CSCI 3313-10: Foundation of Computing

1 Overview

Foundation of Computing

- Theory of Computing

- Automata theory
- Computability
  - solvable vs. unsolvable problems
- Complexity
  - computationally easy vs. hard problems
- Formal language theory

Chomsky Hierarchy

- Type-3: Regular languages (RL); Finite state automata
- Type-2: Context-free languages (CFL); Pushdown automata
- Type-1: Context-sensitive languages (CSL); Linear-bound Turing machines
- Type-0: Recursively enumerable languages (REL); Turing machines

$$RL \subset CFL \subset CSL \subset REL$$

1.1 Mathematical Notations and Terminologies

- *sets:* element, member, subset, proper subset, finite set, infinite set, empty set, union, intersection, complement, power set, Cartesian product (cross product)
- *sequence, tuple:* k-tuple: a sequence with k elements
- *functions:* mapping, domain, co-domain, range, one-to-one function, onto function, one-to-one correcspondence
- *relation:*
  - reflexive: xRx
  - symmetric: $xRy \Rightarrow yRx$
  - transitive: $xRy \wedge yRz \Rightarrow xRz$
  - equivalence relation

- *graphs:*

- *strings, languages:*
  - alphabet: any non-empty finite set
  - string over an alphabet: a finite sequence of symbols from the alphabet
  - $|w|$: length of a string w ($w = w_1 w_2 \cdots w_n$, where $w \in \Sigma$, for an alphabet Σ)
  - empty string: ϵ
  - reverse of w: w^R
  - substring
  - concatenation

- *logic*

- *theorem, proof*
  - by construction, induction contradiction

2 Regular Languages

2.1 Finite State Automata

Definition: A finite state automaton (FSA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*.
2. Σ is a finite set called the *alphabet*.
3. $\delta : Q \times \Sigma \to Q$ is the *transition* function.
4. $q_0 \in Q$ is the *start* state.
5. $F \subseteq Q$ is the set of *accept* states.

Formal definition of computation:

Let $w = w_1 w_2 \cdots w_n$ be a string such that $w \in \Sigma$, and $M = (Q, \Sigma, \delta, q_0, F)$ be a FSA. Then, M *accepts* w if a sequence of states $r_0, r_1, \cdots, r_n \in Q$ exists with conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, 1, \cdots, n-1$, and
3. $r_n \in F$.

We say M *recognizes* A if $A = \{w \mid M \text{ accepts } w\}$.

A language is called a *regular language* if some FSA recognizes it.

2.2 Designing FSA

2.3 Regular Operations

Let A and B be languages. We define regular operations as follows:

union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$

star: $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Example:
$A = \{0, 1\}$, $B = \{a, b\}$:
$A \cup B = \{0, 1, a, b\}$
$A \circ B = \{0a, 0b, 1a, 1b\}$
$A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \cdots, 111, 0000, \cdots\}$

Theorem 2.1 *The class of regular languages is closed under the union operation, i.e., if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.*

Proof: Let A_1 and A_2 be regular languages. By definition, A_1 and A_2 are recognized by FSA M_1 and M_2, resp. Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$. We construct $M = (Q, \Sigma, \delta, q_0, F)$ from M_1 and M_2 such that

1. $Q = Q_1 \times Q_2$,
   i.e., $Q = \{(r_1, r_2) \mid r_1 \in Q_1, r_2 \in Q_2\}$
2. $\Sigma = \Sigma_1 \cup \Sigma_2$
3. $\delta((r_1, r_2), a) = (\delta(r_1, a), \delta(r_2, a))$
4. $q_0 = (q_1, q_2)$
5. $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\ \}$,
   i.e., $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that $F \neq F_1 \times F_2$.)

Example:
Let $L_1 = \{w \mid w \text{ has even number of 1's}\}$ and $L_2 = \{w \mid w \text{ contains 001 as a substring}\}$. Construct a FSA M for $L_1 \cup L_2$.

Theorem 2.2 *The class of regular languages are closed under intersection operation.*

Proof: Proof is same as above, except that $F = F_1 \times F_2$.

Example:
Let $L_1 = \{w \mid w \text{ has odd number of a's}\}$ and $L_2 = \{w \mid w \text{ has one b}\}$. Construct a FSA M for $L = L_1 \cap L_2$, i.e., $L = \{w \mid w \text{ has odd number of a's and one b. }\}$

2.4 Nondeterminism

Formal definition of non-deterministic FSA (NFA):
An NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states.
2. Σ is an alphabet.
3. $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition relation,
   where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $P(Q)$ is the power set of Q.
4. $q_0 \in F$ is the initial state.
5. $F \subseteq Q$ is the set of accept states.

2.5 Equivalence of NFA and DFA

Theorem 2.3 *Every NFA has an equivalent DFA.*

Proof: Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing language A. We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ as follows.

(i) First, assume that N does not have ϵ-transition.

1. $Q' = P(Q)$.
2. For $R \in P(Q)$, let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$
   (or, let $\delta'(R, a) = \cup\{\delta(r, a) \mid r \in R\}$.)
3. $q_0' = \{q_0\}$.
4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N \}$.

(ii) Next, assume that N contains ϵ-transitions. For any $R \in P(Q)$, let
$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along 0 or more } \epsilon \text{ arrow. } \}$
Let $\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R \}$. The rest are same as in case (i)

Example (i): Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA, where

1. $Q = \{q_0, q_1\}$
2. $\Sigma = \{0, 1\}$
3. $\delta(q_0, 0) = \{q_0\}$; $\delta(q_0, 1) = \{q_0, q_1\}$;
4. initial state $= q_0$
5. $F = \{q_1\}$

A DFA $M = (Q', \Sigma, \delta', q_0', F')$ that is equivalent to N is then constructed as:

1. $Q' = \{\{q_0\}, \{q_0, q_1\}\}$
2. $\Sigma = \{0, 1\}$
3. $\delta'(\{q_0\}, 0) = \{q_0\}$; $\delta'(\{q_0\}, 1) = \{q_0, q_1\}$; $\delta'(\{q_0, q_1\}, 0) = \{q_0\}$; $\delta'(\{q_0, q_1\}, 1) = \{q_0, q_1\}$
4. initial state $= \{q_0\}$
5. $F = \{\{q_0, q_1\}\}$

Example (ii): Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA, where

1. $Q = \{q_0, q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{a, b\}$
3. $\delta(q_0, \epsilon) = \{q_1\}$; $\delta(q_0, b) = \{q_2\}$; $\delta(q_1, \epsilon) = \{q_2, q_3\}$; $\delta(q_1, a) = \{q_0, q_4\}$; $\delta(q_2, b) = \{q_4\}$; $\delta(q_3, a) = \{q_4\}$; $\delta(q_4, \epsilon) = \{q_3\}$
4. initial state $= q_0$
5. $F = \{q_4\}$

Note that $E(q_0) = \{q_0, q_1, q_2, q_3\}$, $E(q_1) = \{q_1, q_2, q_3\}$, $E(q_2) = \{q_2\}$, $E(q_3) = \{q_3\}$, and $E(q_4) = \{q_3, q_4\}$. We then construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ by following the algorithm in (i) as follows:

1. $Q' = \{p_0, p_1, p_2, p_3, p_4\}$ where $p_0 = \{q_0, q_1, q_2, q_3\}$, $p_1 = \{q_0, q_1, q_2, q_3, q_4\}$, $p_2 = \{q_2, q_3, q_4\}$, $p_3 = \{q_3, q_4\}$, and $p_4 = \emptyset$ (or a trap state).
2. $\Sigma = \{a, b\}$
3. $\delta'(p_0, a) = p_1$; $\delta'(p_0, b) = p_2$; $\delta'(p_1, b) = p_2$; $\delta'(p_1, a) = p_1$; $\delta'(p_2, a) = p_3$; $\delta'(p_2, b) = p_3$; $\delta'(p_3, a) = p_3$; $\delta'(p_3, b) = p_4$; $\delta'(p_4, a) = p_4$, and $\delta'(o_4, b) = p_3$.
4. initial state $= p_0$
5. $F = \{p_1, p_2, p_3\}$.

2.6 Closure Properties of Regular Languages

Theorem 2.4 *Regular languages are closed under the following operations:*

(1) union

(2) intersection

(3) concatenation

(4) star operation (or Kleene star operation)

Note: We can construct an NFA N for each case and find a DFA M equivalent to N.

2.7 Regular Expressions

- To describe regular languages

Examples: $(0 \cup 1)0^*$, $(0 \cup 1) = (\{0\} \cup \{1\})$, $(0 \cup 1)^*$

Definition: We say R is a regular expression if R is

(1) a for some $a \in \Sigma$

(2) ϵ

(3) $\emptyset$

(4) $R_1 \cup R_2$, where R_1 and R_2 are regular expressions.

(5) $R_1 \circ R_2$, where $\circ$ is a concatenation operation, and R_1 and R_2 are regular expressions.

(6) $(R_1)^*$, where R_1 is a regular expression.

- recursive or inductive definition
- () may be omitted.
- $R^+ = RR^*$ or R^*R
- $R^+ \cup \epsilon = R^*$
- $R^k = R \circ R \circ \cdots \circ R$ (i.e., R is concatenated k times.)
- $L(R)$

Examples: 0^*10^*, $\Sigma^*1\Sigma^*$, $1^*(01^+)^*$, $(0 \cup \epsilon)1^* = 01^* \cup 1^*$, $(0 \cup \epsilon)(1 \cup \epsilon) = \{01, 0, 1, \epsilon\}$, $1^* \circ \emptyset = \emptyset$, $1 \circ \epsilon = 1^*$, $\emptyset^* = \{\epsilon\}$

2.8 Equivalence of Regular Expression and DFA

Recall: A language is regular if and only if a DFA recognizes it.

