X_1 \ldots X_m) \overset{a}{\mapsto} (r, \gamma)$ with $m \geq 2$, add new states $q_1, \ldots, q_{m-1}$ to Q', and add the following transitions to δ':

$$\begin{aligned}
(q, X_1) &\overset{a}{\mapsto} (q_1, \varepsilon),\\
(q_1, X_2) &\overset{\varepsilon}{\mapsto} (q_2, \varepsilon),\\
&\vdots\\
(q_{m-2}, X_{m-1}) &\overset{\varepsilon}{\mapsto} (q_{m-1}, \varepsilon),\\
(q_{m-1}, X_m) &\overset{\varepsilon}{\mapsto} (r, \gamma).
\end{aligned}$$

It is easy to see that the computations of M and M' on the same input are in one-to-one correspondence and $L(M) = L(M')$. □

We call a PDA $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$ *standard* if $Z_I \in \Gamma$, $Z_F = \varepsilon$, and δ is a finite subset of $Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$.

Theorem 3.2. *Given any PDA M, one can construct a standard PDA M' such that $L(M) = L(M')$.*

Exercise 3.6. Prove Theorem 3.2.

Theorem 3.3. *Given any standard PDA M, one can construct a stateless standard PDA M' such that $L(M) = L(M')$.*

Proof (sketch). Let $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, \varepsilon)$ be a standard PDA. We define a stateless PDA $M' = (\Sigma, \Gamma', \delta', Z_I', \varepsilon)$ as follows. Let

$$\Gamma' = \{\, [qXr] \mid q, r \in Q, X \in \Gamma \,\} \cup \{Z_I'\},$$

where Z_I' is a new stack symbol. Let δ' contain the following transitions:

- for each transition $(q, X) \overset{a}{\mapsto} (r, Y_1 \ldots Y_n) \in \delta$ with $n \geq 1$, all transitions of the form $[qXs_n] \overset{a}{\mapsto} [rY_1s_1][s_1Y_2s_2] \ldots [s_{n-1}Y_ns_n]$, where $[qXs_n], [rY_1s_1], [s_1Y_2s_2], \ldots, [s_{n-1}Y_ns_n] \in \Gamma'$,

- for each transition $(q, X) \overset{a}{\mapsto} (r, \varepsilon) \in \delta$, the transition $[qXr] \overset{a}{\mapsto} \varepsilon$,
- for each $q \in Q_F$, the transition $Z'_I \overset{\varepsilon}{\mapsto} [q_I Z_I q]$.

We can show that $(q, X, y) \vdash^*_M (r, \varepsilon, \varepsilon)$ if and only if $([qXr], y) \vdash^*_{M'} (\varepsilon, \varepsilon)$ and that there is a one-to-one correspondence between the accepting computations of M and M' on the same input. □

PDA-to-CFG conversion, method 1 Let $M = (\Sigma, \Gamma, \delta, Z_I, \varepsilon)$ be a stateless standard PDA. We assume that $\Sigma \cap \Gamma = \varnothing$. Define a context-free grammar $G = (N, \Sigma, P, S)$ by

$$\begin{aligned} N &= \Gamma, \\ P &= \{\, X \to aY_1 \dots Y_n \mid X \overset{a}{\mapsto} Y_1 \dots Y_n \in \delta \,\}, \\ S &= Z_I. \end{aligned}$$

Let $x \in \Sigma^*$, and let $\alpha = y\alpha'$ be a left sentential form of G such that $x = yz$ and α' does not begin with a terminal. Define $h_x(\alpha) = (\alpha', z)$. Then, for every leftmost derivation

$$S = \alpha_0 \underset{\mathrm{lm}}{\Rightarrow} \alpha_1 \underset{\mathrm{lm}}{\Rightarrow} \dots \underset{\mathrm{lm}}{\Rightarrow} \alpha_n = x \in \Sigma^*$$

of G,

$$h_x(\alpha_0) \vdash h_x(\alpha_1) \vdash \cdots \vdash h_x(\alpha_n)$$

is an accepting computation of M on input x, and moreover, every accepting computation of M on input x is obtained in this way. Note that the top-down recognizer for G, as defined in Lecture 2, is almost the same as M.

