Notes on Models of Computation

H. Conrad Cunningham

014April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

0 Introduction to the Theory of Computation

Why study theory?

1. To understand the concepts and principles underlying the fundamental nature of computing
   - by constructing abstract models of computers and computation
2. To learn to apply the theory to practical areas of computing
   - in programming languages, compilers, operating systems, networks, etc.
3. To have fun!
   - from tackling challenging "puzzles" and problems

In this course, we study the following models:

1. *automaton* (automata)
   - an abstraction of the computing mechanism
   - takes input, uses temporary storage, makes decisions, and produces output
2. *formal language*
   - an abstraction of a programming language
   - syntax = symbols + grammar rules
3. *algorithm*
   - an abstraction of a mechanical computation
   - what are the limits of what we can and cannot compute?

0.1 Mathematical Preliminaries and Notation

The mathematical concepts used in the Linz textbook include:

- sets
- functions
- relations
- graphs
- trees
- proof techniques

0.1.1 Sets

Students in this course should be familiar with the following set concepts from previous study in mathematics:

- literal set notation such as $\{a, b, c\}$, $\{2, 4, 6, \cdots\}$, and $\{i : i > 10, i < 100\}$
- element (member) $e \in S$
- not an element $e \notin S$
- union $S \cup T$ (in one or both)
- intersection $S \cap T$ (in both)
- difference $S - T$ (in S but not T)
- universal set U (all possible elements)

- complementation $\bar{S}$ (in U but not S)
- empty set $\emptyset$ (no elements)
- subset $S \subseteq T$ (all from S in T)
- proper subset $S \subset T$ (subset but not equal)
- disjoint sets $S \cap T = \emptyset$ (no common elements)
- finite and infinite sets
- cardinality $|S|$ (number elements of finite set)
- powerset 2^S (set of all subsets of a set)
- Cartesian product $S \times T$ (set of all ordered pairs)
- partition of a set (breaking a set into mutually disjoint, nonempty subsets whose intersection is the entire set)

Laws for operations on the empty set:

- $S \cup \emptyset = S$ (identity element for union)
- $S \cap \emptyset = \emptyset$ (zero element for intersection)
- $\bar{\emptyset} = U$
- $\bar{\bar{S}} = S$ (complementation is inverse for itself)

DeMorgan's Laws:

- $S_1 \bar{\cup} S_2 = \bar{S_1} \cap \bar{S_2}$
- $S_1 \bar{\cap} S_2 = \bar{S_1} \cup \bar{S_2}$

0.1.2 Functions

Function $f : D \to R$ means that

- $f \subseteq D \times R$, where
  - *domain* $\subseteq D$
  - *range* $\subseteq R$
  - f maps an element of its domain to a *unique* element of its range

Function f is a

- *total* function if domain = D
- *partial* function otherwise

0.1.3 Relations

A *relation* on X and Y is any subset of $X \times Y$.

An *equivalence relation* $\equiv$ is a generalization of equality. It is:

1. *reflexive*: $x \equiv x \forall x$
2. *symmetric*: if $x \equiv y$ then $y \equiv x$
3. *transitive*: if $x \equiv y$ and $y \equiv z$ then $x \equiv z$

0.1.4 Graphs

A *graph* $\langle V, E \rangle$ is a mathematical entity where

- $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of *vertices* (or *nodes*)
- $E = \{e_1, e_2, \ldots, e_m\}$ is a finite set of *edges* (or *arcs*)
- each edge $e_i = (v_j, v_k)$ is a pair of vertices

A *directed graph* (or *digraph*) is a graph in which each edge $e_i = (v_j, v_k)$ has a direction from vertex v_j to vertex v_k.

Edge $e_i = (v_j, v_k)$ on a digraph is an *outgoing edge* from vertex v_j and an *incoming edge* to vertex v_k.

If there are no directions associated with the edges, then the graph is *undirected*.

Graphs may be *labeled* by assigning names or other information to vertices or edges.

We can visualize a graph with a diagram in which the vertices are shown as circles and edges as lines connecting a pair of vertices. For directed graphs, the direction of an edge is shown by an arrow.

Linz Figure 1.1 shows a digraph $\langle V, E\rangle$ where $V = \{v_1, v_2, v_3\}$ and edges $E = \{(v_1 v_3), (v_3, v_1), (v_3, v_2), (v_3, v_3)\}$.

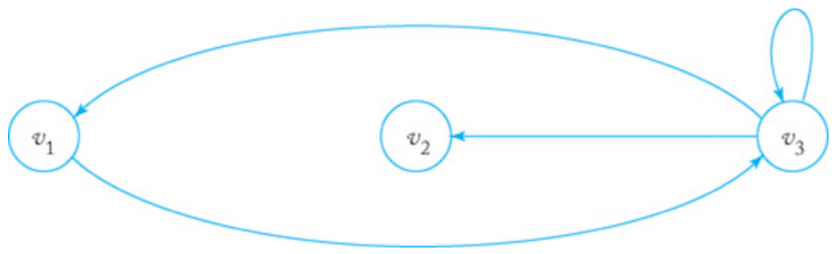


Figure 1: **Linz Fig. 1.1: Diagram of a Digraph**

A sequence of edges $(v_i, vj), (v_j, vk), \ldots, (v_m, v_n)$ is a *walk* from v_j to v_n. The *length* of the walk is the total number of edges traversed.

A *path* is a walk with no edge repeated. A path is *simple* if no vertex is repeated.

A walk from some vertex v_i to itself is a *cycle* with *base* v_i. If no vertex other than the base is repeated, then the cycle is *simple*.

In Linz Figure 1.1:

- $(v_1, v_3), (v_3, v_2)$ is a simple path from v_1 to v_2
- $(v_1, v_3), (v_3, v_3), (v_3, v_1)$ is a cycle but not simple

If the edges of a graph are labelled, then the *label* of a walk (or path) is the sequence of edges encountered on a traversal.

0.1.5 Trees

A *tree* is a directed graph with no cycles and a distinct *root* vertex such that there is exactly one path from the root to every other vertex.

The root of a tree has no incoming edges.

A vertex of a tree without any outgoing edges is a *leaf* of the tree.

If there is an edge from v_i to v_j in a tree, then:

- v_i is the *parent* of v_j
- v_j is a *child* of v_i

The *level* associated with each vertex is the number of edges in the path from the root to the vertex.

The *height* of a tree is the largest level number of any vertex.

If we associated an ordering with the vertices at each level, then the tree is an *ordered tree*.

The above terminology is illustrated in Linz Figure 1.2.

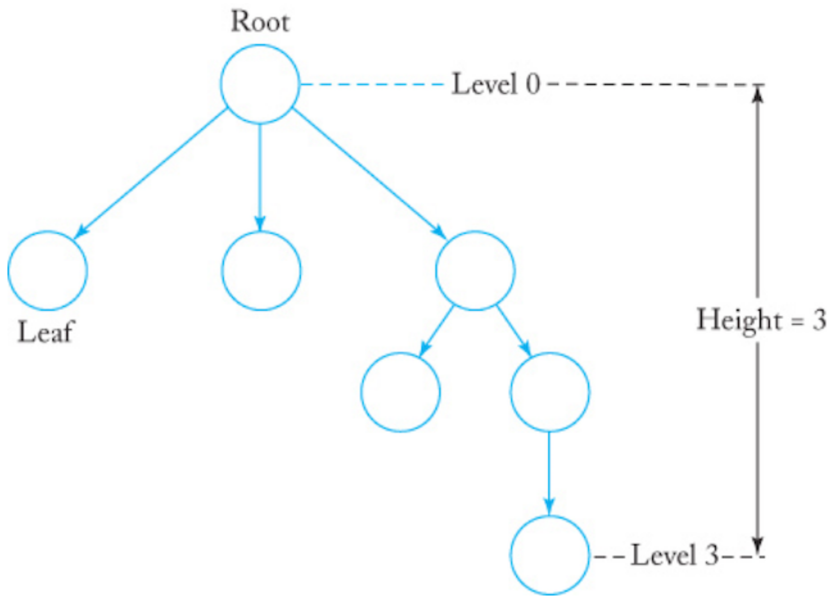


Figure 2: **Linz Fig. 1.2: A Tree**

0.1.6 Proof Techniques

Students in this course should be familiar with the following proof techniques from previous study in mathematics:

- *Deduction*
  - Prove P from axioms and previously proved theorems by a sequence of steps guaranteed by the rules of logic.
- *Contradiction*
  - Assume P is false, prove a sequence of deductive steps that this leads to something we know is false. Hence, this is a contradiction. Thus P must be true.
- *Induction*
  - Basis step: Prove P_0 (i.e, for all primitive cases)
  - Inductive step: Assume $P_n, n \geq 0$, prove P_{n+1}.

We will see an example of an inductive proof in the next section.

0.2 Three Basic Concepts

Three fundamental ideas are the major themes of this course:

1. languages
2. grammars
3. automata

0.2.1 Languages

Our concept of *language* is an abstraction of the concept of a natural language.

0.2.1.1 Language Concepts

Linz Definition (Alphabet): An *alphabet*, denoted by Σ, is a finite, nonempty set of symbols.

By convention, we use lowercase letters near the beginning of the English alphabet $a, b, c, \cdots$ to represent elements of Σ.

For example, if $\Sigma = \{a, b\}$, then the alphabet has two unique symbols denoted by a and b.

Linz Definition (String): A *string* is a finite sequence of symbols from the alphabet.

By convention, we use lowercase letters near the end of the English alphabet $\cdots u, v, w, x, y, z$ to represent strings. We write strings left to right. That is, symbols appearing to the left are before those appearing to the right.

For example, $w = baabaa$ is a string from the above alphabet. The string begins with a b and ends with an a.

Linz Definition (Concatenation): The *concatenation* of strings u and v means appending the symbols of v to the right end (i.e., after) the symbols of u, denoted by uv.

If $u = a_1a_2a_3$ and $v = b_1b_2b_3$, then $uv = a_1a_2a_3b_1b_2b_3$.

Definition (Associativity): Operation $\oplus$ is *associative* on set S if, for all x, y, and z in S, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$. We often write associative expressions without explicit parentheses, e.g., $x \oplus y \oplus z$.

String concatenation is *associative*, that is, $(uv)w = u(vw)$.

Thus we normally just write uvw without parentheses.

Definition (Commutativity): Operation $\oplus$ is *commutative* on set S if, for all x and y in S, $x \oplus y = y \oplus x$.

String concatenation is *not commutative.* That is, $uv \neq vu$.

Linz Definition (Reverse): The *reverse* of a string w, denoted by w^R, is the string with same symbols, but with the order reversed.

If $w = a_1a_2a_3$, then $w^R = a_3a_2a_1$.

Linz Definition (Length): The *length* of a string w, denoted by $|w|$, is the number of symbols in string w.

Linz Definition (Empty String): The *empty string*, denoted by λ, is the string with no symbols, that is, $|\lambda| = 0$.

Definition (Identity Element): An operation $\oplus$ has an *identity element* e on set S if, for all $x \in S$, $x \oplus e = x = e \oplus x$.

The empty string λ is the *identity element* for concatenation. That is, $\lambda w = w\lambda = w$.

0.2.1.2 Formal Interlude: Inductive Definitions and Induction

We can define the *length* of a string with the following *inductive definition*:

1. $|\lambda| = 0$ (*base* case)
2. $|wa| = |w| + 1$ (*inductive* case)

Note: This inductive defintion and proof differs from the textbook. Here we begin with the empty string.

Using the fact that λ is the identity element and the above definition, we see that

$$|a| = |\lambda a| = |\lambda| + 1 = 0 + 1 = 1.$$

Prove $|uv| = |u| + |v|$.

Noting the definition of length above, we choose to do an induction over string v (or, if you prefer, over the length of v, basing induction at 0).

Base case $v = \lambda$ (that is, length is 0)

$$\begin{array}{ll}
\hline
 & |u\lambda| \\
= & \{ \text{ identity for concatenation } \} \longleftarrow \text{justification for step in braces} \\
 & |u|
\end{array}$$

$$\begin{array}{ll}
= & \{ \text{ identity for } + \} \\
 & |u| + 0 \\
= & \{ \text{ definition of length } \} \\
 & |u| + |\lambda|
\end{array}$$

Inductive case $v = wa$ (that is, length is greater than 0)
Induction hypothesis: $|uw| = |u| + |w|$

$$\begin{array}{ll}
 & |u(wa)| \\
= & \{ \text{ associativity of concatenation } \} \\
 & |(uw)a| \\
= & \{ \text{ definition of length } \} \\
 & |uw| + 1 \\
= & \{ \text{ induction hypothesis } \} \\
 & (|u| + |w|) + 1 \\
= & \{ \text{ associativity of } + \} \\
 & |u| + (|w| + 1) \\
= & \{ \text{ definition of length (right to left) } \} \\
 & |u| + (|wa|)
\end{array}$$

Thus we have proved $|uv| = |u| + |v|$. QED.

0.2.1.3 More Language Concepts

Linz Definition (Substring): A *substring* of a string w is any string of consecutive symbols in w.

If $w = a_1a_2a_3$, then the substrings are $\lambda, a_1, a_1a_2, a_1a_2a_3, a_2, a_2a_3, a_3$.

Linz Definition (Prefix, Suffix): If $w = vu$, then v is a *prefix* of w and u is a *suffix*.

If $w = a_1a_2a_3$, the prefixes are $\lambda, a_1, a_1a_2, a_1a_2a_3$.

Linz Definition (w^n): w^n, for any string w and $n \geq 0$ denotes n repetitions of string (or symbol) w. We further define $w^0 = \lambda$.

Linz Definition (Star-Closure): Σ^*, for alphabet Σ, is the set of all strings obtained by concatenating zero or more symbols from the alphabet.

Note: An alphabet must be a finite set.

Linz Definition (Positive Closure): $\Sigma^+ = \Sigma^* - \lambda$

Although Σ is finite, Σ^* and Σ^+ are infinite.

For a string w, we also write w^* and w^+ to denote zero or more repetitions of the string and one or more repetitions of the string, respectively.

Linz Definition (Language): A *language*, for some alphabet Σ, is a subset of Σ^*.

Linz Definition (Sentence): A *sentence* of some language L is any string from L (i.e., from Σ^*).

0.2.1.4 Linz Example 1.9: Example Languages

Let $\Sigma = \{a, b\}$.

- $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \cdots\}$.
- $\{a, aa, aab\}$ is a language on Σ.

Since the language has a finite number of sentences, it is a *finite language*.

- $L = \{a^n b^n : n \geq 0\}$ is also a language on Σ.

Sentences $aabb$ and $aaaabbbb$ are in L, but $aaabb$ is not.

As with most interesting languages, L is an *infinite language*.

0.2.1.5 Operations on Languages

Languages are represented as sets. Operations on languages can be defined in terms of set operations.

Linz Definition (Union, Intersection, and Difference): Language *union*, *intersection*, and *difference* are defined directly as the corresponding operations on sets.

Linz Definition (Concatenation): Language *complementation* with respect to Σ^* is defined such that $\bar{L} = \Sigma^* - L$.

Linz Definition (Reverse): Language *reverse* is defined such that $L^R = \{w^R : w \in L\}$. (That is, reverse all strings.)

Linz Definition (Concatenation): Language *concatenation* is defined such that $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$.

Linz Definition (L^n): L^n means L concatenated with itself n times.