Theorem 2.5 *A language is regular if and only if some regular expression can describe it.*

Proof is based on the following two lemmas.

Lemma 2.1 *If a language L is described by a regular expression R, then it is a regular language, i.e., there is a DFA that recognizes L.*

Proof. We will convert R to an NFA N (equivalently a DFA).

(1) $R = a \Rightarrow L(R) = \{a\}$

(2) $R = \epsilon \Rightarrow L(R) = \{\epsilon\}$

(3) $R = \emptyset \Rightarrow L(R) = \emptyset$

(4) $R = R_1 \cup R_2 \Rightarrow$

(5) $R = R_1 \circ R_2 \Rightarrow$

(6) $R = R_1^* \Rightarrow$

Example: $R = (ab \cup a)^* \Rightarrow N$:

Lemma 2.2 *If L is a regular language, then it can be described by a regular expression.*

Proof: Reference: text, Lemma 1.60.

2.8.1 Alternate proof:

Since L is a regular language, there must be a DFA that recognizes L. We then apply the following result.

Lemma: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then there exists a regular expression E such that $L(E) = L(M)$, where $L(E)$ denotes the language represented by E.

Proof: Let $Q = \{q_1, \cdots, q_m\}$ such that q_1 is the start state of M. For $1 \leq i, j \leq m$ and $1 \leq k \leq m+1$, we let $R(i,j,k)$ denote the set of all strings in Σ^* that derive M from q_i to q_j without passing through any state numbered k or greater.

When $k = m+1$, it follows that

$$R(i,j,m+1) = \{x \in \Sigma^* \mid (q_i, x) \vdash_M^* (q_j, \epsilon)\}.$$

Therefore, $L(M) = \cup\{R(1,j,m+1) \; | q_j \in F\}$.

The crucial point is that each set $R(i,j,k)$ is regular, and hence so is $L(M)$. The proof is by induction on k. For $k = 1$, we have the following.

$$R(i,j,1) = \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{\epsilon\} \cup \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{if } i = j \end{cases}$$

Each of these sets is finite, and therefore regular. For $k = 1, \cdots, m$, provided that all the sets $R(i,j,k)$ have been defined, each set $R(i,j,k+1)$ can be defined in terms of previously defined languages as

$$R(i,j,k+1) = R(i,j,k) \cup R(i,k,k)R(k,k,k)^*R(k,j,k).$$

This equation states that to get from q_i to q_j without passing through a state numbered greater than k, M may either

(i) go from q_i to q_j without passing through a state numbered greater than $k-1$, or

(ii) go from q_i to q_k; then from q_k to q_k repeatedly; and then from q_k to q_j, in each case without passing through a state numbered greater than $k-1$.

Therefore, if each language $R(i,j,k)$ is regular, so is each language $R(i,j,k+1)$. This completes the induction. ■

2.9 Non-regular Languages (Pumping Lemma)

Review ...

- Let L be an arbitrary finite set. Is L a regular language?
- Give a regular expression for the set L_1 of non-negative integers.
  Let $\Sigma = \{0, 1, \cdots, 9\}$. Then, $L_1 = \{0\} \cup \{1, 2, \cdots, 9\} \circ \Sigma^*$.
- Give a regular expression for the set L_2 of non-negative integers that are divisible by 2.
  Then, $L_2 = L_1 \cap \Sigma^* \circ \{0, 2, 4, 6, 8\}$
- Give a regular expression for the set L_3 of integers that are divisible by 3.
  Then, $L_3 = L_1 \cap L(M)$, where
  M is defined as:
- Let $\Sigma = \{a, b\}$, and $L_4 \subseteq \Sigma^*$ be the set of odd length, containing an even # of a's.
  Then, $L_4 = L_5 \cap L_6$, where L_5 is the set of all strings of odd length, i.e., $L_5 = \Sigma(\Sigma\Sigma)^*$, and L_6 is the set of all strings with an even # of a's, i.e., $L_6 = b^*(ab^*ab^*)^*$.

Now, consider the following...

- $A_1 = \{0^n 1^n \mid n \geq 1\}$
- $A_2 = \{w \mid w$ has an equal number of occurrences of a's and b's.$\}$
- $A_3 = \{w \mid w$ has an equal number of occurrences of 01 and 10 as substrings.$\}$

Lemma 2.3 (Pumping Lemma for Regular Languages)
If A is a regular language, then there is a positive integer p called the pumping length where if s is any string in A of length at least p, then s may be divided into three substrings $s = xyz$ for some x, y, and z satisfying the following conditions:

(i) $|y| > 0$ $(|x|, |z| \geq 0)$

(ii) $|xy| \leq p$

(iii) for each $i \geq 0$, $xy^iz \in A$.

2.9.1 Non-regular languages

- $\{ww^R \mid w \in \{0, 1\}^*\}$.
- $\{ww \mid w \in \{0, 1\}^*\}$

- $\{a^n b a^m b a^{n+m} \mid n, m \geq 1\}$
- $\{w\overline{w} \mid w \in \{a,b\}^*$ where $\overline{w}$ stands for w with each occurrence of a replaced by b, and vice versa.$\}$
- $L = \{w \mid w$ has equal number of 0's and 1's $\}$
- $L = \{a^m b^n \mid m \neq n\}$

Answer *true* or *false*:

(a) Every subset of a regular language is regular.

(b) Every regular language has a subset that is regular.

(c) If L is regular, then so is $\{xy \mid x \in L$ and $y \notin L\}$

(d) $L = \{w \mid w = w^R\}$ is regular.

(e) If L is regular, then $L^R = \{w^R \mid w \in L\}$ is regular.

(f) $L = \{xyx^R \mid x, y \in \Sigma^*\}$ is regular.

(g) If L is regular, then $L_1 = \{w \mid w \in L$ and $w^R \in L\}$ is regular.

2.9.2 more non-regular languages proved by Pumping lemma

1. $L = \{a^{n^2} \mid n \geq 1\}$
2. $L = \{a^{2^n} \mid n \geq 1\}$
3. $L = \{a^q \mid q$ is a prime number.$\}$
4. $L = \{a^{n!} \mid n \geq 1\}$
5. $L = \{a^m b^n \mid m > n\}$
6. $L = \{a^m b^n \mid m < n\}$
7. $L = \{w \in \{a,b\}^* \mid n_a(w) = n_b(w)\}$
8. $L = \{w \in \{a,b\}^* \mid n_a(w) \neq n_b(w)\}$
9. $L = \{a^p b^q \mid p$ and q are prime numbers.$\}$
10. $L = \{a^{n^2} b^{m^2} \mid n, m \geq 1\}$
11. $L = \{w \in \{a,b\}^* \mid n_a(w)$ and $n_b(w)$ both are prime numbers$\}$
12. $L = \{a^{n!} b^{m!} \mid n, m \geq 1\}$

2.9.3 additional properties of regular languages

- Given two regular languages L_1 and L_2, describe an algorithm to determine if $L_1 = L_2$.
- There exists an algorithm to determine whether a regular language is empty, finite, or infinite.
- membership

3 Context Free Languages and Context Free Grammars

Definition. A context-free grammar (CFG) is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the variables (or non-terminals).
2. Σ is a finite set called the terminals.
3. R is a finite set of production rules such that
   $R : V \to (V \cup \Sigma)^*$.
4. $S \in V$ is a start symbol.

Examples of Context-Free Grammars

G_0: $E \to E + E \mid E * E \mid id$

G_1:

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE' \mid \epsilon \\
&T \to FT' \\
&T' \to *FT' \mid \epsilon \\
&F \to (E) \mid id
\end{aligned}
$$

G_2:

$$
\begin{aligned}
&E \to E + T \mid T \\
&T \to T * F \mid F \\
&F \to (E) \mid id
\end{aligned}
$$

G_3:

$$
\begin{aligned}
&E' \to E \\
&E \to E + T \mid T \\
&T \to T * F \mid F \\
&F \to (E) \mid id
\end{aligned}
$$

G_4:

$$
\begin{aligned}
&S' \to S \\
&S \to L = R \\
&S \to R \\
&L \to *R \\
&L \to id \\
&R \to L
\end{aligned}
$$

G_5:

$$
\begin{aligned}
&S' \to S \\
&S \to aAd \mid bBd \mid aBe \mid bAe \\
&A \to c \\
&B \to c
\end{aligned}
$$

3.1 Context Free Grammar

1. $L = \{a^n b^n \mid n \geq 0\}$

$$
\begin{array}{lcl}
S & \rightarrow & aSb \mid \epsilon
\end{array}
$$

2. $L = \{a^m b^n \mid m > n\}$

$$
\begin{array}{lcl}
S & \rightarrow & AC \\
C & \rightarrow & aCb \mid \epsilon \\
A & \rightarrow & aA \mid a
\end{array}
$$

3. $L = \{a^m b^n \mid m < n\}$

$$
\begin{array}{lcl}
S & \rightarrow & CB \\
C & \rightarrow & aCb \mid \epsilon \\
B & \rightarrow & bB \mid b
\end{array}
$$

4. $L = \{a^m b^n \mid m \neq n\}$

$$
\begin{array}{lcl}
S & \rightarrow & AC \mid CB \\
C & \rightarrow & aCb \mid \epsilon \\
A & \rightarrow & aA \mid a \\
B & \rightarrow & bB \mid b
\end{array}
$$