Theorem 3.4 (Chomsky)**.** *For every PDA M, there is a context-free grammar G such that $L(M) = L(G)$.*

The above theorem was first obtained by Chomsky (1962) by a slightly different method. The proof given here follows many textbooks (Aho and Ullman 1972, Harrison 1978, Hopcroft and Ullman 1979, Kozen 1997, among others).

The left-corner transform Let us define a stateless standard PDA which is equivalent to the (arc-eager) left-corner recognizer we defined in Lecture 2. Let $G = (N, \Sigma, P, S)$ be a CFG without ε-productions. Define a stateless PDA $M = (\Sigma, \Gamma, \delta, Z_I)$ by

$$\Gamma = \{\, {\sim}X \mid X \in N \cup \Sigma \,\} \cup \{\, [X\,{\sim}Y] \mid X \in N \cup \Sigma, Y \in N \,\},$$

$$\begin{aligned}\delta = {} & \{\, {\sim}X \overset{a}{\mapsto} [a \,{\sim}X] \mid X \in N, a \in \Sigma \,\} \cup \\ & \{\, {\sim}a \overset{a}{\mapsto} \varepsilon \mid a \in \Sigma \,\} \cup \\ & \{\, [Y_1 \,{\sim}Z] \overset{\varepsilon}{\mapsto} {\sim}Y_2 \ldots {\sim}Y_n[X \,{\sim}Z] \mid X \to Y_1 \ldots Y_n \in P \,\} \cup \\ & \{\, [Y_1 \,{\sim}X] \overset{\varepsilon}{\mapsto} {\sim}Y_2 \ldots {\sim}Y_n \mid X \to Y_1 \ldots Y_n \in P \,\}, \\ Z_I = {} & {\sim}S.\end{aligned}$$

Then the CFG $G' = (N', \Sigma, P', S')$ obtained from M by the above method is

$$\begin{aligned}N' = {} & \Gamma, \\ P' = {} & \{\, {\sim}X \to a\,[a \,{\sim}X] \mid X \in N, a \in \Sigma \,\} \cup \\ & \{\, {\sim}a \to a \mid a \in \Sigma \,\} \cup \\ & \{\, [Y_1 \,{\sim}Z] \to {\sim}Y_2 \ldots {\sim}Y_n[X \,{\sim}Z] \mid X \to Y_1 \ldots Y_n \in P \,\} \cup \\ & \{\, [Y_1 \,{\sim}X] \to {\sim}Y_2 \ldots {\sim}Y_n \mid X \to Y_1 \ldots Y_n \in P \,\}, \\ S' = {} & {\sim}S.\end{aligned}$$

The conversion from G to G' is a variant of what is known as the *left-corner transform* (Rosenkrantz and Lewis 1970). The accepting computations of the (arc-eager) left-corner recognizer for G correspond one-to-one to the accepting computations of the top-down recognizer for G'.

Exercise 3.7. Apply the left-corner transform as defined above to the CFG in Figure 3.2.

Tabulation of pushdown automata

Once we have a method of converting a pushdown automaton to an equivalent CFG, we can apply any tabular recognition algorithm for CFGs to obtain a tabular recognition algorithm for pushdown automata. For example, using the method of the preceding section, it is easy to obtain a tabular version of the (arc-eager) left-corner recognizer.

Exercise 3.8. Obtain a deduction system for bottom-up left-corner chart parsing from the result of Exercise 3.7.