$L^0 = \{\lambda\}$ and $L^{n+1} = L^n L$

Definition (Star-Closure): *Star-closure* (Kleene star) is defined such that $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$.

Definition (Positive Closure): *Positive closure* is defined such that $L^+ = L^1 \cup L^2 \cup \cdots$.

0.2.1.6 Language Operation Examples

Let $L = \{a^n b^n : n \geq 0\}$.

- $L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}$ (where n and m are unrelated).
- $abaaabbb \in L^2$.
- $L^R = \{b^n a^n : n \geq 0\}$

How would we express in $\bar{L}$ and L^*?

Although set notation is useful, it is not a convenient notation for expressing complicated languages.

0.2.2 Grammars

0.2.2.1 Grammar Concepts

Linz Definition 1.1 (Grammar): A *grammar* G is a quadruple $G = (V, T, S, P)$ where

> V is a finite set of objects called *variables*.
> T is a finite set of objects called *terminal symbols*.
> $S \in V$ is a special symbol called the *start symbol*.
> P is a finite set of *productions*.
> V and T are nonempty and disjoint.

Linz Definition (Productions): *Productions* have form $x \rightarrow y$ where:

> $x \in (V \cup T)^+$, i.e., x is some non-null string of terminals and variables
> $y \in (V \cup T)^*$, i.e., y is some, possibly null, string of terminals and variables

Consider application of productions, given $w = uxv$:

- $x \rightarrow y$ is *applicable* to string w.
- To use the production, substitute y for x.

  Thus the new string is $z = uyv$.

  We say w *derives* z, written $w \Rightarrow z$.
- Continue by applying any applicable productions in arbitrary order.

  $w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n$.

Linz Definition (Derives): $w_1 \overset{*}{\Rightarrow} w_n$ means that w_1 derives w_n in zero or more production steps.

Linz Definition (Language Generated): Let $G = (V, T, S, P)$ be a grammar. Then $L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$ is the *language generated* by G.

That is, $L(G)$ is the set of all strings that can be generated from the start symbol S using the productions P.

Linz Definition (Derivation): A *derivation* of some sentence $w \in L(G)$ is a sequence $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$.

The strings $S, w_1, \cdots, w_n$ above are *sentential forms* of the derivation of sentence w.

0.2.2.2 Linz Example 1.11 (Grammar)

Consider $G = (\{S\}, \{a, b\}, S, P)$ where P is the set of productions

- $S \to aSb$
- $S \to \lambda$

Consider $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$. Hence, $S \overset{*}{\Rightarrow} aabb$.

aabb is a sentence of the language; the other strings in the derivation are sentential forms.

Conjecture: The language formed by this grammar, $L(G)$, is $\{a^n b^n : n \geq 0\}$.

Usually, however, it is difficult to construct an explicit set definition for a language generated by a grammar.

Now prove the conjecture.

First, prove that all sentential forms of the language have the structure $w_i = a^i S b^i$ for $i \geq 0$ by induction on i.

Basis step: Clearly, $w_0 = S$ is a sentential form, the start symbol.

Inductive step: Assume $w_m = a^m S b^m$ is a sentential form, show that $w_{m+1} = a^{m+1} S b^{m+1}$.

Case 1: If we begin with the assumption and apply production $S \to aSb$, we get sentential form $w_{m+1} = a^{m+1} S b^{m+1}$.

Case 2: If we begin with the assumption and apply production $S \to \lambda$, we get the sentence $a^m b^m$ rather than a sentential form.

Hence, all sentential forms have the form $a^i S b^i$.

Given that $S \to \lambda$ is the only production with terminals on the right side, we must apply it to derive any sentence. As we noted in case 2 above, application of the production to any sentential form gives a sentence of the form $a^m b^m$. QED.

0.2.2.3 Linz Example 1.12: Finding a Grammar for a Language

Given $L = \{a^n b^{n+1} : n \geq 0\}$.

- Recursive production $S \to aSb$ would generate sentential forms $a^n S b^n$.
- Need production(s) to add the extra b to the final sentence.
- Suggest $S \to b$.

A slightly different grammar might introduce nonterminal A as follows:

- $S \to Ab$
- $A \to aAb$
- $A \to \lambda$

0.2.2.4 More Grammar Concepts

To show that a language L is generated by a grammar G, we must prove:

1. For every $w \in L$, there is a derivation using G.
2. Every string derived from G is in L.

Linz Definition (Equivalence): Two grammars are *equivalent* if they generate the same language.

For example, the two grammars given above for the language $L = \{a^n b^{n+1} : n \geq 0\}$ are equivalent.

0.2.2.5 Linz Example 1.13

Let $\Sigma = \{a, b\}$ and let $n_a(w)$ and $n_b(w)$ denote the number of a's and b's in the string w.

Let grammar G have productions

$S \rightarrow SS$
$S \rightarrow \lambda$
$S \rightarrow aSb$
$S \rightarrow bSa$

Let $L = \{w : n_a(w) = n_b(w)\}$.

Prove $L(G) = L$.

Informal argument follows. Actual proof would be an induction over length of w.

Consider cases for w.

1. Case $w \in L(G)$. Show $w \in L$.

   Any production adding an a also adds a b. Thus there is the same number of a's and b's.

2. Case $w \in L$. Show $w \in L(G)$.

   - Consider $w = aw_1b$ or $w = bw_1a$ for some $w_1 \in L$.

     String w was generated by either $S \rightarrow aSb$ or $S \rightarrow bSa$ in the first step.

     Thus $w \in L$.

   - Consider $w = aua$ (or $w = bub$) for some $u \in L$.

     Examine the symbols of w from the left – add 1 for each a, subtract 1 for each b.

     Since sum must be 0 at right, there must be a point where the sum crosses 0.

     Break at that point into form $w = w_1w_2$ where $w_1, w_2 \in L$.

     First production is $S \rightarrow SS$.

Thus $w \in L(G)$.

0.2.3 Automata

An *automaton* is an abstract model of a compute

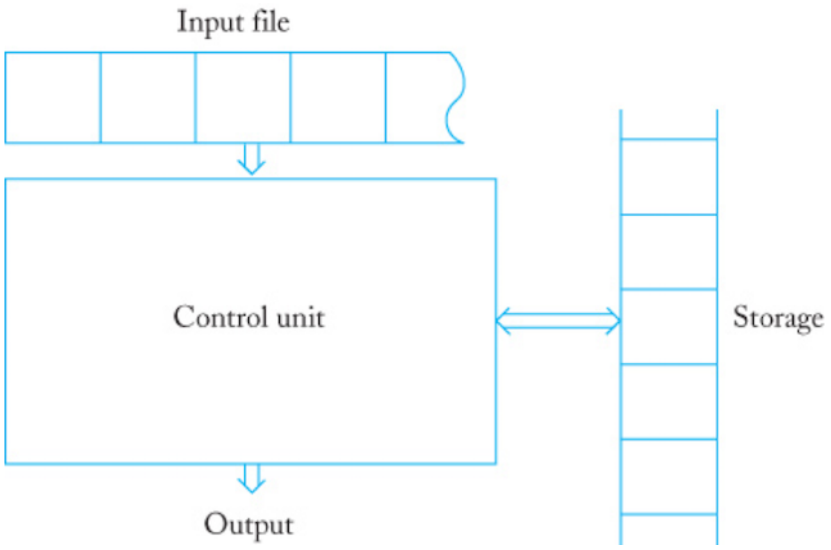


Figure 3: **Linz Fig. 1.4: Schematic Representation of a General Automaton**

As shown in Linz Figure 1.4, a computer:

1. *reads input* from an *input file* – one symbol from each cell – left to right
2. *produces output*
3. *may use storage* – unlimited number of cells (may be different alphabet)
4. *has a control unit*
    - finite number of states
    - state changes in defined manner
    - "next-state" or *transition function* – specifies state changes

A *configuration* is a state of the control unit, input, and storage.

A *move* is a transition from one state configuration to another.

Automata can be categorized based on control:

- A *deterministic automaton* has a unique next state from the current configuration.
- A *nondeterministic automaton* has several possible next states.

Automata can also be categorized based on output:

- An *accepter* has only yes/no output.
- A *transducer* has strings or symbols for output,

Various models differ in

- how the output is produced
- the nature of temporary storage

0.3 Applications

0.3.1 Linz Example 1.15: C Identifiers

The syntax rules for identifiers in the language C are as follows:

- An identifier is a sequence of letters, digits, and underscores.
- An identifier must start with a letter or underscore.
- Identifiers allow both uppercase and lowercase letters.

Formally, we can describe these rules with the grammar:

```
<id>       -> <letter><rest>   | <underscr><rest>
<rest>     -> <letter><rest>   | <digit><rest>    |
              <underscr><rest> | <lambda>
<letter>   -> a|b|c|...|z|A|B|C|...|Z
<digit>    -> 0|1|2|...|9
<underscr> -> _
```

Above `<lambda>` represents the symbol λ,`->` is the $\rightarrow$ for productions, and `|` denotes alternative right-hand-sides of the productions.

The *variables* are `<id>`, `<letter>`, `<digit>`, `<underscr>`, and `<rest>`. The other alphanumeric symbols are *literals*.

Linz Figure 1.6 shows a drawing of an automaton that accepts all legal C identifiers as defined above.

We can interpret the automaton in Linz Figure 1.6 as follows:

- The machine starts in state 1.
- It reads the string left to right, one character at a time.
- If the first character is a `<digit>` then the machine moves to state 3. The machine stops reading with answer No (non-accepting).
- If first character is a `<letter>` or `<underscr>` then it moves to state 2. The machine continues.
- As long as the next character is a `<letter>`, `<underscr>`, or `<digit>`, then the machine reads the input and remains in state 2.
- The machine stops in state 2 when either there is no more input or unacceptable input.

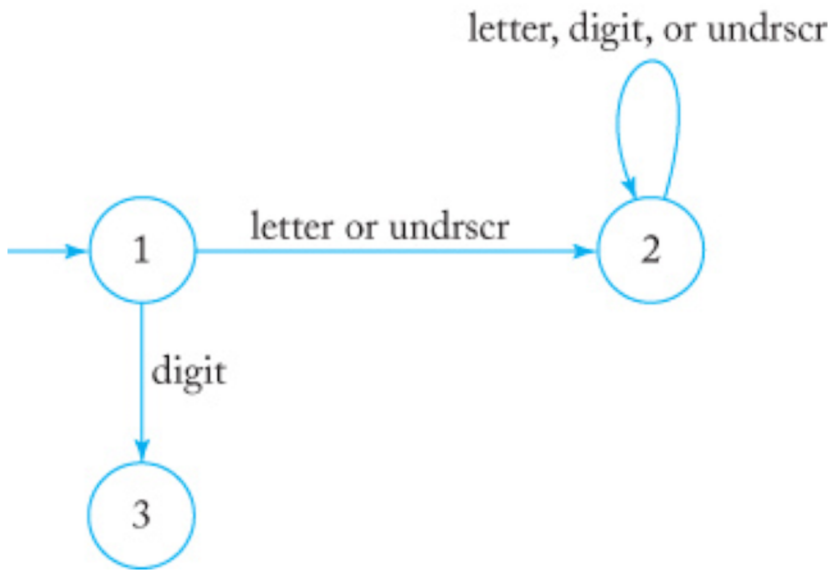


Figure 4: **Linz Fig. 1.7: Automaton to Accept C Identifiers**

- If no more input and in state 2, then machine stops with answer Yes (accepting). Otherwise, it stops with the answer No (non-accepting).

0.3.2 Linz Example 1.17: Binary Adder

Let $x = a_0 a_1 a_0 \cdots a_n$ where a_i are bits.

Then $value(x) = \sum_{i=0}^{n} a_i 2^i$.

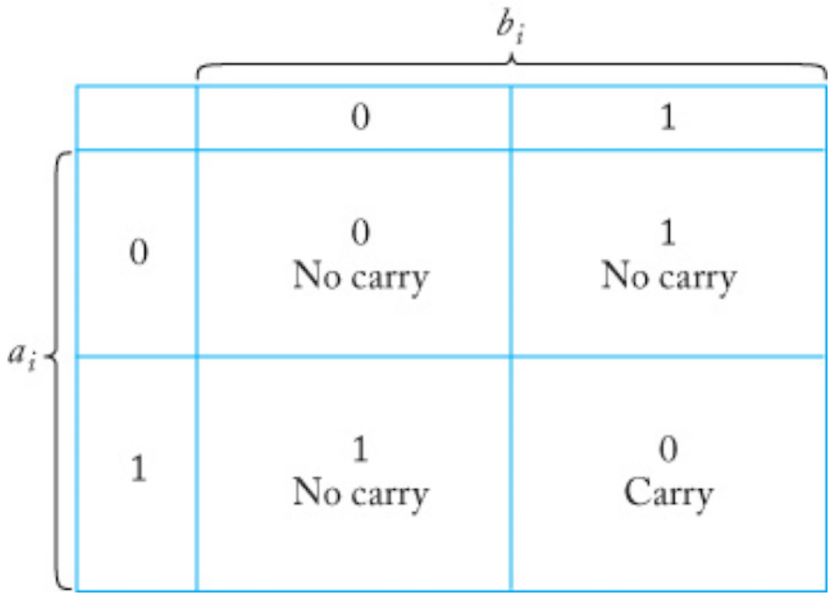
This is the usual binary representation in reverse.

A serial adder process two such numbers x and y, bit by bit, starting at the left end. Each bit addition creates a digit for the sum and a carry bit as shown in Linz Figure 1.7.

A *block diagram* for the machine is shown in Linz Figure 1.8.

A transducer automaton to carry out the addition of two numbers is shown in Linz Figure 1.9.

The pair on the edges represents the two inputs. The value following the slash is the output.



a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

Figure 5: **Linz Fig. 1.7: Binary Addition Table**

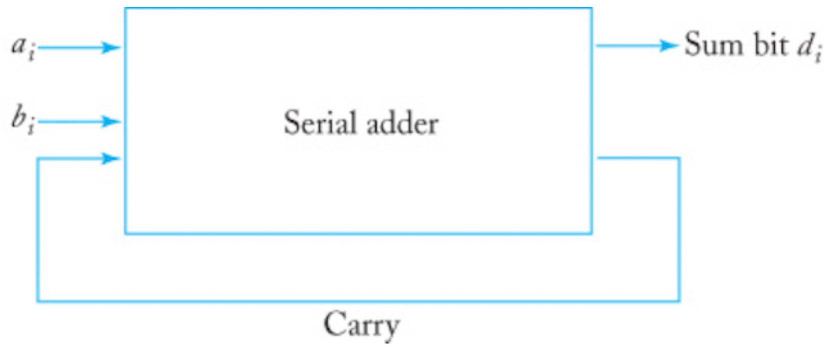


Figure 6: **Linz Fig. 1.8: Binary Adder Block Diagram**

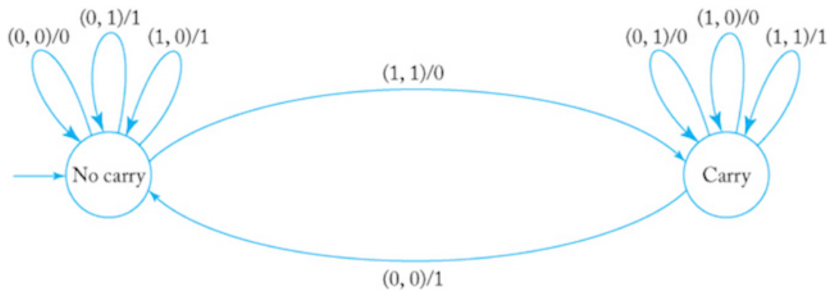


Figure 7: **Linz Fig. 1.9: Binary Adder Transducer Automaton**

1 Finite Automata

In chapter 2, we approach automata and languages more precisely and formally than in chapter 1.