5. $L = \{w \in \{a, b\}^* \mid n_a(w) = n_b(w)\}$.

$$
\begin{array}{lcl}
S & \rightarrow & SS \mid aSb \mid bSa \mid \epsilon
\end{array}
$$

6. $L = \{w \in \{a, b\}^* \mid n_a(w) > n_b(w)\}$.

$$
\begin{array}{lcl}
S_0 & \rightarrow & AS \mid SAS \mid SA \\
S & \rightarrow & SS \mid SAS \mid aSb \mid bSa \mid \epsilon \\
A & \rightarrow & aA \mid a
\end{array}
$$

Proof: Note that any string generated by the above rules has more a's than b's. We next proceed to show that any string $w \in L$ can be generated by these rules. We first note that any string z such that $n_a(z) = n_b(z)$ must be split into substrings such that $z = z_1 z_2 \cdots z_l$

where (i) each z_j has equal number of a's and b's, (ii) the first and the last symbols of z_j are different, and (iii) any such z_j does not contain a substring that has the same number of a's and b's but the first and the last symbols are same. For example, $aabbab$ cannot be such a z_j since it contains $abba$, but $aababb$ can be such a z_j. It is then noted that for any $w \in L$, w can be denoted as:

$$w = a^{l_0} z_1 a^{l_1} z_2 a^{l_2} \cdots z_k a^{l_k},$$

where (1) each z_i satisfies the above three conditions (i) - (iii); (2) for each i, $0 \le i \le k$, $l_i \ge 0$; and (3) $l_0 + l_1 + \cdots + l_k > 0$. For example, $w = aaababbaaaabbaaa$ may be decomposed into $w = aa \cdot ab \cdot ab \cdot ba \cdot a \cdot aabb \cdot aaa$, where $l_0 = 2$, $z_1 = ab$, $l_1 = 0$, $z_2 = ab$, $l_2 = 0$, $z_3 = ba$, $l_3 = 1$, $z_4 = aabb$, and $l_4 = 3$.

From the start state S_0, one of the following three cases occurs: If $l_0 > 0$, $S_0 \Rightarrow AS$; else if $l_k > 0$, $S_0 \Rightarrow SA$; otherwise, $S_0 \Rightarrow SAS$. We then recursively apply $S \to SS$ or $S \to SAS$ such that a single S generates a substring z_j satisfying conditions (i)-(iii) above.

Consider the example above: $w = aaababbaaaabbaaa$. w is then split into $a^2 z_1 z_2 z_3 a^1 z_4 a^3$, and is generated as follows.

$S_0 \overset{S\to AS}{\Longrightarrow} AS \overset{S\to SS}{\Longrightarrow} ASS \overset{S\to SS}{\Longrightarrow} ASSS \overset{S\to SAS}{\Longrightarrow} ASSSAS$
$\overset{S\to SAS}{\Longrightarrow} ASSSASAS \overset{S\to\epsilon}{\Longrightarrow} ASSSASA \overset{*}{\Rightarrow} aaz_1z_2z_3az_4aaa$, which is $aaababbaaaabbaaa$.

Note: The following also work correctly. You can verify the correctness using the similar arguments.

$$\begin{array}{lcl} S & \to & RaR \mid aRR \mid RRa \\ R & \to & RaR \mid aRR \mid RRa \mid aRb \mid bRa \mid \epsilon \end{array}$$

7. $L = \{w \in \{a,b\}^* \mid n_a(w) \neq n_b(w)\}$.

   Note that $L = \{w \in \{a,b\}^* \mid n_a(w) > n_b(w) \text{ or } n_a(w) < n_b(w)\}$.

8. $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) = n_c(w)\}$.

$$S \to SS \mid aSc \mid cSa \mid bSc \mid cSb \mid \epsilon$$

9. $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) > n_c(w)\}$.

$$\begin{array}{lcl} S_0 & \to & TS \mid STS \mid ST \\ S & \to & SS \mid STS \mid aSc \mid cSa \mid bSc \mid cSb \mid \epsilon \\ T & \to & aT \mid bT \mid a \mid b \end{array}$$

10. $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) \neq n_c(w)\}$.

    Note that $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) > n_c(w) \text{ or } n_a(w) + n_b(w) < n_c(w)\}$.

11. $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) > 2n_c(w)\}$.

$$\begin{array}{lcl}
S_0 & \rightarrow & TS \mid STS \mid ST \\
S & \rightarrow & SS \mid STS \mid \epsilon \\
S & \rightarrow & SDDC \mid DSDC \mid DDSC \mid DDCS \\
 & & SDCD \mid DSCD \mid DCSD \mid DCDS \\
 & & SCDD \mid CSDD \mid CDSD \mid CDDS \\
D & \rightarrow & a \mid b \\
C & \rightarrow & c \\
T & \rightarrow & aT \mid bT \mid a \mid b
\end{array}$$

12. $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) < 2n_c(w)\}$.

$$\begin{array}{lcl}
S_0 & \rightarrow & TS \mid STS \mid ST \\
S & \rightarrow & SS \mid STS \mid \epsilon \\
S & \rightarrow & SDDC \mid DSDC \mid DDSC \mid DDCS \\
 & & SDCD \mid DSCD \mid DCSD \mid DCDS \\
 & & SCDD \mid CSDD \mid CDSD \mid CDDS \\
D & \rightarrow & a \mid b \\
C & \rightarrow & c \\
T & \rightarrow & cT \mid c
\end{array}$$

13. $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) \neq 2n_c(w)\}$.

    Note that $L = \{w \in \{a,b,c\}^* \mid n_a(w) + n_b(w) > 2n_c(w)$ or $n_a(w) + n_b(w) < 2n_c(w)\}$.

3.2 Chompsky Normal Form

Definition: A CFG is in *Chomsky Normal Form* if every rule is of the form

$$A \rightarrow BC$$

$$A \rightarrow a$$

where a is any terminal and A, B, and C are any non-terminal (i.e., variable) except that B and C may not be the start symbol. In addition, we permit the rule $S \rightarrow \epsilon$, where S is the start symbol.

Theorem 2.9 (pp. 107). Any context-free languages is generated by a context-free grammar in Chomsky normal form.

3.3 CYK Membership Algorithm for Context-Free Grammars

Let $G = (V, \Sigma, R, S)$ be a CFG in CNF, and consider a string $w = a_1 a_2 \cdots a_n$. We define substrings $w_{ij} = a_i \cdots a_j$ and subset $V_{ij} = \{A \in V \mid A \overset{*}{\Rightarrow} w_{ij}\}$ of V.

Clearly, $w \in L(G)$ if and only if $S \in V_{1n}$. To compute V_{ij}, we observe that $A \in V_{ii}$ if and only if R contains a production $A \rightarrow a_i$. Therefore, V_{ii} can be computed for all $1 \leq i \leq n$ by inspection of w and the production rules of G. To continue, notice that for $j > i$, A derives w_{ij} if and only if there is a production $A \rightarrow BC$ with $B \overset{*}{\Rightarrow} w_{ik}$ and $C \overset{*}{\Rightarrow} w_{k+1j}$ for some k with $i \leq k < j$. In other words,

$$V_{ij} = \cup_{k \in \{i, i+1, \cdots, j-1\}} \{A \mid A \rightarrow BC, \text{ with } B \in V_{ik}, C \in V_{k+1j}\}.$$

The above equation can be used to compute all the V_{ij} if we proceed in the following sequence:

1. Compute $V_{11}, V_{22}, \cdots, V_{nn}$
2. Compute $V_{12}, V_{23}, \cdots, V_{(n-1)n}$
3. Compute $V_{13}, V_{24}, \cdots, V_{(n-2)n}$

and so on.

Time Complexity: $O(n^3)$, where $n = |w|$.

Example: Consider a string $w = aabbb$ and a CFG G with the following production rules:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow BB \mid a \\ B &\rightarrow AB \mid b \end{aligned}$$

	j				
i	1	2	3	4	5
1	A	$\emptyset$	S, B	A	$\mathbf{S}, B$
2		A	S, B	A	B, S
3			B	A	S, B
4				B	A
5					B

Since $S \in V_{15}$, $w \in L(G)$.

3.4 Pushdown Automata

A *pushdown automaton* is a 6-tuples $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \ \rightarrow \ \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Note: An input is accepted only if (i) input is all read and (ii) the stack is empty.

3.4.1 PDA for CFL

- $L = \{0^n 1^n \mid n \geq 0\}$
- $L = \{a^i b^j c^k \mid i = j \text{ or } i = k, \text{ where } i, j, k \geq 0\}$
- $L = \{w \in \{a, b\}^* \mid n_a(w) = n_b(w)\}$
- $L = \{ww^R \mid w \in \{a, b\}^*\}$
- $L = \{a^n b^{2n} \mid n \geq 0\}$
- $L = \{wcw^R \mid w \in \{a, b\}^*\}$
- $L = \{a^n b^m c^{n+m} \mid n, m \geq 0\}$
- $L = \{a^n b^m \mid n \leq m \leq 3n\}$

3.5 Equivalence of PDA and CFG

Theorem 3.1 *A language L is a CFL if and only if some PDA recognizes L.*

3.6 Pumping Lemma for CFL

Let L be a CFL. Then, there exists a number p, called the pumping length, where for any string $w \in L$ with $|w| \geq p$, w may be divided into five substrings $w = uvxyz$ such that

1) $|vy| > 0$

2) $|vxy| \leq p$, and

3) for each $i \geq 0$, $uv^i xy^i z \in L$.