PDA-to-CFG conversion, method 2 Here is another common method for converting PDAs to CFGs. This applies to a restricted kind of pushdown automaton we call *normal*. A PDA $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$ is *normal* if $Z_I = Z_F = \varepsilon$ and each transition in δ is of one of the following two types:

$$(q, X) \overset{a}{\mapsto} (r, \varepsilon),$$

$$(q, \varepsilon) \overset{a}{\mapsto} (r, X),$$

where $X \in \Gamma$. Transitions of the first type are called *pop* transitions, and transitions of the second type are called *push* transitions. Let M be a normal PDA. A pair (p, q) of states of M is called a *realizable pair* if

$$(p, \varepsilon, x) \vdash_M^* (q, \varepsilon, \varepsilon)$$

for some $x \in \Sigma^*$. The context-free grammar G we construct out of M has nonterminals of the form $[pq]$, where (p, q) is a realizable pair, in addition to the start symbol S. G will contain productions of the following forms:

$[pt] \to a\,[qr]\,b\,[st]$ if δ contains $(p, \varepsilon) \overset{a}{\mapsto} (q, X)$ and $(r, X) \overset{b}{\mapsto} (s, \varepsilon)$ for some $X \in \Gamma$,
$[pp] \to \varepsilon$,
$S \to [q_I q]$ for each $q \in Q_F$.

This is closer to Chomsky's (1962) original proof, and variants of this method are found in some textbooks (Floyd and Beigel 1994, Sipser 2012).

Exercise 3.9. Extend the method of PDA-to-CFG conversion based on realizable pairs to PDAs having, in addition to pop and push transitions, transitions of the form

$$(q, \varepsilon) \overset{a}{\mapsto} (r, \varepsilon).$$

Exercise 3.10. Show that every PDA can be converted to an equivalent normal PDA.

PDA-to-CFG conversion, method 3 Here is yet another method, based on the work of Aho et al. (1968).[18] This applies to pushdown automata each of whose transitions is of one of the following forms:

$$(p, X) \overset{a}{\mapsto} (q, \varepsilon)$$
$$(p, X) \overset{a}{\mapsto} (q, YX)$$

The productions of the CFG are

$$[pXq] \to a \qquad \text{for } (p, X) \overset{a}{\mapsto} (q, \varepsilon),$$
$$[pXs] \to a\,[qYr]\,[rXs] \qquad \text{for } (p, X) \overset{a}{\mapsto} (q, YX),$$

in addition to the bookkeeping productions concerning the start symbol.[19]

[18]Aho et al. (1968) present an agenda-driven (!) dynamic programming algorithm without relating it to PDA-to-CFG conversion. However, the connection is obvious. Cook (1971) presents a construction using realizable pairs, calling it a "generalization" of Aho et al.'s (1968) argument, while Ruzzo (1980) calls Cook's (1971) construction one that "generalizes the classic proof that PDAs accept only context-free languages".

[19]Assuming $Z_F = \varepsilon$, we need a production $S \to [q_I Z_I q]$ for each $q \in F$.

Lang's tabular recognition algorithm for PDAs The influential tabular recognition algorithm of Lang (1974) can be seen as a refinement of the *reverse* of Aho et al's (1968) construction. Lang's (1974) method applies to pushdown automata whose transitions are of the forms:

$$(q, \varepsilon) \overset{a}{\mapsto} (r, \varepsilon),$$
$$(q, X) \overset{a}{\mapsto} (r, \varepsilon),$$
$$(q, \varepsilon) \overset{a}{\mapsto} (r, X),$$
$$(q, X) \overset{a}{\mapsto} (r, Y).$$

Nonterminals of the CFG are of the form $[pXqY]$ and correspond to partial computations of the form

$$(p, X, ax) \vdash (r, ZX, x) \vdash^* (q, YX, \varepsilon)$$

where the configurations appearing between (r, ZX, x) and (q, YX) all have stack height ≥ 2. Similarly to Aho et al.'s (1968) method, we obtain the CFG productions:

$$\begin{array}{ll}
[pXrY] \to [pXqY]\,a & \text{for } (q, \varepsilon) \overset{a}{\mapsto} (r, \varepsilon), \\
[qXrY] \to a & \text{for } (q, \varepsilon) \overset{a}{\mapsto} (r, Y), \\
[pXsY] \to [pXqY]\,[qYrZ]\,a & \text{for } (r, Z) \overset{a}{\mapsto} (s, \varepsilon), \\
[pXrY] \to [pXqZ]\,a & \text{for } (q, Z) \overset{a}{\mapsto} (r, Y).
\end{array}$$