A *finite automaton* is an automaton that has no temporary storage (and, like all automata we consider in this course, a finite number of states and input alphabet).

1.1 Deterministic Finite Accepters

1.1.1 Accepters

Linz Definition (DFA): A *deterministic finite accepter*, or *dfa*, is defined by the tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of *internal states*
- Σ is a finite set of symbols called the *input alphabet*
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*
- $q_0 \in Q$ is the *initial state.*
- $F \subseteq Q$ is a set of *final states.*

A dfa operates as described in the following pseudocode: currentState := q_0

```
position input at left end of string

while more input exists currentInput := next_input_symbol
    advance input to right
    currentState := $\delta$(currentState,currentInput)

if currentState $\in F$ then ACCEPT else REJECT
```

1.1.2 Transition Graphs

To visualize a dfa, we use a *transition graph* constructed as follows:

- Vertices represent states
  - labels are state names
  - exactly one vertex for every $q_i \in Q$
- Directed edges represent transitions
  - label on edge is current input symbol
  - directed edge (q, r) with label a if and only if $\delta(q, a) = r$

1.1.3 Linz Example 2.1

The graph pictured in Linz Figure 2.1 represents the dfa $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$, where δ is represented by

$\delta(q_0, 0) = q_0,\ \delta(q_0, 1) = q_1$
$\delta(q_1, 0) = q_0,\ \delta(q_1, 1) = q_2$
$\delta(q_2, 0) = q_2,\ \delta(q_2, 1) = q_1$

Note that q_0 is the *initial state* and q_1 is the only *final state* in this dfa.

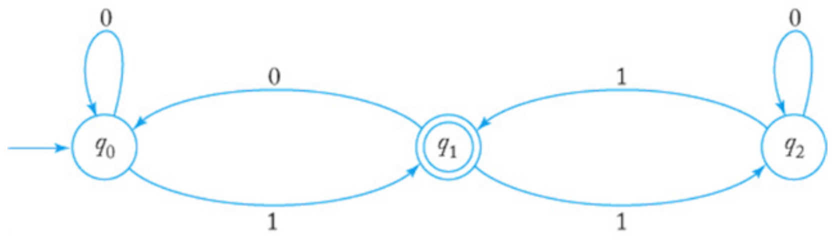


Figure 8: **Linz Fig. 2.1: DFA Transition Graph**

The dfa in Linz Figure 2.1:

- Accepts 01, 101
- Rejects 00, 100

What about 0111? 1100?

1.1.4 Extended Transition Function for a DFA

Linz Definition: The *extended transition function* $\delta^* : Q \times \Sigma^* \to Q$ is defined recursively:

$\delta^*(q, \lambda) = q$
$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$

The extended transition function gives the state of the automaton after reading a string.

1.1.5 Language Accepted by a DFA

Linz Definition 2.2 (Language Accepted by DFA): The language accepted by a dfa $M = (Q, \Sigma, \delta, q_0, F)$ is $L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$.

That is, $L(M)$ is the set of all strings on the input alphabet accepted by automaton M.

Note that above δ and δ^* are total functions (i.e., defined for all strings).

1.1.6 Linz Example 2.2

The automaton in Linz Figure 2.2 accepts all strings consisting of arbitrary numbers of a's followed by a single b.

In set notation, the language accepted by the automaton is $L = \{a^n b : n \geq 0\}$.

Note that q_2 has two self-loop edges, each with a different label. We write this compactly with multiple labels.

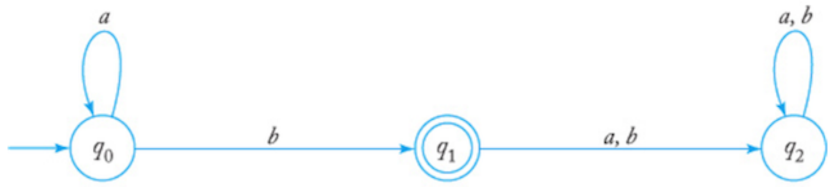


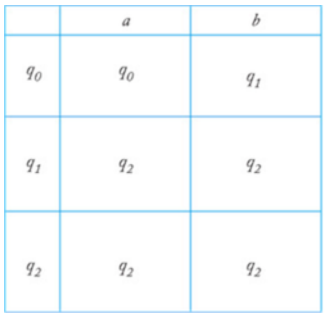
Figure 9: **Linz Fig. 2.2: DFA Transition Graph with Trap State**

A *trap state* is a state from which the automaton can never “escape”.

Note that q_2 is a trap state in the dfa transition graph shown in Linz Figure 2.2.

Transition graphs are quite convenient for understanding finite automata.

For other purposes–such as representing finite automata in programs–a tabular representation of transition function δ may also be convenient (as shown in Linz Fig. 2.3).



	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Figure 10: **Linz Fig. 2.3: DFA Transition Table**

1.1.7 Linz Example 2.3

Find a deterministic finite accepter that recognizes the set of all string on $\Sigma = \{a, b\}$ starting with the prefix ab.

Linz Figure 2.4 shows a transition graph for a dfa for this example.

The dfa must accept ab and then continue until the string ends.

This dfa has a *final* trap state q_2 (accepts) and a *non-final* trap state q_3 (rejects).

1.1.8 Linz Example 2.4

Find a dfa that accepts all strings on $\{0, 1\}$, except those containing the substring 001.

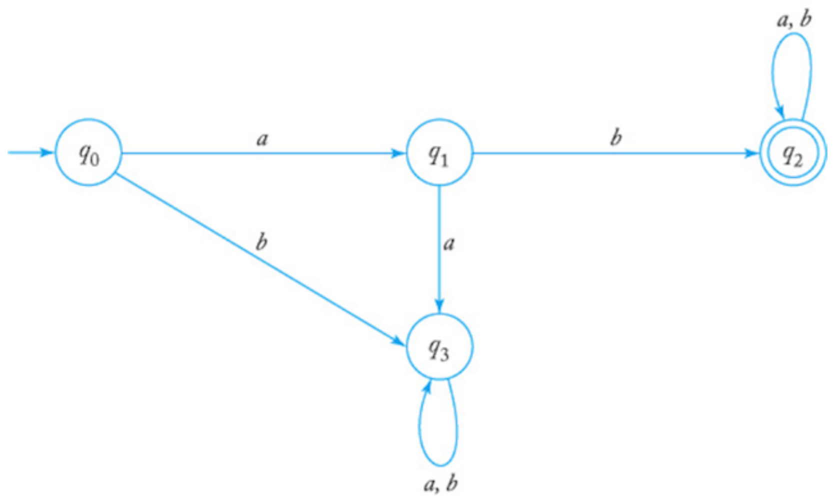


Figure 11: **Linz Fig. 2.4**

- need to “remember” whether last two inputs were 00
- use state changes for “memory”

Linz Figure 2.5 shows a dfa for this example.

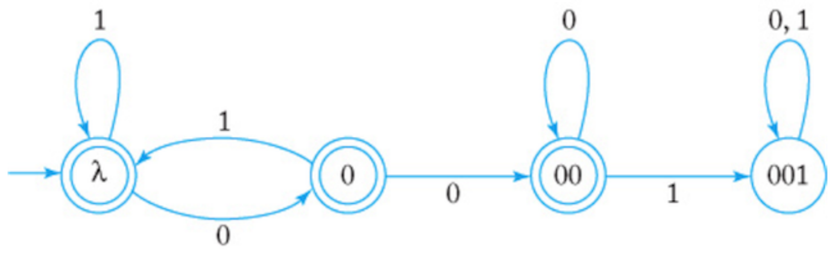


Figure 12: **Linz Fig. 2.5**

Accepts: λ, 0, 00, 01, 010000

Rejects: 001, 000001, 0010101010101

1.1.9 Regular Languages

Linz Definition 2.3 (Regular Language): A language L is called *regular* if and only if there exists a dfa M such that $L = L(M)$.

Thus dfas define the *family* of languages called *regular*.

1.1.10 Linz Example 2.5

Show that the language $L = \{awa : w \in \{a,b\}^*\}$ is regular.

- Construct a dfa.
- Check whether begin/end with "a".
- Am in final state when second a input.

Linz Figure 2.6 shows a dfa for this example.

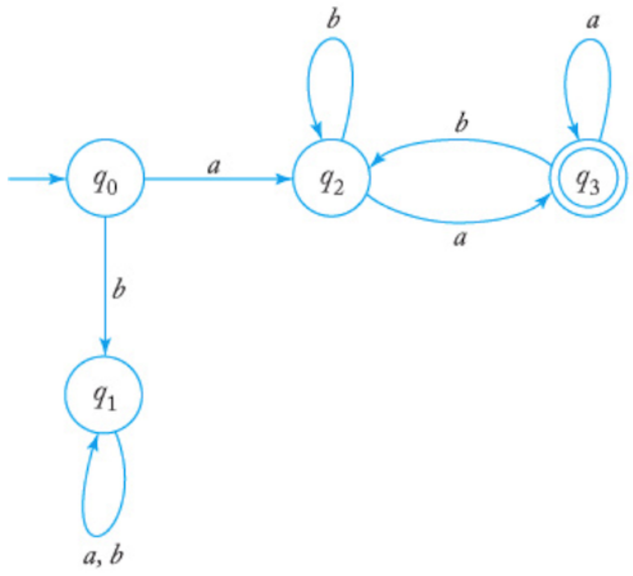


Figure 13: **Linz Fig. 2.6**

Question: How would we prove that a languages is not regular?

We will come back to this question in chapter 4.

1.1.11 Linz Example 2.6

Let L be the language in the previous example (Linz Example 2.5).

Show that L^2 is regular.

$L^2 = \{aw_1aaw_2a : w_1, w_2 \in \{a,b\}^*\}$.

- Construct a dfa.
- Use Example 2.5 dfa as starting point.
- Accept two consecutive strings of form awa.
- Note that any two consecutive a's could start a second string.

Linz Figure 2.7 shows a dfa for this example.

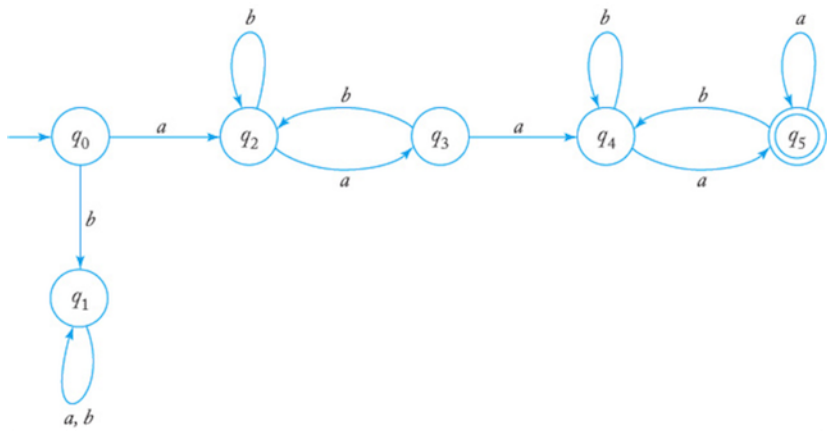


Figure 14: **Linz Fig. 2.7**

The last example suggests the conjecture that if a language L then so is L^2, L^3, etc.

We will come back to this issue in chapter 4.

1.2 Nondeterministic Finite Accepters

1.2.1 Nondeterministic Accepters

Linz Definition 2.4 (NFA): A *nondeterministic finite accepter* or *nfa* is defined by the tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, and F are defined as for deterministic finite accepters, but

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q.$$

Remember for dfas:

- Q is a finite set of *internal states*.
- Σ is a finite set of symbols called the *input alphabet*.
- $q_0 \in Q$ is the *initial state*.
- $F \subseteq Q$ is a set of *final states*.

The key differences between dfas and nfas are

1. dfa: δ yields a single state
   nfa: δ yields a set of states
2. dfa: consumes input on each move
   nfa: can move without input (λ)

3. dfa: moves for all inputs in all states
   nfa: some situations have no defined moves

An nfa *accepts* a string if *some possible* sequence of moves ends in a final state.

An nfa *rejects* a string if *no possible* sequence of moves ends in a final state.

1.2.2 Linz Example 2.7

Consider the transition graph shown in Linz Figure 2.8.

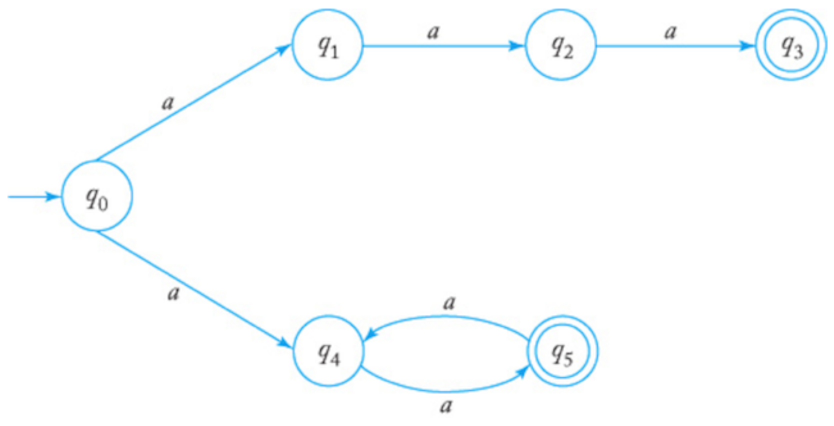


Figure 15: **Linz Fig. 2.8**

Note the nondeterminism in state q_0 with two possible transitions for input a.

Also state q_3 has no transition for any input.

1.2.3 Linz Example 2.8

Consider the transition graph for an nfa shown in Linz Figure 2.9.

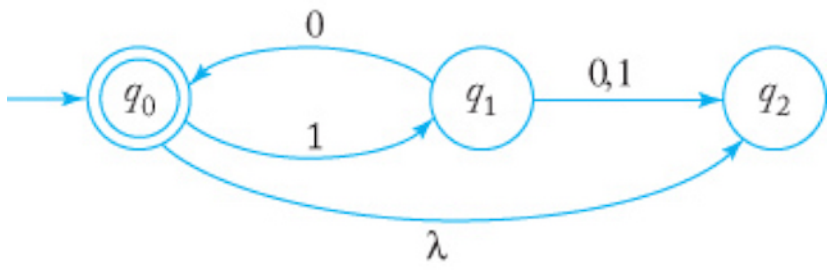


Figure 16: **Linz Fig. 2.9**

Note the nondeterminism and the λ-transition.

Note: Here λ means the move takes place without consuming any input symbol. This is different from accepting an empty string.

Transitions:

- for $(q_0, 0)$?
- for $(q_1, 0)$?
- for $(q_2, 0)$?
- for $(q_2, 1)$?

Accepts: λ, 10, 1010, 101010

Rejects: 0, 11

1.2.4 Extended Transition Function for an NFA

As with dfas, the transition function can be *extended* so its second argument is a string.