3.6.1 Non-Context Free Languages

$L = \{a^n b^n c^n \mid n \geq 0\}$

Let $w = a^p b^p c^p$ and apply the Pumping lemma.

$L = \{ww \mid w \in 0, 1^*\}$

(Try with $w = 0^p 1 0^p 1$. Pumping lemma is not working!)

Let $w = 0^p 1^p 0^p 1^p$ and apply Pumping lemma.

$L = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k \leq n\}$

Let $w = a^p b^p c^p$ and apply Pumping lemma.

$L = \{a^{n!} \mid n \geq 0\}$

(Recall: L is not regular.)

Let $w = a^{p!}$ and apply Pumping lemma.

$L = \{a^n b^j \ |n = j^2\}$.

Let $w = a^{p^2} b^p$ and apply Pumping lemma. We then have $w = uvxyz$ and three cases to consider.

(i) $vy = a^\alpha$ or $vy = b^\beta$. Let $i = 0$ and come up with a contradiction.

(ii) $v = a^\alpha b^\beta$ or $y = a^\alpha b^\beta$. Let $i = 2$ and come up with a contradiction.

(iii) $v = a^\alpha$ and $y = b^\beta$, where $\alpha \neq 0$ and $\beta \neq 0$.

Let's first consider $i = 0$. If $p^2 - \alpha \neq (p - \beta)^2$, then we are done. So assume that $p^2 - \alpha = (p - \beta)^2$, i.e., we assume $\alpha = 2p\beta - \beta^2$. We then consider $i = 2$. The number of a's in w^2 is $p^2 + \alpha = p^2 + 2p\beta - \beta^2$, and the number of b's in w^2 is $p + \beta$. Note that $p^2 + 2p\beta - \beta^2 \neq (p + \beta)^2$ since $\beta \neq 0$. Therefore, $p^2 + \alpha \neq (p + \beta)^2$, a contradiction to the Pumping lemma.

From (i) - (iii), we conclude that L cannot be a CFL.

$L = \{a^{r+s} \mid r$ and s are both prime numbers.$\}$

Let $w = a^{2+p}$ where p is a prime number that is larger than or equal to the pumping length. Then, by the Pumping lemma, $w = uvxyz$ where $v = a^\alpha$ and $y = a^\beta$. Consider $i = 2p + 1$. Then, $|w^{2p+1}| = 2 + p + 2p(\alpha + \beta) = 2 + p(1 + 2(\alpha + \beta))$, which is an odd number since $p(1+2(\alpha+\beta)$ is an odd number (odd * odd). However, $p(1+2(\alpha+\beta)$ is not a prime number; hence, w^{2p+1} cannot be in L. Consequently, L cannot be a CFL.

3.7 Closure Properties

- CFL's are closed under the union operation.
- CFL's are not closed under the intersection operation.
- CFL's are not closed under the complementation operation.
- CFL's are closed under the concatenation operation.

- CFL's are closed under the kleene star operation.
- The intersection of a CFL and a RL is a CFL.

3.8 Top-Down Parsing

3.8.1 Transform to Unambiguous Grammar

A grammar is called *ambiguous* if there is some sentence in its language for which there us more than one parse tree.

Example: $E \to E + E \mid E * E \mid id;$
$w \;=\; id + id * id.$

In general, we may not be able to determine which tree to use. In fact, determining whether a given arbitrary CFG is ambiguous or not is undecidable.

Solution:

(a) Transform the grammar to an equivalent unambiguous one, or

(b) Use *disambiguating rule* with the ambiguous grammar to specify, for ambiguous cases, which parse tree to use.

if then else statement

G_1: $stmt \to$ if exp then $stmt$ |
if exp then $stmt$ else $stmt$

For an input "if E_1 then if E_2 then S_1 else S_2," two parse trees can be constructed; hence, G_1 is ambiguous. An unambiguous grammar G_2 which is equivalent to G_1 can be constructed as follows:

G_2: *stmt* → *matched_stmt* |
unmatched_stmt
matched_stmt → if *exp* then *matched_stmt* else *matched_stmt* |
other_stmt
unmatched_stmt → if *exp* then *stmt* |
if *exp* then *matched_stmt* else *unmatched_stmt*

3.8.2 Left-factoring and Removing left recursions

Consider the following grammar G_1 and a token string $w = bede$.

$$
\begin{aligned}
G_1: \quad & S \rightarrow ee \mid bAc \mid bAe \\
& A \rightarrow d \mid eA
\end{aligned}
$$

Since the initial b is in two production rules, $S \rightarrow bAc$ and $S \rightarrow bAe$, the parser cannot make a correct decision without backtracking. This problem may be solved to redesign the grammar as shown in G_2.

$$
\begin{aligned}
G_2: \quad & S \rightarrow ee \mid bAQ \\
& Q \rightarrow c \mid e \\
& A \rightarrow d \mid eA
\end{aligned}
$$

In G_2, we have factored out the common prefix bA and used another non-terminal symbol Q to permit the choice between the final c and a. Such a transformation is called as *left factorization* or *left factoring*.

Now, consider the following grammar G_3 and consider a token string $w = id + id + id$.

$$
\begin{aligned}
G_3: \quad & E \rightarrow E + T \mid T \\
& T \rightarrow T * F \mid F \\
& F \rightarrow id \mid (E)
\end{aligned}
$$

A top-down parser for this grammar will start by expanding E with the production $E \rightarrow E + T$. It will then expand E in the same way. In the next step, the parser should expand E by $E \rightarrow T$ instead of $E \rightarrow E + T$. But there is no way for the parser to know which choice it should make. In general, there is no solution to this problem as long as the grammar has productions of the form $A \rightarrow A\alpha$, called *left-recursive* productions. The solution to this problem is to rewrite the grammar in such a way to eliminate the left recursions. There are two types of left recursions: *immediate left recursions*, where the productions are of the form $A \rightarrow A\alpha$, and *non-immediate left recursions*, where the productions are of the form $A \rightarrow B\alpha$; $B \rightarrow A\beta$. In the latter case, A will use $B\alpha$, and B will use $A\beta$, resulting in the same problem as the immediate left recursions have.

We now have the following formal definition: "A grammar is *left-recursive* if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α."

Removing immediate left recursions:

Input: $A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
Output: $A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$

Consider the above example G_3 in which two productions have left recursions. Applying the above algorithm to remove immediate left recursions, we have

(i) $E \to E + T \mid T$

$$\Rightarrow \quad E \to TE'$$
$$E' \to +TE' \mid \epsilon$$

(ii) $T \to T * F \mid F$

$$\Rightarrow \quad T \to FT'$$
$$T' \to *FT' \mid \epsilon$$

Now, we have following grammar G_4 which is equivalent to G_3:

$$\begin{aligned} G_4 : \quad & E \to TE' \\ & E' \to +TE' \mid \epsilon \\ & T \to FT' \\ & T' \to *FT' \mid \epsilon \\ & F \to (E) \mid id \end{aligned}$$

The following is an algorithm for eliminating all left recursions including non-immediate left recursions.

Algorithm: Eliminating left recursioin.
Input: Grammar G with no cycles or ϵ-productions.
Output: An equivalent grammar with no left recursion.

1. Arrange the nonterminals in some order $A_1, A_2, \cdots, A_n$.
2. **for** $i = 1$ **to** n **begin**
    - **for** $j = 1$ **to** $i - 1$ **do begin**
        - replace each production of the form $A_i \to A_j\gamma$
            - by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$.
            - where $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current A_j-productions;
    - **end**
    - eliminate the immediate left recursion among the A_i-productions

    end

end.

Examples

EXAMPLE 1: Consider the following example:

$$
\begin{array}{ll}
G: & S \to Ba \mid b \\
 & B \to Bc \mid Sd \mid e
\end{array}
$$

Let $A_1 = S$ and $A_2 = B$. We then have,

$$
\begin{array}{ll}
G: & A_1 \to A_2a \mid b \\
 & A_2 \to A_2c \mid A_1d \mid e
\end{array}
$$

(i) i=1:

$A_1 \to A_2a \mid b$, OK

(ii) i=2:

$A_2 \to A_1d$ is replace by $A_2 \to A_2ad \mid bd$

Now, G becomes

$$
\begin{array}{ll}
G: & A_1 \to A_2a \mid b \\
 & A_2 \to A_2c \mid A_2ad \mid bd \mid e
\end{array}
$$

By eliminating immediate recursions in A_2-productions, we have

(i) $A_2 \to A_2c \mid bd \mid e$ are replaced by

$$A_2 \to bdA_3$$
$$A_2 \to eA_3$$
$$A_3 \to cA_3 \mid \epsilon$$

(ii) $A_2 \to A_2ad \mid bd \mid e$ are replaced by

$$A_2 \to bdA_4$$
$$A_2 \to eA_4$$
$$A_4 \to adA_4 \mid \epsilon$$

(i) and (ii) can be combined as

$$A_2 \to bdA_3 \mid eA_3$$
$$A_3 \to cA_3 \mid adA_3 \mid \epsilon$$

Therefore, we have

$S \to Ba \mid b$
$B \to bdD \mid eD$
$D \to cD \mid adD \mid \epsilon$

3.8.3 First and Follow Sets

Consider every string derivable from some sentential form α by a leftmost derivation. If $\alpha \overset{*}{\Longrightarrow} \beta$, where β begins with some terminal, then that terminal is in $FIRST(\alpha)$.

Algorithm: Computing $FIRST(A)$.