A slight modification of the Datalog representation of the above productions gives Lang's (1974) deduction system:

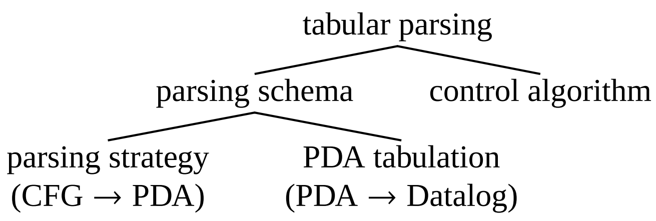
$$\begin{array}{ll}
[pXrY](\boldsymbol{i}, \boldsymbol{k}) \leftarrow [pXqY](\boldsymbol{i}, \boldsymbol{j}),\ a(\boldsymbol{j}, \boldsymbol{k}) & \text{for } (q, \varepsilon) \overset{a}{\mapsto} (r, \varepsilon), \\
[qXrY](\boldsymbol{j}, \boldsymbol{k}) \leftarrow [pZqX](\boldsymbol{i}, \boldsymbol{j}),\ a(\boldsymbol{j}, \boldsymbol{k}) & \text{for } (q, \varepsilon) \overset{a}{\mapsto} (r, Y), \\
[pXsY](\boldsymbol{i}, \boldsymbol{l}) \leftarrow [pXqY](\boldsymbol{i}, \boldsymbol{j}),\ [qYrZ](\boldsymbol{j}, \boldsymbol{k}),\ a(\boldsymbol{k}, \boldsymbol{l}) & \text{for } (r, Z) \overset{a}{\mapsto} (s, \varepsilon), \\
[pXrY](\boldsymbol{i}, \boldsymbol{k}) \leftarrow [pXqZ](\boldsymbol{i}, \boldsymbol{j}),\ a(\boldsymbol{j}, \boldsymbol{k}) & \text{for } (q, Z) \overset{a}{\mapsto} (r, Y),
\end{array}$$

where $\varepsilon(\boldsymbol{i}, \boldsymbol{j})$ should be read as $\boldsymbol{i} = \boldsymbol{j}$. Note the additional goal $[pZqX](\boldsymbol{i}, \boldsymbol{j})$ in the second class of rules.[20]

Note that different methods of converting PDAs to CFGs will similarly lead to tabular recognition algorithms for PDAs, given that a CFG is directly amenable to tabular recognition through its Datalog representation.

[20]Lang (1974) adds a special seed fact $[q_I\$q_I\$](0, 0)$ to get the process going.

Nederhof and Satta (2004) present a simplified version of Lang's (1974) algorithm for a restricted type of *stateless* extended PDA and show that many tabular parsing algorithms can be obtained from appropriate PDAs using this method. Nederhof and Satta (2004) call mappings from CFGs to PDAs *parsing strategies*. Thus, parsing schemata for tabular parsing can be viewed as the composition of parsing strategies and PDA tabulation techniques:



This is supposed to offer a more modular approach to tabular parsing.

Trying to view every parsing schema through the lens of PDAs may get awkward at times. It is also restrictive since it does not generalize to grammar formalisms beyond CFGs that have no good automata models. In contrast, understanding parsing schemata in terms of Datalog program transformations such as generalized supplementary magic-sets rewriting offers a general point of view that applies to all sorts of grammar formalisms that can be represented by Datalog, as we will demonstrate in later lectures.

Problems

3.1. Modify the Bottom-up-lc-chart procedure so that it will work with CFGs with ε-productions.

3.2. Let $G = (N, \Sigma, P, S)$ be a CFG generating $L \subseteq \Sigma^*$, and let $\mathbf{P}_G$ be the Datalog program directly representing it. Let $M = (Q, \Sigma, \delta, q_I, F)$ be a DFA recognizing $R \subseteq \Sigma^*$. Define a database D_M by

$$D_M = \{\, a(q, r) \mid \delta(q, a) = r \,\},$$

using states of M as database constants. Show that the following conditions are equivalent:

(i) $\mathbf{P}_G \cup D_M \vdash S(q_I, q)$ for some $q \in F$.