Requirement: $\delta^*(q_i, w) = Q_j$ where Q_j is the set of all possible states the automaton may be in, having started in state q_i and read string w.

Linz Definition (Extended Transition Function): For an nfa, the *extended transition function* is defined so that $\delta^*(q_i, w)$ contains q_j if there is a walk in the transition graph from q_i to q_j labelled w.

1.2.5 Language Accepted by an NFA

Linz Definition 2.6 (Language Accepted by NFA): The language L accepted by the nfa $M = (Q, \Sigma, \delta, q_0, F)$ is defined

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F \neq \emptyset\}.$$

That is, $L(M)$ is the set of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

1.2.6 Linz Example 2.10 (Example 2.8 Revisited)

Let's again examine the automaton given in Linz Figure 2.9 (Example 2.8).

This nfa, call it M:

- must end in q_0
- $L(M) = \{(10)^n : n \geq 0\}$

Note that q_2 is a *dead configuration* because $\delta^*(q_0, 110) = \emptyset$.

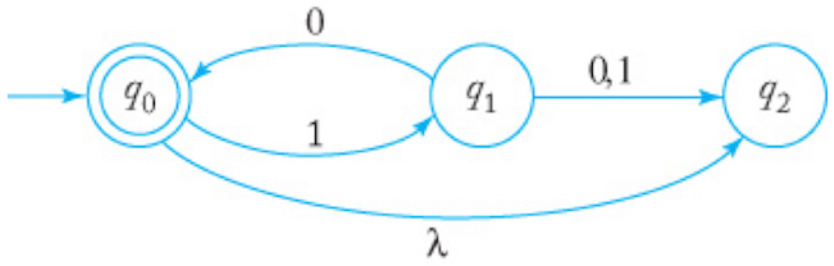


Figure 17: **Linz Fig. 2.9 (Repeated)**

1.2.7 Why Nondeterminism

When computers are deterministic?

- an nfa can model a search or backtracking algorithm
- nfa solutions may be simpler than dfa solutions (can convert from nfa to dfa)
- nondeterminism may model externally influenced interactions (or abstract more detailed computations)

1.3 Equivalence of DFAs and NFAs

1.3.1 Meaning of Equivalence

When are two mechanisms (e.g., programs) equivalent?

- When they have exactly the same descriptions?
- When they always go through the exact same sequence of steps?
- When the same input generates the same output for both?

The last seems to be the best approach.

Linz Definition 2.7 (DFA-NFA Equivalence): Two finite accepters M_1 and M_2 are said to be *equivalent* if $L(M_1) = L(M_2)$. That is, if they both accept the same language.

Here "same language" refers to the *input* and "both accept" refers to the *output*.

Often there are many accepters for a language.

Thus there are many equivalent dfa and nfas.

1.3.2 Linz Example 2.11

Again consider the nfa represented by the graph in Linz Fig. 2.9. Call this M_1.

As we saw, $L(M_1) = \{(10)^n : n \geq 0\}$.

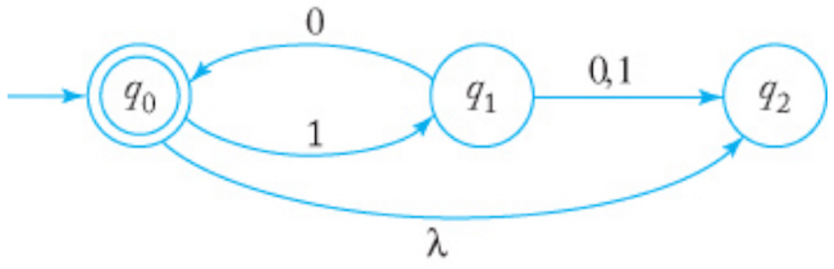


Figure 18: **Linz Fig. 2.9 (Repeated): An NFA**

Now consider the dfa represented by the graph in Linz Figure 2.11. Call this M_2.

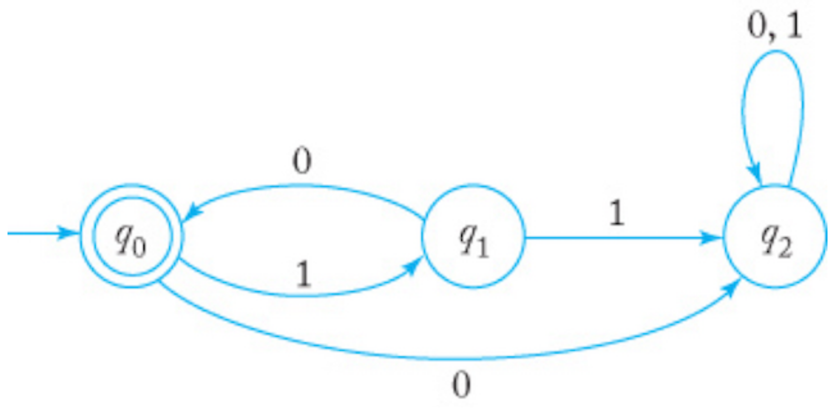


Figure 19: **Linz Fig. 2.11: DFA Equivalent to Fig. 9 NFA**

$L(M_2) = \{(10)^n : n \geq 0\}$.

Thus, M_1 is equivalent to M_2.

1.3.3 Power of NFA versus DFA

Which is more powerful, dfa or nfa?

Clearly, any dfa D can be made into a nfa N.

- Keep the same states.
- Define $\delta_N(q, a) = \{\delta_D(q, a)\}$.

Can any nfa N be made into a dfa D?

- Yes, but it is less obvious. (See theorem below.)

Thus, dfas and nfas are "equally powerful".

Linz Theorem 2.2 (Existence of DFA Equivalent to NFA): Let L be the language accepted by the nfa $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Then there exists a dfa $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L = L(M_D)$.

A pure mathematician would be content with an existence proof.

But, as computing scientists, we want an *algorithm* for construction of M_D from M_N. The proof of the theorem follows from the correctness of the following algorithm.

Key ideas:

- After reading w, M_N will be in the some state from $\{q_i, q_j, \ldots\ q_k\}$. That is, $\delta^*(q_0, w) = \{q_i, q_j, \ldots\ q_k\}$.
- Label the *dfa state* that has accepted w with the set of nfa states $\{q_i, q_j, \ldots\ q_k\}$. This is an interesting "trick"!

Remember from the earlier definitions in these notes and from discrete mathematics:

- Σ is finite (and the same for the nfa and dfa).
- Q_N is finite.
- δ_D is a total function. That is, every vertex of the dfa graph has $|\Sigma|$ outgoing edges.
- The maximum number of dfa states with the above labeling is $|2^{Q_N}| = 2^{|Q_N|}$. Hence, finite.
- The maximum number of dfa edges is $2^{|Q_D|}|\Sigma|$. Hence, finite.

Procedure nfa_to_dfa

Given a transition graph G_N for nfa $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, construct a transition graph G_D for a dfa $M_D = (Q_D, \Sigma, \delta_D, q_0, F_D)$. Label each vertex in G_D with a *subset* of the vertices in G_N.

1. Initialize graph G_D to have an initial vertex $\{q_0\}$ where q_0 is the initial state of G_N.
2. Repeat the following steps until no more edges are missing from G_D:
   a. Take any vertex from G_D labeled $\{q_i, q_j, \ldots\ q_k\}$ that has no outgoing edge for some $a \in \Sigma$.
   b. For this vertex and input, compute $\delta_N^*(q_i, a), \delta_N^*(q_j, a), \ldots\ \delta_N^*(q_k, a)$. (Each of these is a set of states from Q_N.)
   c. Let $\{q_l, q_m, \ldots\ q_n\}$ be the union of all δ_N^* sets formed in the previous step.
   d. If vertex $\{q_l, q_m, \ldots\ q_n\}$ constructed in the previous step (step 2c) is not already in G_D, then add it to G_D.

e. Add an edge to G_D from vertex $\{q_i, q_j, \ldots\ q_k\}$ (vertex selected in step 2b) to vertex $\{q_l, q_m, \ldots\ q_n\}$ (vertex possibly created in step 2d) and label the new edge with a (input selected in step 2b).

3. Make every vertex of G_D whose label contains any vertex $q_f \in F_N$ a final vertex of G_D.
4. If M_N accepts λ, then vertex $\{q_0\}$ in G_D is also a final vertex.

This, if the loop terminates, it constructs the dfa corresponding to the nfa.

Does the loop terminate?

- Each iteration of the loop adds one edge to G_D.
- There are a finite number of edges possible in G_D.
- Thus the loop must terminate.

What is the loop invariant? (This ia a property always that must hold at the loop test.)

- If there is a walk $(\{q_0\}, \ldots\ \{\ldots\ q_i, \ldots\})$ in G_D labeled w, then there is a walk $(q_0, \ldots\ q_i)$ in G_N labeled w.

1.3.4 Linz Example 2.12

Convert the nfa in Linz Figure 2.12 to an equivalent dfa.

Intermediate steps are shown in Figures 2.12-1 and 2.12-2, with the final results in Linz Figure 2.13.

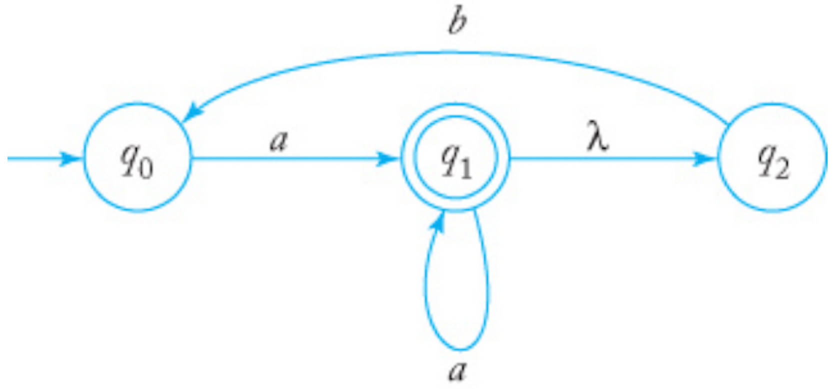


Figure 20: **Linz Fig. 2.12: An NFA**

1.3.5 Linz Example 2.13

Convert the nfa shown in Linz Figure 2.14 into an equivalent dfa.

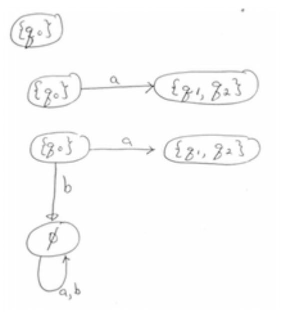


Figure 21: **Intermediate Fig. 2.12-1**

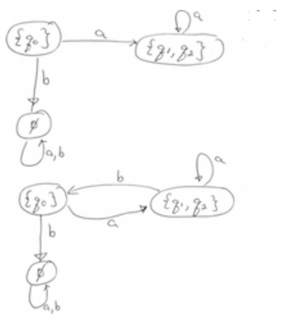


Figure 22: **Intermediate Fig. 2.12-2**

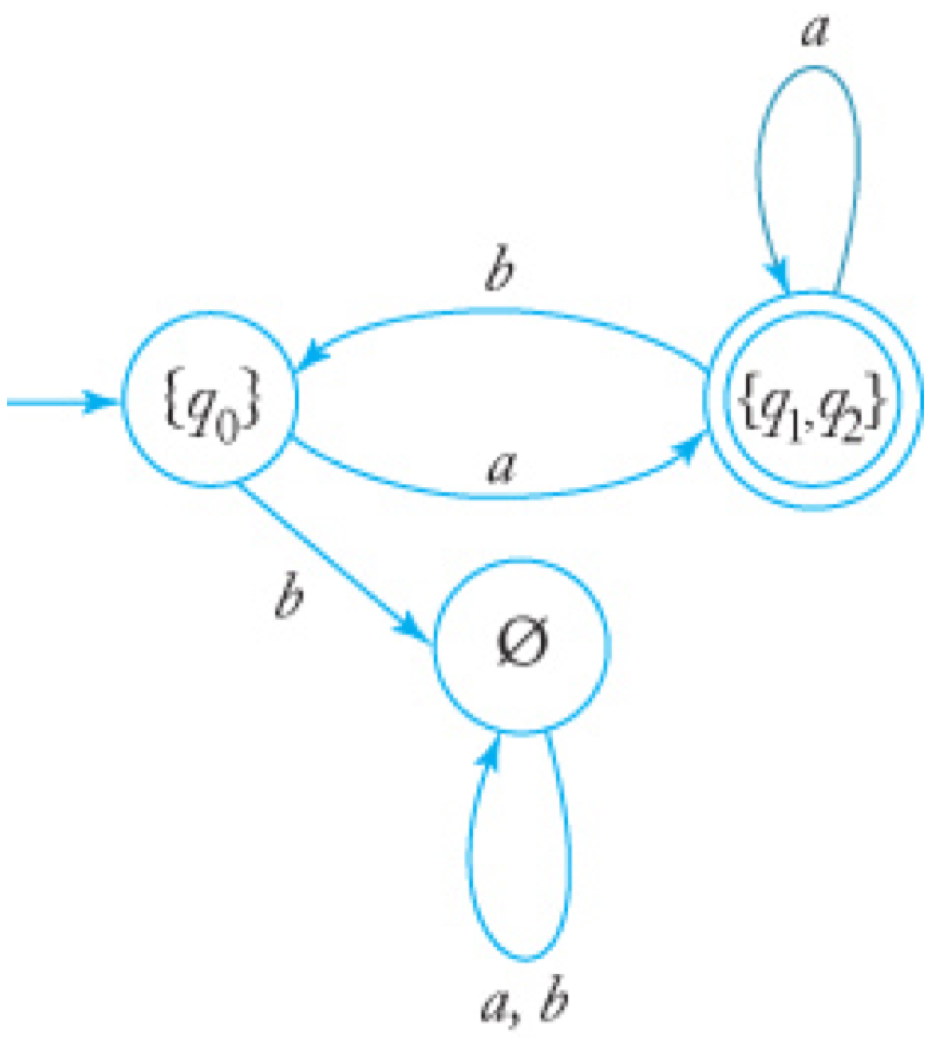


Figure 23: **Linz Fig. 2.13: Corresponding DFA**

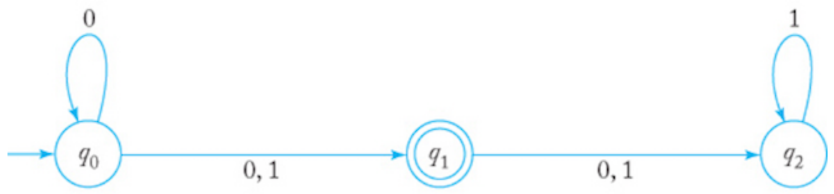


Figure 24: **Linz Fig. 2.14: An NFA**

$\delta_D(\{q_0\}, 0) = \delta_N^*(q_0, 0) = \{q_0, q_1\}$

$\delta_D(\{q_0\}, 1) = \delta_N^*(q_0, 1) = \{q_1\}$

$\delta_D(\{q_0, q_1\}, 0) = \delta_N^*(q_0, 0) \cup \delta_N^*(q_1, 0) = \{q_0, q_1, q_2\}$

$\delta_D(\{q_0, q_1, q_2\}, 1) = \delta_N^*(q_0, 1) \cup \delta_N^*(q_1, 1) \cup \delta_N^*(q_2, 1) = \{q_1, q_2\}$

The above gives us the partially constructed dfa shown in Linz Figure 2.15.