1. If A is a terminal, $FIRST(A) = \{A\}$.
2. If $A \to \epsilon$, add ϵ to $FIRST(A)$.
3. if $A \to Y_1 Y_2 \cdots Y_k$, then

 for $i = 1$ **to** $k - 1$ **do**

  if $[\epsilon \in FIRST(Y_1) \cap FIRST(Y_2) \cap \cdots \cap FIRST(Y_{i-1})]$ (i.e., $Y_1 Y_2 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$) **and**

   $a \in FIRST(Y_i)$, then add a to $FIRST(A)$.

 end

 if $\epsilon \in FIRST(Y_1) \cap \cdots \cap FIRST(Y_k)$, then add ϵ to $FIRST(A)$.

end.

Now, we define $FOLLOW(A)$ as the set of terminals that cam come right after A in any sentential form of $L(G)$. If A comes at the end, then $FOLLOW(A)$ includes the end marker \$.

Algorithm: Computing $FOLLOW(B)$.

1. \$ is in $FOLLOW(S)$.
2. if $A \to \alpha B \beta$, then $FIRST(\beta) - \{\epsilon\} \subseteq FOLLOW(B)$.
3. if $A \to \alpha B$ or $A \to \alpha B \beta$ where $\epsilon \in FIRST(\beta)$ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$),

 $FOLLOW(A) \subseteq FOLLOW(B)$

end.

Note: In Step 3, $FOLLOW(B) \not\subseteq FOLLOW(A)$. To prove this, consider the following example: $S \to Ab \mid Bc$; $A \to aB$; $B \to c$. Clearly, $c \in FOLLOW(B)$ but $c \notin FOLLOW(A)$.

EXAMPLE:

For the grammar G_4,

$$
\begin{aligned}
G_4: \quad & E \to TE' \\
& E' \to +TE' \mid \epsilon \\
& T \to FT' \\
& T' \to *FT' \mid \epsilon \\
& F \to (E) \mid id
\end{aligned}
$$

$FIRST(E) = FIRST(T) = FIRST(F) = \{(, \mathbf{id}\}$.
$FIRST(E') = \{+, \epsilon\}$.

$FIRST(T') = \{*, \epsilon\}$.
$FOLLOW(E) = FOLLOW(E') = \{), \$\}$.
$FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$.
$FOLLOW(F) = \{+, *,), \$\}$.

3.8.4 Constructing a predictive parser

Algorithm: Predictive parser contruction.
Input: Grammar G.
Output: Parsing table M.

1. for each $A \to \alpha$, do Steps 2 & 3.
2. for each terminal $a \in FIRST(\alpha)$,
   add $A \to \alpha$ to $M[A, a]$.
3. 3.1 if $\epsilon \in FIRST(\alpha)$,
      add $A \to \alpha$ to $M[A, b]$ for each terminal $b \in FOLLOW(A)$.
   3.2 if $\epsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$,
      add $A \to \alpha$ to $M[A, \$]$.

end.

EXAMPLE:

$$
\begin{aligned}
G_4: \quad & E \to TE' \\
& E' \to +TE' \mid \epsilon \\
& T \to FT' \\
& T' \to *FT' \mid \epsilon \\
& F \to (E) \mid id
\end{aligned}
$$

	Input symbol					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Stack Operation

Stack	Input	Action
$\$E$	$id + id * id\$$	$E \to TE'$
$\$E'T$	$id + id * id\$$	$T \to FT'$
$\$E'T'F$	$id + id * id\$$	$F \to id$
$\$E'T'id$	$id + id * id\$$	match
$\$E'T'$	$+id * id\$$	$T' \to \epsilon$
$\$E'$	$+id * id\$$	$E' \to +TE'$
$\$E'T+$	$+id * id\$$	match
$\$E'T$	$id * id\$$	$T \to FT'$
$\$E'T'F$	$id * id\$$	$F \to id$
$\$E'T'id$	$id * id\$$	match
$\$E'T'$	$*id\$$	$T' \to *FT'$
$\$E'T'F*$	$*id\$$	match
$\$E'T'F$	$id\$$	$F \to$
$\$E'T'id$	$id\$$	match
$\$E'T'$	$\$$	$T' \to \epsilon$
$\$E'$	$\$$	$E' \to \epsilon$
$\$E$	$\$$	accept

3.8.5 Properties of LL(1) Grammars

A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

Properties:

1. No ambiguous or left-recursive grammar can be LL(1).

2. A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions, the following conditions hold:

2.1 For any terminal a, there exist no derivations that $\alpha \stackrel{*}{\Rightarrow} a\alpha'$ and $\beta \stackrel{*}{\Rightarrow} a\beta'$.

2.2 Either α or β, but not both, can derive ϵ.

2.3 If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in $FOLLOW(A)$.

Proof of Condition 2.2: Suppose $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and $\beta \stackrel{*}{\Rightarrow} \epsilon$. Consider $S \stackrel{*}{\Rightarrow} \gamma_1 A \gamma_2$. Then, two possibilities exist: $S \stackrel{*}{\Rightarrow} \gamma_1 A \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \alpha \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \gamma_2$ and $S \stackrel{*}{\Rightarrow} \gamma_1 A \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \beta \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \gamma_2$. G must be then ambiguous.

Proof of Condition 2.3: Suppose $\beta \stackrel{*}{\Rightarrow} \epsilon$ and $\alpha \stackrel{*}{\Rightarrow} a\alpha'$, where $a \in FOLLOW(A)$. Also, assume that $\gamma_2 \stackrel{*}{\Rightarrow} a\gamma_2'$. We then have two possibilities: (i) $S \stackrel{*}{\Rightarrow} \gamma_1 A \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \alpha \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 a \alpha' \gamma_2$, and (ii) $S \stackrel{*}{\Rightarrow} \gamma_1 A \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \beta \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 \gamma_2 \stackrel{*}{\Rightarrow} \gamma_1 a \gamma_2'$. Hence, after taking care of the input tokens corresponding to γ_1, the parser cannot make a clear choice between the two productions $A \to \alpha$ and $A \to \beta$.

3.9 Bottom-Up Parsing

3.9.1 SLR Parser

Computation of Closure

If I is a set of items for a grammar G, then $closure(I)$ is the set of items constructed from I by the two rules.

1. Initially, every item in I is added to $closure(I)$.
2. If $A \to \alpha \cdot B\beta$ is in $closre(I)$ and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to I, if it is not already in I. We apply this rule until no more new items can be added to $closure(I)$.

function $closure(I)$:
begin
  $J = I$;
  repeat
    for each item $A \to \alpha \cdot B\beta$ in J and each production
      $B \to \gamma$ of G such that $B \to \cdot\gamma$ is not in J **do**
        add $B \to \cdot\gamma$ to J
  until no more items can be added to J
  return
end

We are now ready to give the algorithm to construct C, the canonical collection of stes of $LR(0)$ items for an augmenting grammar G'.

procedure $items(G')$:
begin
  $C = \{closure(\{[S' \to \cdot S]\})\}$;
  repeat
    for each set of items I in C and each grammar symbol X
      such that $goto(I, X)$ is not empty and not in C **do**
        add $goto(I, X)$ to C
  until no more sets of items can be added to C
end

Constructing SLR Parsing Table

Algorithm: Constructing an SLR parsing table.
Input: An augmenting grammar G'.
Output: The SLR parsing table functions *action* and *goto* for G'.

1. Construct $C = \{I_0, \cdots, I_n\}$, the collection of sets of LR(0) items for G'.
2. State i constructed from I_i. The parsing actions for state i are determined as follows:
   a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $goto(I_i, a) = I_j$, then set $action[i, a]$ to "shift j." Here a must be a terminal.
   b) If $[A \to \alpha\cdot]$ is in I_i, then set $action[i, a]$ to "reduce $A \to \alpha$" for all a in $FOLLOW(A)$; here A may not be S'.
   c) If $[S' \to S\cdot]$ is in I_i, then set $action[i, \$]$ to "accept."

   If any conflicting actions are generated by the above rules, we say the grammar is not SLR(0). The algorithm fails to produce a parser in this case.
3. The *goto* transitions for state i are constructed for all nonterminals A using the rule: If $goto(I_i, A) = I_j$, then $goto[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is teh one constructed from the set of items containing $[S' \to \cdot S]$.

end.