(ii) $L \cap R \neq \varnothing$.

3.3. Let $\mathbf{P}$ be the deduction system (Datalog program) for Earley's algorithm obtained from a CFG $G = (N, \Sigma, P, S)$. Let $M = (\Sigma, N \cup \Sigma, \delta, S, \varepsilon)$ be the stateless standard PDA representing the top-down recognizer for G. Let $A \in N$, $a_1 \dots a_n \in \Sigma^*$, and $D = \{ {\sim}S(0), a_1(0,1), \dots, a_n(\overline{n-1}, \overline{n})\}$. Show that the following are equivalent:

(i) $\mathbf{P} \cup D \vdash {\sim}A(\bar{i})$.

(ii) $(S, a_1 \dots a_n) \vdash_M^* (A\alpha, a_{i+1} \dots a_n)$ for some $\alpha \in (N \cup \Sigma)^*$.

3.4. Give a version of the left-corner transform on the basis of the arc-standard left-corner recognizer defined in Lecture 2. What is the relation with the left-corner transform defined in the text?

3.5. Let G be a CFG in Chomsky normal form, and consider the left-corner tranform G' of G. Show that G' can easily be converted to an equivalent CFG G'' in *quadratic Greibach normal form*, where each production is of the form

$$A \to aB_1 \dots B_n$$

where a is a terminal, $B_1, \dots, B_n$ are nonterminals, and $n \leq 2$.

3.6. Try to define the "bottom-up transform". That is, give a transformation of CFGs that yields an equivalent CFG on which the behavior of the top-down recognizer corresponds to the behavior of the bottom-up recognizer on the original CFG.

3.7. For a PDA $M = (Q, \Sigma, \Gamma, \delta, q_I, Z_I, Q_F, Z_F)$, define the set $T(M)$ of *accepting transition sequences* of M by

$$\begin{aligned} T(M) = \{ \tau_1 \dots \tau_n \in \delta^* \mid{} & M \text{ has an accepting computation } C_0 \vdash_M \cdots \vdash_M C_n \\ & \text{such that for each } i = 1, \dots, n, \ \tau_i \text{ sends } C_{i-1} \text{ to } C_i \}. \end{aligned}$$

$T(M)$ is a deterministic context-free language over the alphabet consisting of the transitions in δ. Give a context-free grammar G such that $L(G) = T(M)$.

References

Abiteboul, Serge, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Reading, Massachusetts: Addison-Wesley.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1968. Time and tape complexity of pushdown automaton languages. *Information and Control* 13:186–206.

Aho, Alfred V. and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing*. Englewood Cliffs, N.J.: Prentice Hall.

Beeri, Catriel and Raghu Ramakrishnan. 1991. On the power of magic. *Journal of Logic Programming* 10(3–4):255–299.

Chandra, Ashok K., Dexter C. Kozen, and Larry J. Stockmeyer. 1980. Alternation. *Journal of the Association for Computing Machinery* 28(1):114–133.

Chomsky, Noam. 1962. Context-free grammars and pushdown storage. In *Quarterly Progress Report no. 65*, pages 187–194. Cambridge, Mass.: MIT Research Laboratory of Electronics.

Cook, Stephen A. 1971. Characterization of pushdown machines in terms of time-bounded computers. *Journal of the Association for Computing Machinery* 18(1):4–18.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. Cambridge, Massachusetts: MIT Press.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*. Cambridge, Massachusetts: MIT Press.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Communications of the ACM* 13(2):94–102.

Floyd, Robert W. and Richard Beigel. 1994. *The Language of Machines*. New York: Freeman.

Gazdar, Gerald and Chris Mellish. 1989. *Natural Language Processing in Prolog*. Workingham, England: Addison-Wesley.