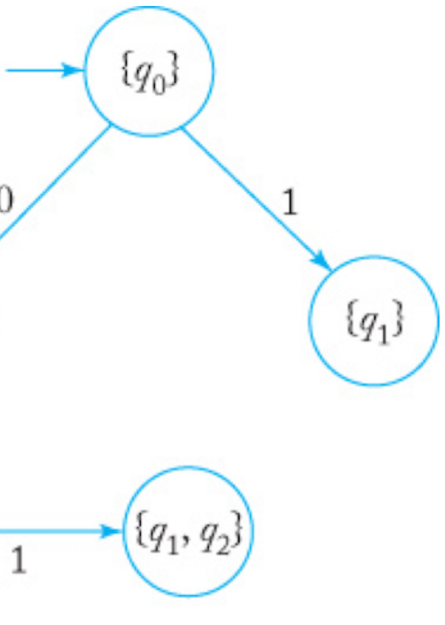


Figure 25: **Linz Fig. 2.15**

$\delta_D(\{q_1\}, 0) = \delta_N^*(q_1, 0) = \{q_2\}$

$\delta_D(\{q_1\}, 1) = \delta_N^*(q_1, 1) = \{q_2\}$

$\delta_D(\{q_2\}, 0) = \delta_N^*(q_1, 0) = \emptyset$

$\delta_D(\{q_2\}, 1) = \delta_N^*(q_2, 1) = \{q_2\}$

$\delta_D(\{q_0, q_1\}, 1) = \delta_N^*(q_0, 1) \cup \delta_N^*(q_1, 1) = \{q_1, q_2\}$

$\delta_D(\{q_0, q_1, q_2\}, 0) = \delta_N^*(q_0, 0) \cup \delta_N^*(q_1, 0) \cup \delta_N^*(q_2, 0) = \{q_0, q_1, q_2\}$

$\delta_D(\{q_1, q_2\}, 0) = \delta_N^*(q_1, 0) \cup \delta_N^*(q_2, 0) = \{q_2\}$

$\delta_D(\{q_1, q_2\}, 1) = \delta_N^*(q_1, 1) \cup \delta_N^*(q_2, 1) = \{q_2\}$

Now, the above gives us the dfa shown in Linz Figure 2.16.

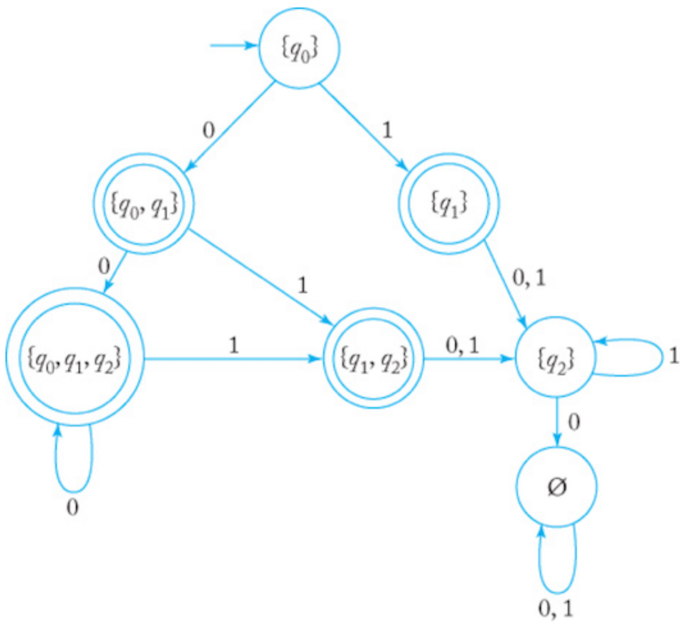


Figure 26: **Linz Fig. 2.16: Corresponding DFA for NFA**

1.4 Reduction in the Number of States in Finite Automata

This section is not covered in this course.

2 Regular Languages and Regular Grammars

Regular languages

- are accepted by dfas and nfas
- but dfas and nfas are **not** concise descriptions

Thus we will examine other notations for representing regular languages.

2.1 Regular Expressions

2.1.1 Syntax

We define the *syntax* (or structure) of regular expressions with an inductive definition.

Linz Definition 3.1 (Regular Expression): Let Σ be a given alphabet. Then:

1. $\emptyset$, λ, and $a \in \Sigma$ are all *regular expressions*. These are called *primitive regular expressions*.
2. If r_1 and r_2 are regular expressions, then $r_1 + r_2$, $r_1 \cdot r_2$, r_1^*, and (r_1) are also regular expressions.
3. A string is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules in (2).

We use the the regular expression operators as follows:

- $r + s$ represents the *union* of two regular expressions.
- $r \cdot s$ is the *concatenation* of two regular expressions.
- r^* is the *star closure* of a regular expression.
- (r) is the same as regular expression r. It is parenthesized to express the order of operations explicitly.

For example, consider regular expression $(a + (b \cdot c))^*$ over the alphabet $\{a, b, c\}$. Note the use of parentheses.

- a, b, and c are *primitive regular expressions*.
- $(b \cdot c)$ is a *concatenation* of regular expressions a and b.
- $(a + (b \cdot c))$ is *union* of regular expressions a and $(b \cdot c)$.
- $(a + (b \cdot c))^*$ is the *star-closure* of regular expression $(a + (b \cdot c))$.

As with arithmetic expressions, *precedence rules* and conventions can be used to relax the need for parentheses.

- *Star-closure* (*) has a higher precedence (i.e., priority or binding power) than concatenation ($\cdot$). That is, $r \cdot s^*$ is equal to $r \cdot (s^*)$, not $(r \cdot s)^*$.
- *Concatenation* ($\cdot$) higher precedence than union ($+$). That is, $r \cdot s + t$ is equal to $(r \cdot s) + t$, not $r \cdot (s + t)$. And, transitively, *star-closure* has a higher precedence than concatenation.
- *Concatenation* operator ($\cdot$) can usually be omitted. That is, rs means $r \cdot s$.

A string $(a + b+)$ is *not* a regular expression. It cannot be generated using the above definition (as augmented by the precedence rules and convention).

2.1.2 Languages Associated with Regular Expressions

But what do we "mean" by a regular expression? That is, what is its *semantics*.

In particular, what languages do regular expressions describe?

Consider the regular expression $(a+(b\cdot c))^*$ from above. As implied by the names for the operators, we intend this regular expression to represent the language $(\{a\} \cup \{bc\})^*$ which is $\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, bcaa, \ldots\}$.

We again give an inductive definition for the language described by a regular expression. It must consider all the cases given in the definition of regular expression itself.

Linz Definition 3.2: The *language* $L(r)$ denoted by any regular expression r is defined (inductively) by the following rules.

Base cases:

1. $\emptyset$ is a regular expression denoting the empty set.
2. λ is a regular expression denoting $\{\lambda\}$.
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

Inductive cases: If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$
6. $L((r_1)) = L(r_1)$
7. $L(r_1^*) = (L(r_1))^*$

2.1.3 Linz Example 3.2

Show the language $L(a^* \cdot (a + b))$ in set notation.

$$\begin{array}{ll}
 & L(a^* \cdot (a+b)) \\
= & \{ \text{ Rule 5 } \} \\
 & L(a^*)L(a+b) \\
= & \{ \text{ Rule 7 } \} \\
 & (L(a))^* L(a+b)
\end{array}$$

$$\begin{array}{ll}
= & \{ \text{ Rule 4 } \} \\
 & (L(a))^*(L(a) \cup L(b)) \\
= & \{ \text{ definition of star-closure of languages } \} \\
 & \{\lambda, a, aa, aaa, \ldots\}\{a, b\} \\
= & \{ \text{ definition of concatenation of languages } \} \\
 & \{a, aa, aaa, ..., b, ab, aab, aaab, \ldots\}
\end{array}$$

2.1.4 Examples of Languages for Regular Expressions

Consider the languages for the following regular expressions.

$L(a^* \cdot b \cdot a^* \cdot b \cdot (a+b)^*) = \{a\}^*\{b\}\{a\}^*\{b\}\{a,b\}^*$
$\quad = \{w : w \in \{a,b\}^*, n_b(w) \geq 2\}$
$L((a+b)^* \cdot b \cdot a^* \cdot b \cdot a^*) = \{a,b\}^*\{b\}\{a\}^*\{b\}\{a\}^*$
$\quad =$ same as above
$L((a+b)^* \cdot b \cdot (a+b)^* \cdot b \cdot (a+b)^*) = \{a,b\}^*\{b\}\{a,b\}^*\{b\}\{a,b\}^*$
$\quad =$ same as above

2.1.5 Linz Example 3.4

Consider the regular expression $r = (aa)^*(bb)^*b$.

- This expression denotes the set of all strings with an even number of a's followed by an odd number of b's.
- In set notation, $L(r) = \{a^{2n}b^{2m+1} : n \geq 0, m \geq 0\}$.

2.1.6 Linz Example 3.5

For $\Sigma = \{0, 1\}$, give a regular expression r such that $L(r) = \{w \in \Sigma^* : w \text{ has at least one pair of consecutive zeros } \}$.

- 00 must appear somewhere in any string.
- Before and after 00 there is an arbitrary string $(0+1)^*$.
- $r = (0+1)^*00(0+1)^*$

2.1.7 Examples of Regular Expressions for Languages

Show regular expressions on the alphabet $\{a, b\}$ for the following languages.

- **exactly one "a"** b^*ab^*
- **at least one "a"** $b^*a(a+b)^*$ – featuring first a
  $(a+b)^*a(a+b)^*$ – featuring middle a
  $(a+b)^*ab^*$ – featuring last a
- **at most one "a"** $b^*ab^* + b^*$
  $b^*(a+\lambda)b^*$

- **all a's immediately followed by a b** $(b^*abb^*)^* + b^*$

2.2 Connection Between Regular Expressions and Regular Languages

2.2.1 Regular Expressions Denote Regular Languages

Regular expressions provide a convenient and concise notation for describing regular languages.

Linz Theorem 3.1 (NFAs for Regular Expressions): Let r be a regular expression. Then there exists some nondeterministic finite accepter (nfa) that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof Sketch: Show that any regular expression generated from the inductive definition corresponds to an nfa. Here we proceed informally.

Linz Figure 3.1 diagrammatically demonstrates that there are nfas that correspond to the primitive regular expressions.

(a) nfa accepts $\emptyset$
(b) nfa accepts $\{\lambda\}$
(c) nfa accepts $\{a\}$

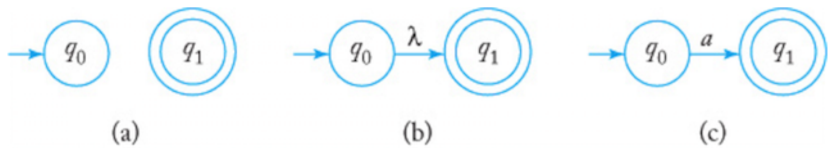


Figure 27: **Linz Fig. 3.1: Primitive Regular Expressions as NFA**

Linz Figure 3.2 shows a general scheme for a nondeterministic finite accepter (nfa) that accepts $L(r)$, with an initial state and one final state.

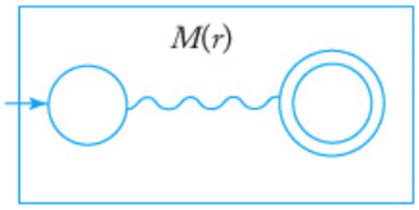


Figure 28: **Linz Fig. 3.2: Scheme for NFA Accepting L(r)**

Linz Figure 3.3 gives an nfa for $L(r_1 + r_2)$. Note the use of λ-transitions to connect the two machines to the new initial and final states.

Linz Figure 3.4 shows an nfa for $L(r_1 r_2)$. Again note the use of λ-transitions to connect the two machines to the new initial and final states.

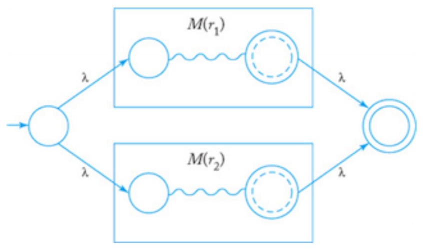


Figure 29: **Linz Fig. 3.3: NFA for Union**

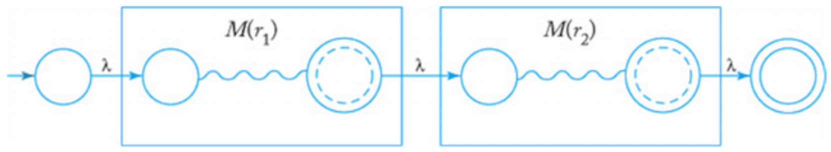


Figure 30: **Linz Fig. 3.4: NFA for Concatenation**

Linz Figure 3.5 shows an nfa for $L(r_1^*)$. Note the use of λ-transitions to represent zero-or-more repetitions of the machine and to connect it to the new initial and final states.

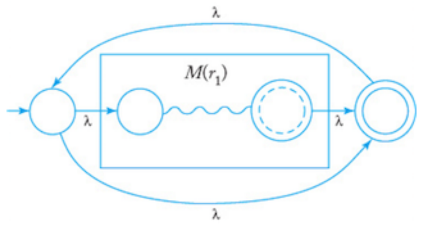


Figure 31: **Linz Fig. 3.5: NFA for Star-Closure**

Thus, Linz Figures 3.3 to 3.5 illustrate composing nfas for any regular expression from the nfas for its subexpressions. Of course, the initial and final states of components are replaced by the initial and final states of the composite nfa.

2.2.2 Linz Example 3.7

Show an nfa that accepts $r = (a + bb)^*(ba^* + \lambda)$.

Linz Figure 3.6, part (a), shows M_1 that accepts $L(a + bb)$. Part (b) shows M_2 that accepts $L(ba^* + \lambda)$.

Linz Figure 3.7 shows an nfa that accepts $L((a + bb)^*(ba^* + \lambda)$.

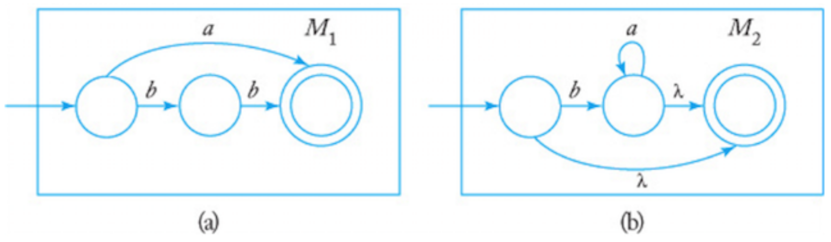


Figure 32: **Linz Fig. 3.6: Toward a Solution to Ex. 3.6**

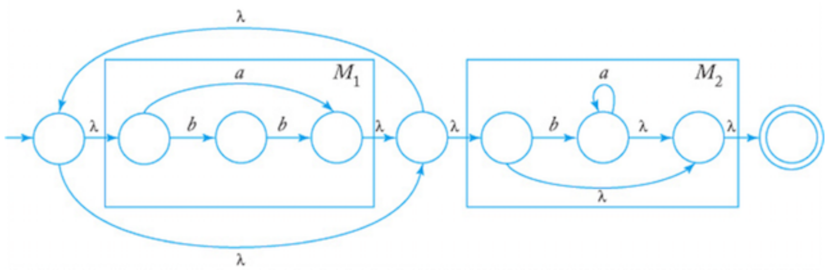


Figure 33: **Linz Fig. 3.7: Solution for Ex. 3.6**

2.2.3 Converting Regular Expressions to Finite Automata

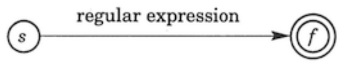
The construction in the proof sketch and example above suggest an algorithm for converting regular expressions to nfas.