Example

Consider the following grammar G:

(0)	$E' \to E$
(1)	$E \to E + T$
(2)	$E \to T$
(3)	$T \to T * F$
(4)	$T \to F$
(5)	$F \to (E)$
(6)	$F \to id$

The canonical LR(0) collection for G is:

$I_0:$
$$E' \rightarrow \cdot E$$
$$E \rightarrow \cdot E + T$$
$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot id$$

$I_1:$
$$E' \rightarrow E \cdot$$
$$E \rightarrow E \cdot +T$$

$I_2:$
$$E \rightarrow T \cdot$$
$$T \rightarrow T \cdot *F$$

$I_3:$
$$T \rightarrow F \cdot$$

$I_4:$
$$F \rightarrow (\cdot E)$$
$$E \rightarrow \cdot E + T$$
$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot id$$

$I_5:$
$$F \rightarrow id \cdot$$

$I_6:$
$$E \rightarrow E + \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot id$$

$I_7:$
$$T \rightarrow T * \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot id$$

$I_8:$
$$F \rightarrow (E \cdot)$$
$$E \rightarrow E \cdot +T$$

$I_9:$
$$E \rightarrow E + T \cdot$$
$$T \rightarrow T \cdot *F$$

I_{10}
$$T \rightarrow T * F \cdot$$

$I_{11:}$
$$F \rightarrow (E) \cdot$$

The transition for viable prefixes is:

I_0: $goto(I_0, E) = I_1$; $goto(I_0, T) = I_2$; $goto(I_0, F) = I_3$; $goto(I_0, () = I_4$; $goto(I_0, id) = I_5$;

I_1: $goto(I_1, +) = I_6$;

I_2: $goto(I_2, *) = I_7$;

I_4: $goto(I_4, E) = I_8$; $goto(I_4, T) = I_2$; $goto(I_4, F) = I_3$; $goto(I_4, () = I_4$;

I_6: $goto(I_6, T) = I_9$; $goto(I_6, F) = I_3$; $goto(I_6, () = I_4$; $goto(I_6, id) = I_5$;

I_7: $goto(I_7, F) = I_{10}$; $goto(I_7, () = I_4$; $goto(I_7, id) = I_5$;

I_8: $goto(I_8,)) = I_{11}$; $goto(I_8, +) = I_6$;

I_9: $goto(I_9, *) = I_7$;

The FOLLOW set is: $FOLLOW(E') = \{\$\}$; $FOLLOW(E) = \{+,), \$\}$; $FOLLOW(T) = FOLLOW(F) = \{+,), \$, *\}$.

State	*action*						*goto*		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The moves of the SLR parser on input $id * id + id$ is:

Step	Stack	Input	ACtion
(1)	0	id * id + id $	shift
(2)	0id5	*id+id$	reduce by $F \to id$
(3)	0F3	*id+id$	reduce by $T \to F$
(4)	0T2	*id+id$	shift
(5)	0T2*7	id+id$	shift
(6)	0T2*7id5	+id$	reduce by $F \to id$
(7)	0T2*7F10	+id$	reduce by $T \to T * F$
(8)	0T2	+id$	reduce by $E \to T$
(9)	0E1	+id$	shift
(10)	0E1+6	id$	shift
(11)	0E1 + 6id5	$	reduce by $F \to id$
(12)	0E1+6F3	$	reduce by $T \to F$
(13)	0E1+6T9	$	reduce by $E \to E + T$
(14)	0E1	$	accept

3.9.2 Canonical LR(1) Parser

Consider the following grammar G with productions:

$$
\begin{aligned}
&S' \to S \\
&S \to L = R \\
&S \to R \\
&L \to *R \\
&L \to id \\
&R \to L
\end{aligned}
$$

Let's construct the canonical sets of LR(0) items for G:

$$
\begin{array}{ll}
I_0 : S' \to \cdot S & I_5 : L \to id\cdot \\
\quad S \to \cdot L = R & \\
\quad S \to \cdot R & I_6 : S \to L = \cdot R \\
\quad L \to \cdot * R & \quad R \to \cdot L \\
\quad L \to \cdot id & \quad L \to \cdot * R \\
\quad R \to \cdot L & \quad L \to \cdot id \\
 & \\
I_1 : S' \to S\cdot & I_7 : L \to *R\cdot \\
 & \\
I_2 : S \to L\cdot = R & I_8 : R \to L\cdot \\
\quad R \to L\cdot & \\
 & \\
I_3 : S \to R\cdot & I_9 : S \to L = R\cdot \\
 & \\
I_4 : L \to * \cdot R & \\
\quad R \to \cdot L & \\
\quad L \to \cdot * L & \\
\quad L \to \cdot id &
\end{array}
$$

Note that $= \in FOLLOW(R)$ since $S \Rightarrow L = R \Rightarrow *R = R$. Consider the state I_2 and the input symbol is "=." From $[R \to L\cdot]$, the parser will reduce by $R \to L$ since $= \in FOLLOW(R)$. But due to $[S \to L\cdot = R]$, it will try to shift the input as well, a conflict. Therefore, this grammar G cannot be handled by the SLR(0) parser. In fact, G can be parsed using the canonical-LR(1) parser that will be discussed next.

Construction of LR(1) Items

Let G' be an augmented grammar of G.

function $closure(I)$:
begin
    repeat
        for each item $[A \to \alpha \cdot B\beta, a]$ in I,
            each production $B \to \gamma$ in G',
            and each terminal b in $FIRST(\beta a)$
            such that $[B \to \cdot\gamma, b]$ is not in I **do**
                add $[B \to \cdot\gamma, b]$ to I;
    until no more items can be added to I
    return I
end

function $goto(I, X)$:
begin
    let J be the set of items $[A \to \alpha X \cdot \beta, a]$ such that
        $[A \to \alpha \cdot X\beta, a]$ is in I;
    return $closure(J)$
end

procedure $items(G')$:
begin
    $C = \{closure(\{[S' \to \cdot S, \$]\})\}$;
    repeat
        for each set of items I in C and each grammar symbol X
            such that $goto(I, X)$ is not empty and not in C **do**
                add $goto(I, X)$ to C
    until no more sets of items can be added to C
end

Construction of canonical-LR(1) parser

Algorithm: Constructing a canonical LR(1) parsing table.
Input: An augmenting grammar G'.
Output: The canonical LR(1) parsing table functions *action* and *goto* for G'.

1. Construct $C = \{I_0, \cdots, I_n\}$, the collection of sets of LR(1) items for G'.
2. State i constructed from I_i. The parsing actions for state i are determined as follows:
   a) If $[A \to \alpha \cdot a\beta,\ b]$ is in I_i and $goto(I_i, a) = I_j$, then set $action[i, a]$ to "shift j."
      Here a must be a terminal.
   b) If $[A \to \alpha\cdot,\ a]$ is in I_i, then set $action[i, a]$ to "reduce $A \to \alpha$";
      here A may not be S'.
   c) If $[S' \to S\cdot,\ \$]$ is in I_i, then set $action[i, \$]$ to "accept."

   If any conflicting actions are generated by the above rules, we say the grammar is not to be LR(1).
   The algorithm fails to produce a parser in this case.
3. The *goto* transitions for state i are constructed for all nonterminals A using the rule:
   If $goto(I_i, A) = I_j$, then $goto[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S,\ \$]$.

end.

Construction of LALR Parsing Table

Algorithm: Constructing an LALR parsing table.
Input: A grammar G.
Output: The LALR parsing table for G.

1. Construct $C = \{I_0, \cdots, I_n\}$, the collection of sets of LR(1) items for G.
2. Final all sets having the same core, and replace these sets by their union.
3. Let $C' = \{J_1, J_2, \cdots, J_m\}$ be the resulting sets of LR(1) items.
   Action table is constructed in the same manner as in Algorithm for Canonical LR(1) parsing table.
4. goto table is constructed as follows.
   Note that if $J_q = I_1 \cup I_2 \cup \cdots \cup I_k$, and for a non-terminal X,
   $goto(I_1, X) = J_{p_1}$, $goto(I_2, X) = J_{p_2}$, $\cdots$, $goto(I_k, X) = J_{p_k}$,
   then make $goto(J_q, X) = s$ where $s = J_{p_1} \cup J_{p_2} \cup \cdots \cup J_{p_k}$.
   (Note that $J_{p_1}, \cdots, J_{p_k}$ all have the same core.) **end.**

Example 1: Consider the following grammar G'.

(0) $S' \to S$
(1) $S \to L = R$
(2) $S \to R$
(3) $L \to *R$
(4) $L \to id$
(5) $R \to L$

The canonical LR(1) collection for G' is:

$I_0:$
$S' \to \cdot S,\ \$$
$S \to \cdot L = R,\ \$$
$S \to \cdot R,\ \$$
$L \to \cdot * R,\ =$
$L \to \cdot id,\ =$
$R \to \cdot L,\ \$$
$L \to \cdot * R,\ \$$
$L \to \cdot id,\ \$$

$I_1:$
$S' \to S\cdot,\ \$$

$I_2:$
$S \to L \cdot = R,\ \$$
$R \to L\cdot,\ \$$

$I_3:$
$S \to R\cdot,\ \$$

$I_4:$
$L \to * \cdot R,\ =$
$L \to * \cdot R,\ \$$
$R \to \cdot L,\ =/\$$
$L \to \cdot * R,\ =/\$$
$L \to \cdot id,\ =/\$$

$I_5:$
$L \to id\cdot,\ =/\$$

$I_6:$
$S \to L = \cdot R,\ \$$
$R \to \cdot L,\ \$$
$L \to \cdot * R,\ \$$
$L \to \cdot id,\ \$$

$I_7:$
$L \to *R\cdot,\ =/\$$

$I_8:\quad R \to L\cdot,\ =/\$$

$I_9:\quad S \to L = R\cdot,\ \$$

$I_{10}:\quad R \to L\cdot,\ \$$

$I_{11}:\quad L \to * \cdot R,\ \$$
$\quad\quad R \to \cdot L,\ \$$
$\quad\quad L \to \cdot * R,\ \$$
$\quad\quad L \to \cdot id,\ \$$

$I_{12}:\quad L \to id\cdot,\ \$$

$I_{13}:\quad L \to *R\cdot,\ \$$

Example 2:

Consider the following grammar G':

(0) $S' \to S$
(1) $S \to CC$
(2) $C \to cC$
(3) $C \to d$

The canonical LR(1) collection for G' is:

$I_0:\quad S' \to \cdot S,\ \$$
$\quad\quad S \to \cdot CC,\ \$$
$\quad\quad C \to \cdot cC,\ c/d$
$\quad\quad C \to \cdot d,\ c/d$