Hall, Patrick A. V. 1973. Equivalence between AND/OR graphs and context-free grammars. *Communications of the ACM* 16(7):444–445.

Harrison, Michael A. 1978. *Introduction to Formal Language Theory*. Reading, Massachusetts: Addison-Wesley.

Hays, David G. 1962. Automatic langauge-data processing. In H. Borko, ed., *Comupter Applications in the Behavioral Sciences*. Englewood Cliffs, NJ: Prentice Hall.

Hays, David G. 1967. *Introduction to Computational Linguistics*. New York: American Elsevier.

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

Kasami, T. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Kasami, T. 1966. An efficient recognition and syntax-analysis algorithm. Control Systems Laboratory Report R-257, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Kay, Martin. 1963. A parsing procedure. In C. M. Popplewell, ed., *Information Processing 1962*, pages 328–329. Amsterdam: North Holland.

Kay, Martin. 1986. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck-Jones, and B. L. Webber, eds., *Readings in Natural Language Processing*, pages 35–70. San Francisco, Calif.: Morgan Kaufmann.

Kay, Martin. 2000. David G. Hays. In W. J. Hutchins, ed., *Early Years in Machine Translation: Memoirs and Biographies of Pioneers*, pages 165–170. Amsterdam/Philadelphia: John Benjamins.

Kay, Martin. 2005. A life of language. *Computational Linguistics* 31(4):425–438.

Kleinberg, Jon and Éva Tardos. 2006. *Algorithm Design*. Boston, Massachusetts: Pearson/Addison-Wesley.

Kozen, Dexter C. 1997. *Automata and Computability*. New York: Springer.

Lang, Bernard. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269. London: Springer.

Leiss, Hans. 1990. On Kilbury's modification of Earley's algorithm. *ACM Transactions on Programming Languages and Systems* 12(4):610–640.

Ljunglöf, Peter and Mats Wirén. 2010. Syntactic parsing. In N. Indurkhya and F. J. Damerau, eds., *Handbook of Natural Language Processing*, pages 59–91. Boca Raton, FL: CRC Press, 2nd edn.

Nederhof, Mark-Jan and Giorgio Satta. 2004. Tabular parsing. In C. Martín-Vide, V. Mitrana, and G. Păun, eds., *Formal Languages and Applications*, pages 529–549. Berlin: Springer.

Nilsson, Nils J. 1982. *Principles of Artificial Intelligence*. Berlin: Springer.

Rosenkrantz, D.J. and P.M. Lewis, II. 1970. Deterministic left corner parser. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata*, pages 139–152.

Ruzzo, Walter L. 1980. Tree-size bounded alternation. *Journal of Computer and System Sciences* 21:218–235.

Samuelsson, Christer and Mats Wirén. 2000. Parsing techniques. In R. Dale, H. Moisl, and H. Somers, eds., *Handbook of Natural Language Processing*, pages 59–91. New York: Marcel Dekker.

Shieber, Stuart M., Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementations of deductive parsing. *Journal of Logic Programming* 24:3–36.

Sikkel, Klaas. 1997. *Parsing Schemata*. Berlin: Springer.

Sikkel, Klaas. 1998. Parsing schemeta and correctness of parsing algorithms. *Theoretical Computer Science* 199(1–2):87–103.

Simon, Richard and Richard C. T. Lee. 1971. On the optimal solutions to AND/OR series-parallel graphs. *Journal of the Association for Computing Machinery* 18(3):354–372.

Sippu, Seppo and Eljas Soisalon-Soininen. 1990. *Parsing Theory. Volume II: LR(k) and LL(k) Parsing*. Berlin: Springer.

Sipser, Michael. 2012. *Introduction to the Theory of Computation, Third Edition*. Boston: Cengage Learning.

Ullman, Jeffrey D. 1989a. Bottom-up beats top-down in Datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 140–149.

Ullman, Jeffrey D. 1989b. *Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies*. Rockville, M.D.: Computer Science Press.

Younger, Daniel H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control* 10:189–208.