This algorithm is adapted from pages 273-4 of the book: James L. Hein, *Theory of Computation: An Introduction*, Jones and Bartlett, 1996.

The diagrams in this section are from the Hein book, which uses a slightly different notation than the Linz book. In particular, these diagrams use capital letters for the expressions.

Algorithm to convert a regular expression to an nfa

- Start with a "machine" with a single start state, a single final state, and a connecting edge labeled with the regular expression.

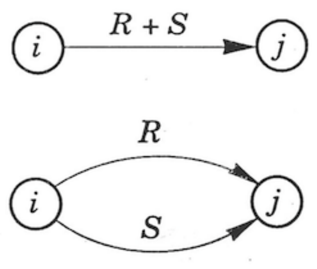


- While there are edges labeled with regular expressions other than elements of the alphabet or λ apply any of the following rules that are applicable:

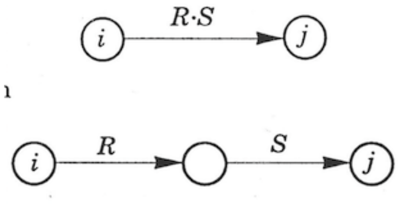
  1. If an edge is labeled with $\emptyset$, then remove the edge.

  2. If an edge is labeled with $r + s$, then replace the edge with two edges labeled with r and s connecting the same source and destination

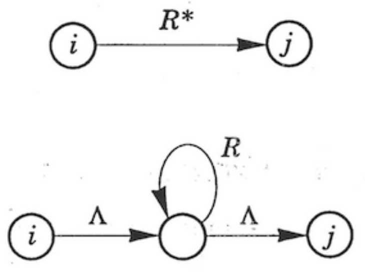
states.



3. If an edge is labeled with $r \cdot s$, the replace the edge with an edge labeled r connecting the source to a new intermediate state, followed by an edge labeled s connecting the intermediate state to the destination.



4. If an edge is labeled with r^*, then replace the edge with a new intermediate state with a self-loop labeled r with edges labeled λ connecting the source to the intermediate state and the intermediate state to the destination.

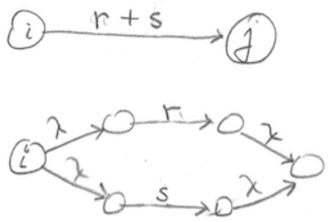


End of Algorithm

Rule 2 in the above algorithm can result in an unbounded number of edges originating at the same state. This makes the algorithm difficult to implement. To remedy this situation, replace Rule 2 as follows.

2. If an edge is labeled with $r + s$, then replace the edge with subgraphs for each of r and s. The subgraph for r consists of with a new source state connected to a new destination state with an edge labeled r. Add

edges labeled λ to connect the original source state to the new source state and the original destination state to the new destination state. Proceed similarly for s.

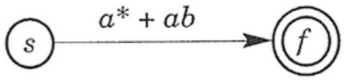


2.2.4 Example Conversion of Regular Expression to NFA

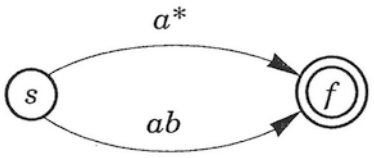
This example is from page 275 of the Hein textbook cited above.

Construct an nfa for $a^* + a \cdot b$.

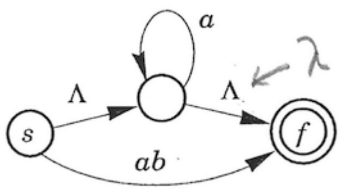
Start with a the two-state initial diagram.



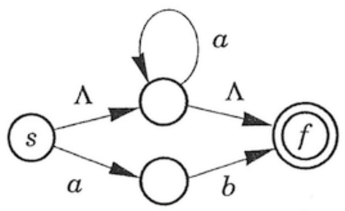
Next, apply Rule 2 to $a^* + a \cdot b$.



Next, apply Rule 4 to a^*.



Finally, apply Rule 3 to $a \cdot b$.



2.2.5 Converting Finite Automata to Regular Expressions

The construction in the proof sketch and example above suggest an algorithm for converting finita automata to regular expressions.

This algorithm is adapted from page 276 of the book: James L. Hein, *Theory of Computation: An Introduction*, Jones and Bartlett, 1996.

Algorithm to convert a finite automaton to a regular expression

Begin with a dfba or an nfa.

1. Create a new start state s and connect this to the original start state with an edge labeled λ.
2. Create a new final state f and connect the original final states to this state by edges labeled λ.
3. For each pair of states i and j that has more than one edge connecting them, replace all the edges with the regular expression formed using union (+) to combine the labels on the previous edges.
4. Construct a sequence of new machines by eliminating one state at a time until the only states remaining are s and f. To eliminate some state k, construct a new machine as follows.
    - Let old(i, j) represent the label on the edge (i, j) on the current (i.e., old) machine.
    - If there is no edge (i, j), then set old$(i, j) = \emptyset$.
    - For every pair of edges (i, k) and (k, j), where $i \neq k$ and $j \neq k$, calculate a new edge label new(i, j) as follows:

      new(i, j) = old(i, j) + old(i, k) old$(k, k)^*$ old(k, j)
    - For all other edges (i, j), where $i \neq k$ and $j \neq k$, set:

      new(i, j) = old(i, j).
    - The states of the new machine are the states of the old machine with state k eliminated. The edges of the new machine are the (i, j) where the new(i, j) has been calculated.

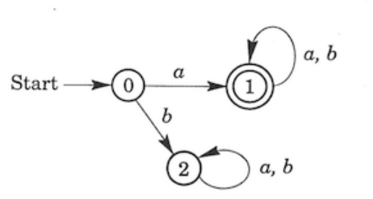
After eliminating all states except s and f, the regular expression is the label on the one edge remaining.

End of Algorithm

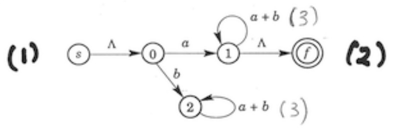
2.2.6 Example Conversion of Finite Automata to Regular Expressions

This example is from pages 277-8 of the Hein textbook cited above.

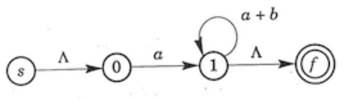
Consider the following dfa.



After applying Rule 1 (new start state), Rule 2 (new final state), and Rule 3 (create union), we get the following machine.



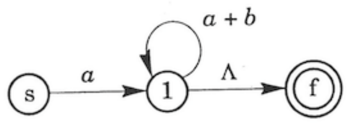
We can eliminate state 2 readily because it is trap state. That is, there is no path through 2 between edges adjacent to 2, so new(i, j) = old(i, j) for any states $i \neq 2$ and $j \neq 2$. The resulting machine is as follows.



To eliminate state 0, we construct a new edge that is labeled as follows:

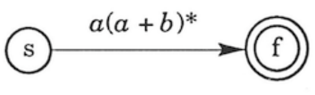
- **new**$(s, 1)$ $=$ old$(s, 1)$ + old$(s, 0)$ old$(0, 0)^*$ old$(0, 1)$
  $= \emptyset + \lambda\emptyset^* a$
  $= a$

Thus, we can eliminate state 0 and its edges and add a new edge $(s, 1)$ labeled a.



We can eliminate state 1 by adding a new edge (s, f) labeled as follows

- **new**(s, f) $=$ old(s, f) $+$ old$(s, 1)$ old$(1, 1)^*$ old$(1, f)$
  $= \emptyset + a(a + b)^*\lambda$
  $= a(a + b)^*$

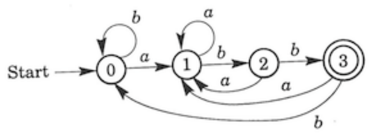


Thus the regular expression is $a(a + b)^*$.

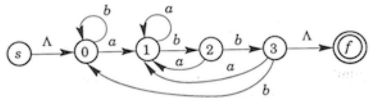
2.2.7 Another Example Conversion of Finite Automa to Regular Expressions

This example is from pages 277-8 of the Hein textbook cited above.

Consider the following dfa. Verify that it corresponds to the regular expression $(a + b)^*abb$.

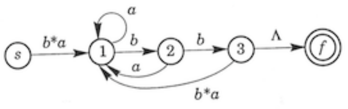


Applying Rules 1 and 2 (adding new start and final states), we get the following machine.



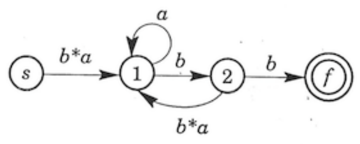
To eliminate state 0, we add the following new edges.

- new$(s, 1) = \emptyset + \lambda b^*a = b^*a$
- new$(3, 1) = a + bb^*a = (\lambda + bb^*)a = b^*a$



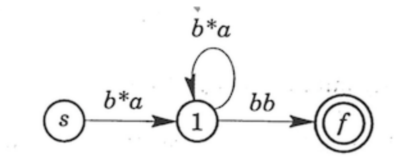
We can eliminate either state 2 or state 3 next. Let's choose 3. Thus we create the following new edges.

- new$(2, f) = \emptyset + b\emptyset^*\lambda = b$
- new$(2, 1) = a + b\emptyset^*b^*a = a + bb^*a = (\lambda + bb^*)a = b^*a$



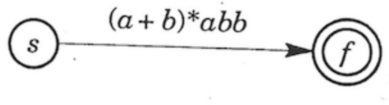
Now we eliminate state 2 and thus create the new edges.

- new$(1, f) = \emptyset + b\emptyset^*b = bb$
- new$(1, 1) = a + b\emptyset^*b^*a = (\lambda + bb^*)a = b^*a$



Finally, we remove state 1 by creating a new edge.

- **new**$(s, f) = \emptyset + b^*a(b^*a)^*bb$
  $= b^*(b^*a)^*abb$
  $= (a + b)^*abb$



2.2.8 Regular Expressions for Describing Simple Patterns

Pascal integer constants

Regular expression sdd^* where

- s : sign from $\{+, -, \lambda\}$
- d : digit from $\{0, 1, ..., 9\}$

Pattern matching

- Unix ed $/aba^*c/$ (different syntax)
- Find pattern in text

Program for Pattern Matching

We can convert a regular expression to an equivalent nfa, the nfa to a dfa, and the dfa to a transition table. We can use the transition table to drive a program for pattern matching.

For a more effiicent program, we can apply the *state reduction algorithm* to the dfa before converting to a transition table. Linz section 2.4, which we did not cover this semester, discusses this algorithm.

2.3 Regular Grammars

We have studied two ways of describing regular languages–finite automata (i.e. dfas, nfas) and regular expressions. Here we examine a third way–*regular grammars*.

Linz Definition 3.3 (Right-Linear Grammar): A grammar $G = (V, T, S, P)$ is said to be *right-linear* if all productions are of one of the forms

$A \rightarrow xB$
$A \rightarrow x$

where $A, B \in V$ and $x \in T^*$.

Similarly, a grammar is said to be *left-linear* if all productions are of the form $A \rightarrow Bx$ or $A \rightarrow x$.

A *regular grammar* is one that is either right-linear or left-linear.

- one variable on right at most
- consistently rightmost (or leftmost)

2.3.1 Linz Example 3.13

The grammar $G_1 = (\{S\}, \{a, b\}, S, P_1)$, with P_1 given as

- $S \rightarrow abS \mid a$

is right-linear.

The grammar $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$, with productions

- $S \rightarrow S_1ab$
- $S_1 \rightarrow S_1ab \mid S_2$
- $S_2 \rightarrow a$

is left linear. Both G_1 and G_2 are regular grammars.

$L(G_1) = L((ab)^*a)$

$L(G_2) = L(aab(ab)^*)$

2.3.2 Linz Example 3.14

The grammar $G = (\{S, A, B\}, \{a, b\}, S, P)$ with productions

- $S \rightarrow A$
- $A \rightarrow aB \mid \lambda$
- $B \rightarrow Ab$

is *not regular*.

Although every production is either in right-linear or left-linear form, the grammar itself is neither right-linear nor left-linear, and therefore is not regular. The grammar is an example of a linear grammar.

Definition (Linear Grammar): A *linear grammar* is a grammar in which at most one variable can appear on the right side of any production.

2.3.3 Right-Linear Grammars Generate Regular Languages

Linz Theorem 3.3 (Regular Languages for Right-Linear Grammars): Let $G = (V, T, S, P)$ be a right-linear grammar. Then $L(G)$ is a *regular language*.

Strategy: Because a regular language is any language accepted by a dfa or nfa, we seek to construct an nfa that simulates the derivations of the right-linear grammar.

The algorithm below incorporates this idea. It is based on the algorithm given on page 314 of the book: James L. Hein, *Theory of Computation: An Introduction*, Jones and Bartlett, 1996.

Algorithm to convert a regular grammar to an nfa

Start with a right-linear grammar and construct an equivalent nfa. We label the nfa's states primarily with variables from the grammar and label edges with terminals in the grammar or λ.

1. If necessary, transform the grammar so that all productions have the form $A \rightarrow x$ or $A \rightarrow xB$, where x is either a terminal in the grammar or λ.
2. Label the start state of the nfa with the start symbol of the grammar.
3. For each production $I \rightarrow aJ$, add a state transition (edge) from a state I to a state J with the edge labeled with the symbol a.
4. For each production $I \rightarrow J$, add a state transition (edge) from a state I to a state J with the edge labeled with λ.
5. If there exist productions of the form $I \rightarrow a$, then add a single new state symbol F. For each production of the form $I \rightarrow a$, add a state transition from I to F labeled with symbol a.
6. The final states of the nfa are F plus all I such there is a production of the form $I \rightarrow \lambda$.