$I_1:\quad S' \to S\cdot,\ \$$

$I_2:\quad S \to C \cdot C,\ \$$
$\quad\quad C \to \cdot cC,\ \$$
$\quad\quad C \to \cdot d,\ \$$

$I_3:\quad C \to c \cdot C,\ c/d$
$\quad\quad C \to \cdot cC,\ c/d$
$\quad\quad C \to \cdot d,\ c/d$

$I_4 : C \to d\cdot,\ c/d$

$$
\begin{aligned}
I_5 : \quad & S \to CC\cdot,\ \$
\end{aligned}
$$

$$
\begin{aligned}
I_6 : \quad & C \to c \cdot C,\ \$ \\
& C \to \cdot cC,\ \$ \\
& C \to \cdot d,\ \$
\end{aligned}
$$

$$
\begin{aligned}
I_7 : \quad & C \to d\cdot,\ \$
\end{aligned}
$$

$I_8 : C \to cC\cdot,\ c/d$

$$
\begin{aligned}
I_9 : \quad & C \to cC\cdot,\ \$
\end{aligned}
$$

The transition for viable prefixes is:

I_0: $goto(I_0, S) = I_1$; $goto(I_0, C) = I_2$; $goto(I_0, c) = I_3$; $goto(I_0, d) = I_4$;

I_2: $goto(I_2, C) = I_5$; $goto(I_2, c) = I_6$; $goto(I_2, d) = I_7$;

I_3: $goto(I_3, c) = I_3$; $goto(I_3, d) = I_4$; $goto(I_3, C) = I_8$;

I_6: $goto(I_6, C) = I_9$;

A. Canonical-LR(1) parsing table

State	*action*			*goto*	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

B. LALR(1) parsing table

State	*action*			*goto*	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Note on LALR Parsing Table

Suppose we have an LR(1) grammar, that is, one whose sets of LR(1) items produce no parsing action conflicts. If we replace all states having the same core with their union, it is possible that the resulting union wil have a conflict, but it is unlikely for the following reasons.

Suppose in the union there is a conflict on lookahead a because there is an item $[B \to \beta \cdot a\gamma,\ b]$ calling for a reduction by $A \to \alpha$, and there is another item $[B \to \beta \cdot a\gamma,\ b]$ calling for a shift. Then, some set of items from which the union was formed has item $[A \to \alpha\cdot,\ a]$, and since the cores of all these states are the same, it must have an item $[B \to \beta \cdot a\gamma,\ c]$ for some c. But then this state has the same shift/reduce conflict on a, and the grammar was not LR(1) as we assumed. Thus, the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states, because shift actions depend only on core, not the lookahead.

It is possible, however, that a merger will produce a reduce/reduce conflict as the following example shows.

Example:

$$
\begin{aligned}
&S' \to S \\
&S \to aAd \mid bBd \mid aBe \mid bAe \\
&A \to c \\
&B \to c
\end{aligned}
$$

which generates the four strings acd, ace, bcd, bce. This grammar can be checked to be LR(1) by constructing the sets of items. Upon doing so, we find the set of items $\{[A \to c\cdot,\ d],\ [B \to c\cdot,\ e]\}$ valid for viable prefix ac and $\{[A \to c\cdot,\ e],\ [B \to c\cdot,\ d]\}$ valid for bc. Neither of these sets generates a conflict, and their cores are the same. However, their union, which is

$$
\begin{aligned}
&A \to c\cdot,\ d/e \\
&B \to c\cdot,\ d/e
\end{aligned}
$$

generates a reduce/reduce conflict, since reduction by both $A \to c$ and $B \to c$ are called for on input d and e.

4 Turing Machine

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. where

$\Sigma \subseteq \Gamma$

$\delta : Q \times \Gamma \ \rightarrow \ Q \times \Gamma \times \{L, R\}$

$q_{accept} \neq q_{reject}$

- L is Turing-decidable if some TM decides it (always halts with *accept* or *reject*).
- L is Turing-recognizable if some TM recognizes it (*accept, reject,* or *loop*).

Examples of Turing-decidable languages:

1. $L = \{w \mid |w| \text{ is a multiple of three. } \}$
2. $L = \{a^n b^m \mid n, m \geq 1, n \neq m\}$
3. $L = \{a^n b^n c^n \mid n \geq 1\}$
4. $L = \{ww \mid w \in \{a, b\}^*\}$
5. $L = \{a^{2^n} \mid n \geq 1\}$
6. $L = \{a^{n^2} \mid n \geq 1\}$
7. $L = \{a^i b^j c^k \mid i \cdot j = k\}$
8. $L = \{a^n \mid n \text{ is a prime number. } \}$

Hilbert's 10th problem:
Let $D = \{P \mid P \text{ is a polynomial with an integral root. } \}$ Is D decidable?

- D is not Turing-decidable.
- D is Turing-recognizable.

Church's Thesis: Turing machine is equivalent in computing power to the digital computers.

4.1 Turing Decidable Languages

1. $A_{DFA} = \{< M, w > \mid M \text{ is a DFA that accepts } w. \}$ (Theorem 4.1, TEXT)
2. A_{NFA} (Theorem 4.2, TEXT)
3. $A_{REX} = \{< R, w > \mid R \text{ is a regular expression that generates } w. \}$ (Theorem 4.3, TEXT)

4. $E_{DFA} = \{< A > \mid A \text{ is a DFA such that } L(A) = \emptyset. \}$ (Theorem 4.4, TEXT)

5. $EQ_{DFA} = \{< A, B > \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B). \}$ (Theorem 4.5, TEXT)

6. $A_{CFG} = \{< G, w > \mid G \text{ is a CFG that generates } w. \}$ (Theorem 4.7, TEXT)

7. $E_{CFG} = \{< G > \mid G \text{ is a CFG and } L(G) = \emptyset. \}$ (Theorem 4.8, TEXT)

8. $EQ_{CFG} = \{< G, H > \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H). \}$ (**Not decidable**)

4.2 Diagonalization Method

Goal: Some languages are not Turing-decidable.

Definition: A set A is countable if and only if either A is finite or A has the same size of N. That is, there exists a bijection f such that $f: N \to A$.

example: $N = \{1, 2, 3, \cdots, \}$ and $E = \{2, 4, 6, \cdots, \}$.

1. The set of rational numbers are countable. (Example 4.15, TEXT)
2. The set of real numbers are uncountable. (Theorem 4.17, TEXT)
3. The set of all strings over Σ is countable. (Proof: Corollary 4.18, TEXT)
4. The set of all TMs is countable. (Proof: Corollary 4.18, TEXT)
5. The set of all binary sequences of infinite length is uncountable. (Proof: Corollary 4.18, TEXT)
6. The set of all languages over Σ is uncountable. (Proof: Corollary 4.18, TEXT)

From 4 and 6 above, we have:

Theorem 4.1 *There exists a language that is not Turing-recognizable. (Corollary 4.18, TEXT)*

5 Turing Undecidable Problems and Reducibility

5.1 A_{TM}

Let $A_{TM} = \{< M, w > \mid M \text{ is a TM and } M \text{ accepts } w.\}$

Theorem 5.1 *A_{TM} is Turing undecidable.*

Proof: Suppose A_{TM} is decidable, and let H be a decider (i.e, H is a TM that decides A_{TM}.) Thus,

$$H(< M, w >) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

Now, we construct a new TM D with H as a subroutine:

Given a TM M, D take $< M >$ as an input, and (1) run H on input $< M, < M >>$, (2) output the opposite of what H outputs, i.e., if H accepts, then "reject" and if H rejects, then "accept."

In summary,

$$D(< M >) = \begin{cases} \text{accept} & \text{if } M \text{ does not accepts } < M > \\ \text{reject} & \text{if } M \text{ accepts } < M > \end{cases}$$

What happens when we run D with its own description $< D >$ as input? In that case, we get

$$D(< D >) = \begin{cases} \text{accept} & \text{if } D \text{ does not accepts } < D > \\ \text{reject} & \text{if } D \text{ accepts } < D > \end{cases}$$

That is, no matter what D does, it is forced to do the opposite, a contradiction. Thus, neither TM D nor TM H can exist. Therefore, A_{TM} is not Turing-decidable. ■

However, A_{TM} is Turing-recognizable.

Theorem 5.2 *A language is Turing-decidable if and only if it is Turing-recognizable and also co-Turing-recognizable.*

Corollary 5.1 $\overline{A_{TM}}$ *is not Turing-recognizable.*

5.2 Halting Problem

Let $HALT_{TM} = \{< M, w > \mid M \text{ is a TM and } M \text{ halts on } w, \}$

Theorem 5.3 $HALT_{TM}$ *is Turing undecidable.*

Proof: Suppose $HALT_{TM}$ is Turing-decidable, and let R be a decider. We then use R as a subroutine to construct a TM S that decides A_{TM} as follows. $S =$ "On input $< M, w >$":

1. Run R on $< M, w >$
2. If R reject, reject
3. If R accepts, accept, simulate M until it halts.
4. If M has accepted, accept; if M has rejected, reject.

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM}.Since A_{TM} is undecidable, $HALT_{TM}$ must be undecidable.

Theorem 5.4 *(Theorem 5.2, TEXT) E_{TM} is Turing undecidable.*

Proof: Suppose E_{TM} is decidable. Let R be a decider. We then construct two TMs M_1 and S that takes $< M, w >$, an input to A_{TM} and run as follows.

M_1 = "On input x":

1. If $x \neq w$, reject.
2. If $x = w$, run M on w and accept if M does.

Note that M_1 has w as a part of its description.

S = "On input $< M, w >$":

1. Use the description of M and w to construct M_1
2. Run R on input $< M_1 >$
3. If R accepts, reject; if R rejects, accept.

Clearly, if E_{TM} is TM decidable, then A_{TM} is also TM decidable. However, we already proved A_{TM} is not TM decidable. Hence, E_{TM} is TM undecidable.

5.3 More Turing undecidable Problems

- Post Correspondence Problem (PCP)
- Deciding whether an arbitrary CFG G is ambiguous
- Deciding whether $L(G_1) \cap L(G_2) = \emptyset$ for arbitrary two CFG G_1 and G_2.
-

6 NP-Completeness

6.1 Problem Transformation (Reduction)

Let A and B be two *decision* problems. We say problem A is **transformed** to B using a transformation algorithm f that takes I_A (an arbitrary input to A) and computes $f(I_A)$ (an input to B) such that problem A with input I_A is YES if and only if problem B with input $f(I_A)$ is YES.

EXAMPLES:

- Hamiltonian Path Problem to Hamiltonian Cycle Problem
- Hamiltonian Cycle Problem to Hamiltonian Path Problem

- 3COLORABILITY to 4COLORABILITY
- SAT to 3SAT
- $\cdots$

6.1.1 Upper Bound Analysis

Suppose A is a new problem for which we are interested in computing an upper bound, i.e., finding an algorithm to solve A. Assume we have an algorithm $ALGO_B$ to solve B in $O(n_B)$ time where n_B is the size of an input to B. We can then solve A using the following steps: (i) for an arbitrary instance I_A to A, transform I_A to $f(I_A)$ where $f(I_A)$ is an instance to B; (ii) solve $f(I_A)$ to B using $ALGO_B$; (iii) if $ALGO_B$ taking $f(I_A)$ as an input reports YES, we report I_A is YES; otherwise, NO.

6.1.2 Lower Bound Analysis

6.2 Satisfiability Problem

Let $U = \{u_1, u_2, \cdots, u_n\}$ be a set of boolean *variables*. A *truth* assignment for U is a function $f : U \to \{T, F\}$. If $f(u_i) = T$, we say u_i is *true* under f; and if $f(u_i) = F$, we say u_i is false under f. For each $u_i \in U$, u_i and $\overline{u}_i$ are *literals* over U. The literal $\overline{u}_i$ is true under f if and only if the variable u_i is false under f. A *clause* over U is a set of literals over U such as $\{u_1, \overline{u}_3, u_8, u_9\}$. Each clause represents the disjunction of its literals, and we say it is *satisfied* by a truth assignment function if and only if at least one of its members is true under that assignment. A collection C over U is *satisfiable* if and only if there exists a truth assignment for U that simultaneously satisfies all the clauses in C.

Satisfiability (SAT) Problem

Given: a set U of variable and a collection C of clauses over U

Question: is there a satisfying truth assignment for C?

Example:
$U = \{x_1, x_2, x_3, x_4\}$
$C = \{\{x_1, x_2, x_3\}, \{\overline{x}_1, \overline{x}_3, \overline{x}_4\}, \{\overline{x}_2, \overline{x}_3, x_4\}, \{\overline{x}_1, x_2, x_4\}\}$.

The input to SAT is also given as a *well-formed formula in conjunctive normal form (i.e., sum-of-product form*:

$$w = (x_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_3 + \overline{x}_4)(\overline{x}_2 + \overline{x}_3 + x_4)(\overline{x}_1 + x_2 + x_4)$$

Let $x_1 = T$, $x_2 = F$, $x_3 = F$, $x_4 = T$. Then, $w = T$.

Ans: yes

Reduction from SAT to 3SAT:

(1) $(x_1) \;\rightarrow\; (x_1 + a + b)(x_1 + a + \overline{b})(x_1 + \overline{a} + b)(x_1 + \overline{a} + \overline{b})$

(2) $(x_1 + x_2) \;\rightarrow\; (x_1 + x_2 + a)(x_1 + x_2 + \overline{a})$

(3) $(x_1 + x_2 + x_3 + x_4 + x_5) \;\rightarrow\; (x_1 + x_2 + a_1)(\overline{a_1} + x_3 + a_2)(\overline{a_2} + x_4 + x_5)$

- **3SAT**
- **Not-All-Equal 3SAT**: Each clause has at least one true literal and one false literal, i,e, not all three literals can be true.
- **One-In-Three 3SAT**: Each clause has exactly one true literal and two false literals.

Definition:

P: a set of problems that can be solved deterministically in polynomial time.

NP: a set of problems that can be solved nondeterministically in polynomial time.

NPC: a problem B is called NP-complete or a NP-complete problem if (i) $B \in NP$, i.e., B can be solved nondeterministically in polynomial time, and (ii) for all $B' \in NP$, $B' \leq_P B$, i.e., any problem in NP can be transformed to B deterministically in polynomial time.

Cook's Theorem: Every problem in NP can be transformed to the Satisfiability problem deterministicall in polynomial time.

Note:

(i) The SAT is the first problem belonging to NPC.

(ii) To prove a new problem, say B, being NPC, we need to show (1) B is in NP and (2) any known NPC problem, say B', can be transformed to B deterministically in polynomial time. (By definition of $B' \in NPC$, every problem in NP can be transformed to B in polynomial time. As polynomial time transformation is transitive, it implies that every problem in NP can be transformed to B in polynomial time.)

Theorem: $P = NP$ if and only if there exists a problem $B \in NPC \cap P$.

Proof: If $P = NP$, it is clear that every problem in NPC belongs to P. Now assume that there is a problem $B \in NPC$ that can be solved in polynomial time deterministically. Then by definition of $B \in NPC$, any problem in NP can be transformed to B in polynomial time derterministically, which can then be solved in polynomial time deterministically using the algorithm for B. Hence, $NP \subseteq P$. Since $P \subseteq NP$, we conclude that $P = NP$, which completes the proof of the theorem.

Problem Transformations:

Node Cover Problem:

Given: a graph G and an integer k,

Objective: to find a subset $S \subseteq V$ such that (i) for each $(u, v) \in E$, either u or v (or both) is in S, and (ii) $|S| \leq k$.

Hamiltonian Cycle Problem:

Given: a graph G

Objective: to find a simple cycle of G that goes through every vertex exactly once.

Hamiltonian Path Problem:

Given: a graph G

Objective: to find a simple path of G that goes through every vertex exactly once.

Vertex Coloring Problem:

Given: a graph G and an integer k

Objective: to decide if there exists a *proper* coloring of V (i.e., a coloring of vertices in V such that no two adjacent vertices receive the same color) using k colors.

- $3SAT \leq_P Node - Cover$

Let W be an arbitrary well-formed formula in conjunctive normal form, i.e., in sum-of-product form, where W has n variables and m clauses. We then construct a graph G from W as follows.

The vertex set $V(G)$ is defined as $V(G) = X \cup Y$, where $X = \{x_i, \overline{x}_i \mid 1 \leq i \leq n\}$ and $Y = \{p_j, q_j, r_j \mid 1 \leq j \leq m\}$. The edge set of G is defined to be $E(G) = E_1 \cup E_2 \cup, E_3$, where $E_1 = \{(x_i, \overline{x}_i) \mid 1 \leq i \leq n\}$, $E_2 = \{(p_j, q_j), (q_j, r_j), (r_j, p_j) \mid 1 \leq j \leq m\}$, and E_3 is defined to be a set of edges such that p_j, q_j, and r_j are respectively connected to c_j^1, c_j^2, and c_j^3, where c_j^1, c_j^2, and c_j^3 denote the first, second and the third literals in clause C_j.

For example, let $W = (x_1 + x_2 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$. Then G is defined such that $V(G) = \{x_1, \overline{x}_1, x_2, \overline{x}_2, x_3, \overline{x}_3, p_1, q_1, r_1, p_2, q_2, r_2, p_3, q_3, r_3\}$ and $E(G) = \{(x_1, \overline{x}_1)$, $(x_2, \overline{x}_2)$, $(x_3, \overline{x}_3)$, (p_1, q_1), (q_1, r_1), (r_1, q_1), (p_2, q_2), (q_2, r_2), (r_2, p_2), (p_3, q_3), (q_3, r_3), (r_3, p_3), (p_1, x_1), (q_1, x_2), (r_1, x_3), $(p_2, \overline{x}_1)$, (q_2, x_2), $(r_3, \overline{x}_3)$, $(p_3, \overline{x}_1)$, $(q_3, \overline{x}_2)$, $(r_3, \overline{x}_3)\}$.

We now claim that there exists a truth assignment to make $W = T$ if and only if G has a node cover of size $k = n + 2m$.

To prove this claim, suppose there exists a truth assignment. We then construct a node cover S such that $x_i \in S$ if $x_i = T$ and $\overline{x}_i \in S$ if $x_i = F$. Since at least one literal in each clause C_j must be true, we include the other two nodes in each triangle (i.e., p_j, q_j, r_j) in S. Conversely, assume that there exists a node cover of size $n + 2m$. We then note that exactly one of $x_i, \overline{x}_i$ for each $1 \leq i \leq n$ must be in S, and exactly two nodes in p_j, q_j, r_j for each $1 \leq j \leq m$ must be in S. It is then easy to see the S must be such that at least one node in each p_j, q_j, r_j for $1 \leq j \leq m$ must be connected to a node x_i or $\overline{x}_i$ for $1 \leq i \leq n$. Hence we can find a truth assignment to W by assigning x_i true if $x_i \in S$ and false $\overline{x}_i \in S$.