End of algorithm

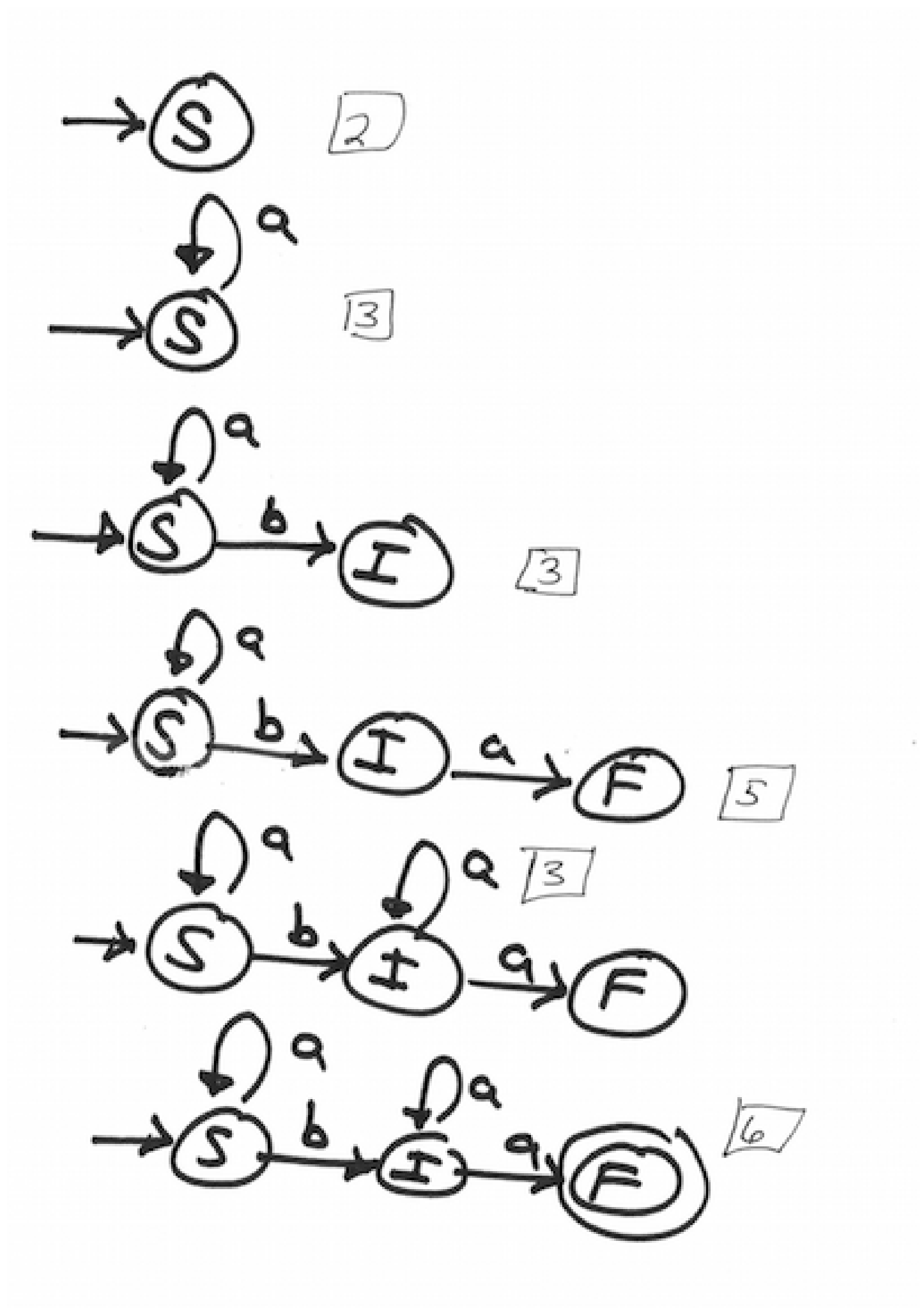
2.3.4 Example: Converting Regular Grammar to NFA

Construct an nfa for the following regular grammar G:

- $S \rightarrow aS \mid bI$

- $I \rightarrow a \mid aI$

The grammar is in the correct form, so step 1 of the grammar is not applicable. The following sequence of diagrams shows the use of steps 2, 3 (three times), 5, and 6 of the algorithm. Step 4 is not applicable to this grammar.



Note that $L(G) = L(a^*ba^*a)$.

2.3.5 Linz Example 3.5

This is similar to the example in the Linz textbook, but we apply the algorithm as stated above.

Construct an nfa for the regular grammar G:

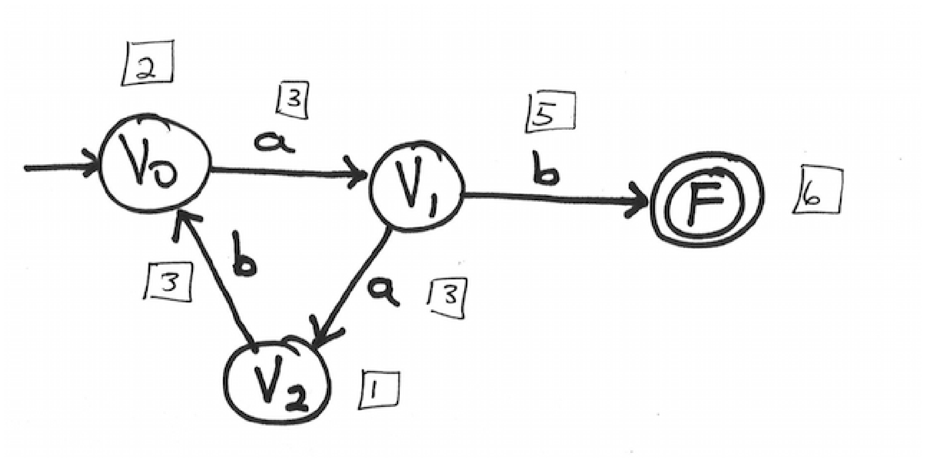
- $V_0 \rightarrow aV_1$

- $V_1 \rightarrow abV_0 \mid b$

First, let's transform the grammar according to step 1 of the regular grammar to nfa algorithm above.

- $V_0 \rightarrow aV_1$
- $V_1 \rightarrow aV_2 \mid b$
- $V_2 \rightarrow bV_0$

Applying steps 2, 3 (three times), 5, and 6 of algorithm as show below, we construct the following nfa. Step 4 was not applicable in this problem.



Note that $L(G) = L((aab)^*ab)$.

2.3.6 Right-Linear Grammars for Regular Languages

Linz Theorem 3.4 (Right-Linear Grammars for Regular Languages): If L is a regular language on the alphabet Σ, then there exists a right-linear grammar $G = (V, \Sigma, S, P)$ such that $L = L(G)$.

Strategy: Reverse the construction of an nfa from a regular grammar given above.

The algorithm below incorporates this idea. It is based on the algorithm given on page 312 of the Hein textbook cited above.

Algorithm to convert an nfa to a regular grammar

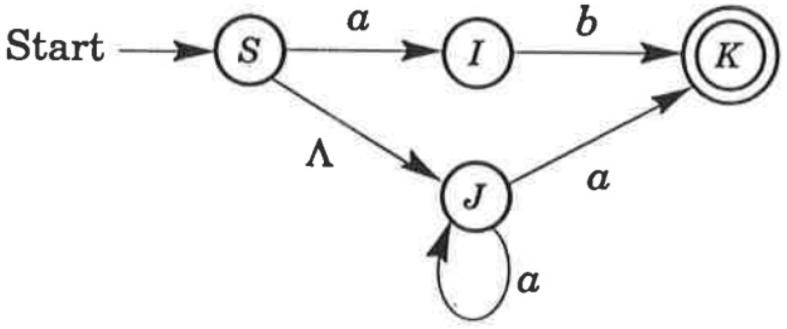
Start with an nfa and construct a regular grammar.

1. Relabel the states of the nfa with capital letters.
2. Make the start state label the start symbol for the grammar.
3. For each transition (edge) from a state I to a state J labeled with an alphabetic symbol a, add a production $I \to aJ$ to the grammar.
4. For each transition (edge) from a state I to a state J labeled with λ, add a production $I \to J$ to the grammar.
5. For each final state labeled K, add a production $K \to \lambda$ to the grammar.

End of algorithm

2.3.7 Example: Converting NFA to Regular Grammar

Consider the following nfa (adapted from the Hein textbook page 313). (The Hein book uses Λ instead of λ to label silent moves and empty strings.)



We apply the steps of the algorithm as follows.

1. The nfa states are already labeled as specified.
2. Choose S as start symbol for grammar.
3. Add the following productions:
   - $S \to aI$
   - $I \to bK$
   - $J \to aJ$
   - $J \to aK$
4. Add the following production:
   - $S \to J$

5. Add the following production:
    - $K \rightarrow \lambda$

So, combining the above productions, we get the final grammar:

- $S \rightarrow aI \mid J$
- $I \rightarrow bK$
- $J \rightarrow aJ \mid aK$
- $K \rightarrow \lambda$

2.3.8 Equivalence Between Regular Languages and Regular Grammars

Linz Theorem 3.5 (Equivalence of Regular Languages and Left-Linear Grammars): A language L is *regular* if and only if there exists a left-linear grammar G such that $L = L(G)$.

Linz Theorem 3.6(Equivalence of Regular Languages and Right-Linear Grammars): A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$.

The four theorems from this section enable us to convert back and forth among finite automata and regular languages as shown in Linz Figure 3.19. Remember that Linz Theorem 2.2 enabled us to translate from nfa to dfa.

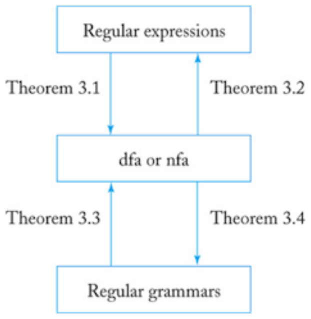


Figure 34: **Linz Fig. 3.19: Equivalence of Regular Languages and Regular Grammars**

3 Properties of Regular Languages

The questions answered in this chapter include:

- What can regular languages *do*?
- What can regular languages *not do*?

The concepts introduced in this chapter are:

- Closure of operations on regular languages
- Membership, finiteness, and equality of regular languages
- Identification of nonregular languages

3.1 Closure Properties of Regular Languages

3.1.1 Mathematical Interlude: Operations and Closure

Definition (Operation): An *operation* is a function $p : V \rightarrow Y$ where $V \in X_1 \times X_2 \times \cdots \times X_k$ for some sets X_i with $0 \leq i \leq k$. k is the number of operands (or arguments) of the operation.

- If $k = 0$, then p is a *nullary* operation.
- If $k = 1$, then p is a *unary* operation.
- If $k = 2$, then p is a *binary* operation.
- etc.

We often use special notation and conventions for unary and binary operations. For example:

- a binary operation may be written in an *infix* style as in $x + y$ and $x \cdot y$
- a unary operation may be written in a *prefix* style as in $-x$, *suffix* style such as x^*, or special style such as $\sqrt{3}$ or $\bar{S}$
- a binary operation may be implied by the juxtaposition such as $3x$ for multiplication or (in a different context) xy for string concatenation or implied by superscripting such as x^2 for exponentiation

Often we consider an operations *on a set*, where all the operands and the result are drawn from the same set.

Definition (Closure): A set S is *closed* under a unary operation p if, for all $x \in S$, $p(x) \in S$. Similarly, a set S is *closed* under a binary operation $\odot$ if, for all $x \in S$ and $y \in S$, $x \odot y \in S$.

Examples arithmetic on the set of natural numbers ($\mathbb{N} = \{0, 1, ...\}$)

- Binary operations addition ($+$) and multiplication ($*$ in programming languages) are closed on $\mathbb{N}$
  - $\forall x, y \in \mathbb{N}, x + y \in \mathbb{N}$
  - $\forall x, y \in \mathbb{N}, x * y \in \mathbb{N}$

- Binary operations subtraction ($-$) and division ($/$) are not closed on $\mathbb{N}$
  - $\exists x, y \in \mathbb{N}, x - y \notin \mathbb{N}$
    For example, $1 - 2$ is not a natural number.
  - $\exists x, y \in \mathbb{N}, x/y \notin \mathbb{N}$
    For example, $3/2$ is not a natural number.
- Unary operation negation (operator $-$ written in prefix form) is not closed on $\mathbb{N}$.

However, the set of *integers* is closed under subtraction and negation. But it is not closed under division or square root (as we normally define the operations).

Now, let's consider closure of the set of regular languages with respect to the simple set operations.

3.1.2 Closure under Simple Set Operations

Linz Theorem 4.1 (Closure under Simple Set Operations): If L_1 and L_2 are regular languages, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 L_2$, $\bar{L_1}$, and L_1^*.

That is, we say that the family of regular languages is *closed* under *union*, *intersection*, *concatenation*, *complementation*, and *star-closure*.

Proof of $L_1 \cup L_2$

Let L_1 and L_2 be regular languages.

$$\begin{array}{ll} & L_1 \cup L_2 \\ = & \{\text{ Th. 3.2: there exist regular expressions } r_1,\, r_2 \text{ }\} \\ & L(r_1) \cup L(r_2) \\ = & \{\text{ Def. 3.2, rule 4 }\} \\ & L(r_1 + r_2) \end{array}$$

Thus, by Theorem 3.1 (regular expressions describe regular languages), the union is a regular language.

Thus $L_1 \cup L_2$ is a regular language. QED.

Proofs of $L_1 L_2$ and L_1^*

Similar to the proof of $L_1 \cup L_2$.

Proof of $\bar{L_1}$

Strategy: Given a dfa M for the regular language, construct a new dfa $\widehat{M}$ that *accepts everything rejected* and *rejects everything accepted* by the given dfa.

$$\begin{array}{ll} & L_1 \text{ is a regular language on } \Sigma. \\ \equiv & \{\text{ Def. 2.3 }\} \end{array}$$

$\exists$ dfa $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = L_1$.

Thus

	$\omega \in \Sigma^*$
$\Rightarrow$	{ by the properties of dfas and sets }
	Either $\delta^*(q_0, \omega) \in F$ or $\delta^*(q_0, \omega) \in Q - F$
$\Rightarrow$	{ Def. 2.2: language accepted by dfa }
	Either $\omega \in L(M)$ or $\omega \in L(\widehat{M})$ for some dfa $\widehat{M}$

Let's construct dfa $\widehat{M} = (Q, \Sigma, \delta, q_0, Q - F)$.

Clearly, $L(\widehat{M}) = \bar{L_1}$. Thus $\bar{L_1}$ is a regular language. QED.

Proof of $L_1 \cap L_2$

Strategy: Given two dfas for the two regular languages, construct a new dfa that accepts a string if and only if both original dfas accept the string.

Let $L_1 = L(M_1)$ and $L_2 = L(M_2)$ for dfas:

$M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$

$M_2 = (P, \Sigma, \delta_2, p_0, F_2)$

Construct $\widehat{M} = (\widehat{Q}, \Sigma, \widehat{\delta}, (q_0, p_0), \widehat{F})$, where

$\widehat{Q} = Q \times P$

$\widehat{\delta}((q_i, p_j), a) = (q_k, p_l)$ **when** $\delta_1(q_i, a) = q_k$

$\delta_2(p_j, a) = p_l$

$\widehat{F} = \{(q, p) : q \in F_1, p \in F_2\}$

Clearly, $\omega \in L_1 \cap L_2$ if and only if ω accepted by $\widehat{M}$.

Thus, $L_1 \cap L_2$ is regular. QED.

The previous proof is *constructive.*

- It establishes desired result.
- It provides an *algorithm* for building an item of interest (e.g., dfa to accept $L_1 \cap L_2$).

Sometimes nonconstructive proofs are shorter and easier to understand. But they provide no algorithm.

Alternate (nonconstructive) proof for $L_1 \cap L_2$

	L_1 and L_2 are regular.
$\equiv$	{ previously proved part of Theorem 4.1 }
	$\bar{L_1}$ and $\bar{L_2}$ are regular.
$\Rightarrow$	{ previously proved part of Theorem 4.1 }
	$\bar{L_1} \cup \bar{L}$ is regular
$\Rightarrow$	{ previously proved part of Theorem 4.1 }
	$\overline{\bar{L_1} \cup \bar{L_2}}$ is regular
$\equiv$	{ deMorgan's Law for sets }
	$L_1 \cap L_2$ is regular

QED.

3.1.3 Closure under Difference (Linz Example 4.1)

Consider the *difference* between two regular languages L_1 and L_2, written $L_1 - L_2$.

But this is just set difference, which is defined $L_1 - L_2 = L_1 \cap \bar{L_2}$.

From Theorem 4.1 above, we know that regular languages are closed under both complementation and intersection. Thus, regular languages are closed under difference as well.

3.1.4 Closure under Reversal

Linz Theorem 4.2 (Closure under Reversal): The family of regular languages is closed under *reversal*.

Proof (constructive)

Strategy: Construct an nfa for the regular language and then reverse all the edges and exchange roles of the initial and final states.

Let L_1 be a regular language. Construct an nfa M such that $L_1 = L(M)$ and M has a single final state. (We can add λ transitions from the previous final states to create a single new final state.)

Now construct a new nfa $\hat{M}$ as follows.

- Make the initial state of M the final state of $\hat{M}$.
- Make the final state of M the initial state of $\hat{M}$.
- Reverse the direction of all edges of M keeping the same labels and add the edges to $\hat{M}$.

Thus nfa $\hat{M}$ accepts $\omega^R \in \Sigma^*$ if and only if the original nfa accepts $\omega \in \Sigma^*$. QED.

3.1.5 Homomorphism Definition

In mathematics, a homomorphism is a mapping between two mathematical structures that *preserves* the essential structure.

Linz Definition 4.1 (Homomorphism): Suppose Σ and Γ are alphabets. A function

$$h : \Sigma \to \Gamma^*$$

is called a *homomorphism*.

In words, a homomorphism is a substitution in which a single letter is replaced with a string.

We can extend the domain of a function h to strings in an obvious fashion. If

$$w = a_1 a_2 \ \cdots \ a_n \text{ for } n \geq 0$$

then

$$h(w) = h(a_1)h(a_2) \ \cdots \ h(a_n).$$

If L is a language on Σ, then we define its *homomorphic image* as

$$h(L) = \{h(w) : w \in L\}.$$

Note: The homomorphism function h *preserves* the essential structure of the language. In particular, it preserves operation concatenation on strings, i.e., $h(\lambda) = \lambda$ and $h(uv) = h(u)h(v)$.

3.1.6 Linz Example 4.2

Let $\Sigma = \{a, b\}$ and $\Gamma = \{a, b, c\}$.

Define h as follows:

$$h(a) = ab,$$

$$h(b) = bbc$$

Then $h(aba) = abbbcab$.

The homomorphic image of $L = \{aa, aba\}$ is the language $h(L) = \{abab, abbbcab\}$.

If we have a regular expression r for a language L, then a regular expression for $h(L)$ can be obtained by simply applying the homomorphism to each Σ symbol of r. We show this in the next example.

3.1.7 Linz Example 4.3

For $\Sigma = \{a, b\}$ and $\Gamma = \{b, c, d\}$, define h:

$$h(a) = dbcc$$

$$h(b) = bdc$$

If L is a regular language denoted by the regular expression

$$r = (a + b^*)(aa)^*$$

then

$$r_1 = (dbcc + (bdc)^*)(dbccdbcc)^*$$

denotes the regular language $h(L)$.

The general result on the closure of regular languages under any homomorphism follows from this example in an obvious manner.

3.1.8 Closure under Homomorphism Theorem

Linz Theorem 4.3 (Closure under Homomorphism): Let h be a homomorphism. If L is a regular language, then its homomorphic image $h(L)$ is also regular.

Proof: Similar to the argument in Example 4.3. See Linz textbook for full proof.

The family of regular languages is therefore closed under arbitrary homomorphisms.

3.1.9 Right Quotient Definition

Linz Definition 4.2 (Right Quotient): Let L_1 and L_2 be languages on the same alphabet. Then the *right quotient* of L_1 with L_2 is defined as

$$L_1/L_2 = \{x : xy \in L_1 \text{ for some } y \in L_2\}$$

3.1.10 Linz Example 4.4

Given languages L_1 and L_2 such that

$$L_1 = \{a^n b^m : n \geq 1, m \geq 0\} \cup \{ba\}$$

$$L_2 = \{b^m : m \geq 1\}$$

Then

$$L_1/L_2 = \{a^n b^m : n \geq 1, m \geq 0\}.$$

The strings in L_2 consist of one or more b's. Therefore, we arrive at the answer by removing one or more b's from those strings in L_1 that terminate with at least one b as a suffix.

Note that in this example L_1, L_2, and L_1/L_2 are regular.

Can we construct a dfa for L_1/L_2 from dfas for L_1 and L_2?

Linz Figure 4.1 shows a dfa M_1 that accepts L_1.

An automaton for L_1/L_2 must accept any x such that $xy \in L_1$ and $y \in L_2$.

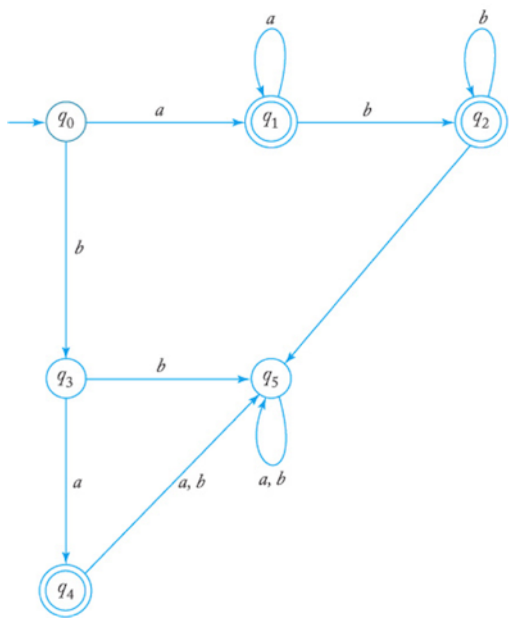


Figure 35: **Linz Fig. 4.1: DFA for Example 4.4 L_1**

For all states $q \in M_1$, if there exists a walk labeled v from q to a final state q_f such that $v \in L_2$, then make q a final state of the automaton for L_1/L_2.

In this example, we check states to see if there is bb^* walk to any of the final states q_1, q_2, or q_4.

- q_1 and q_2 have such walks.
- q_0, q_3, and q_4 do not.

The resulting automaton is shown in Linz Figure 4.2.

The next theorem generalizes this construction.

3.1.11 Closure under Right Quotient

Linz Theorem 4.4 (Closure under Right Quotient): If L_1 and L_2 are regular languages, then L_1/L_2 is also regular. We say that the family of regular languages is *closed under right quotient* with a regular language.

Proof

Let dfa $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = L_1$.

Construct dfa $\widehat{M} = (Q, \Sigma, \delta, q_0, \widehat{F})$ for L_1/L_2 as follows.

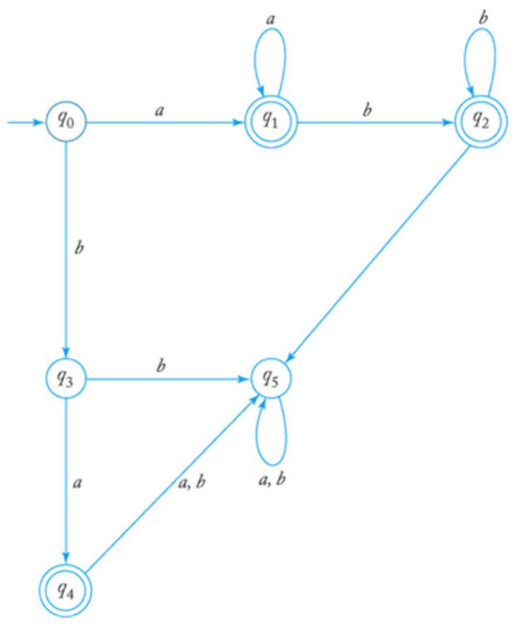


Figure 36: **Linz Fig. 4.2: DFA for Example 4.4 L_1/L_2 EXCEPT q_4 NOT FINAL**

For all $q_i \in Q$, let dfa $M_i = (Q, \Sigma, \delta, q_i, F)$. That is, dfa M_i is the same as M except that it starts at q_i.

- From Theorem 4.1, we know $L(M_i) \cap L_2$ is regular. Thus we can construct the intersection machine as show in the proof of Theorem 4.1.
- If there is any path in the intersection machine from its initial state to a final state, then $L(M_i) \cap L_2 \neq \emptyset$. Thus $q_i \in \widehat{F}$ in machine $\widehat{M}$.

Does $L(\widehat{M}) = L_1/L_2$?

First, let $x \in L_1/L_2$.

- By definition, there must be $y \in L_2$ such that $xy \in L_1$.
- Thus $\delta^*(q_0, xy) \in F$.
- There must be some q such that $\delta^*(q_0, x) = q$ and $\delta^*(q, y) \in F$.
- Thus, by construction, $q \in \widehat{F}$. Hence, $\widehat{M}$ accepts x.

Now, let x be accepted by $\widehat{M}$.

- $\delta^*(q_0, x) = q \in \widehat{F}$.

- Thus, by construction, we know there is a $y \in L_2$ such that $\delta^*(q, y) \in F$.

Thus $L(\widehat{M}) = L_1/L_2$, which means L_1/L_2 is regular.

3.1.12 Linz Example 4.5

Find L_1/L_2 for

$L_1 = L(a^*baa^*)$

$L_2 = L(ab^*)$

We apply the construction (algorithm) used in the proof of Theorem 4.4.

Linz Figure 4.3 shows a dfa for L_1.

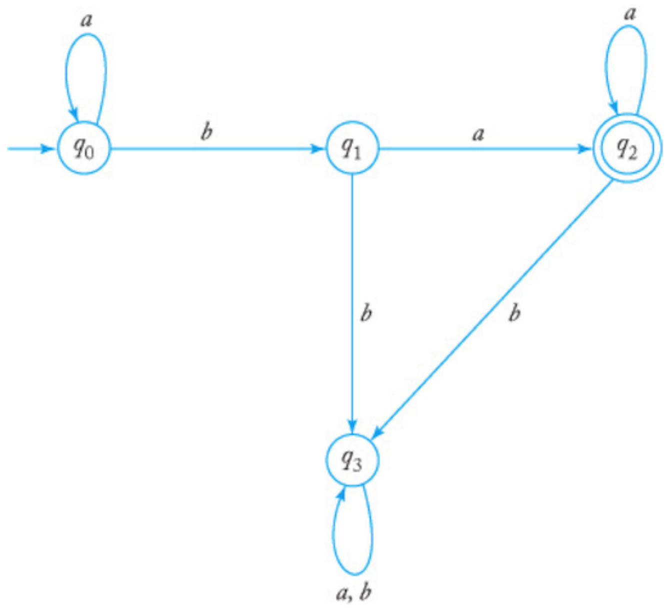


Figure 37: **Linz Fig. 4.3: DFA for Example 4.5** L_1

Let $M = (Q, \Sigma, \delta, q_0, F)$.

Thus if we construct the sequence of machines M_i

$L(M_0) \cap L_2 = \emptyset$

$L(M_1) \cap L_2 = \{a\} \neq \emptyset$

$L(M_2) \cap L_2 = \{a\} \neq \emptyset$

$L(M_3) \cap L_2 = \emptyset$

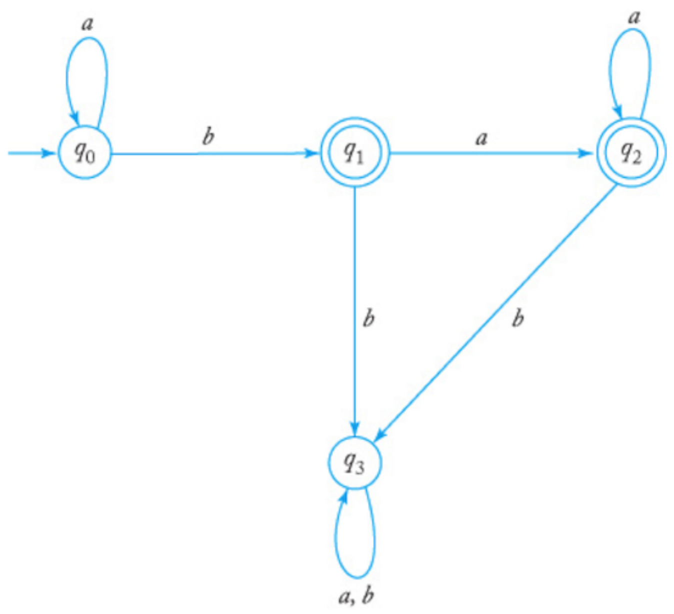


Figure 38: **Linz Fig. 4.4: DFA for Example 4.5** L_1/L_2

then the resulting dfa for L_1/L_2 is shown in Linz Figure 4.4.

The automaton shown in Figure 4.4 accepts the language denoted by the regular expression

$$a^*b + a^*baa^*$$

which can be simplified to

$$a^*ba^*$$

3.2 Elementary Questions about Regular Languages

3.2.1 Membership?

Fundamental question: Is $w \in L$?

It is difficult to find a *membership* algorithm for languages in general. But it is relatively easy to do for regular languages.

A regular language is given in a *standard representation* if and only if described with one of:

- a dfa or nfa
- a regular expression
- a regular grammar

Linz Theorem 4.5 (Membership): Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof

We represent the language by some dfa, then test w to see if it is accepted by this automaton. QED.

3.2.2 Finite or Infinite?

Linz Theorem 4.6 (Finiteness): There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof

Represent L as a transition graph of a dfa.

- If simple path exists from the initial state to any final state, then it is *not empty*. Otherwise, it is *empty*.
- If any vertex on a cycle is in a path from the initial state to any final state, then the language is *infinite*. Otherwise, it is *finite*.

QED.

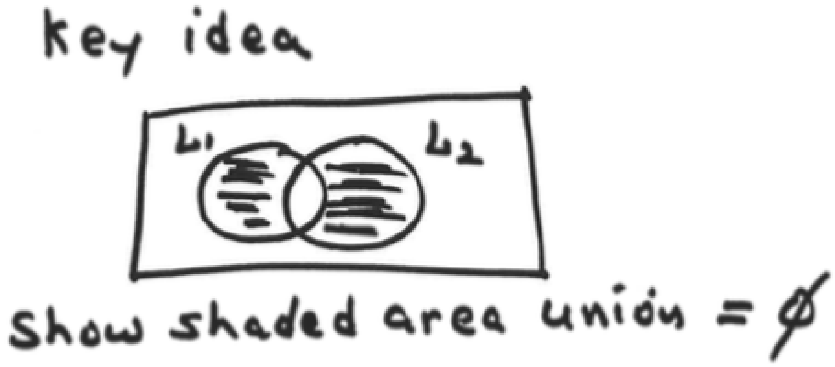
3.2.3 Equality?

Consider the question $L_1 = L_2$?

This is practically important. But it is a difficult issue because there are many ways to represent L_1 and L_2.

Linz Theorem 4.7 (Equality): Given a standard representation of two regular languages L_1 and L_2, there exists an algorithm to determine whether or whether not $L_1 = L_2$.

Proof



Let $L_3 = (L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2)$.

By closure, L_3 is regular. Hence, there is a dfa M that accepts L_3.

Because of Theorem 4.6, we can determine whether L_3 is empty or not.

But from Excerise 8, Section 1.1, we see that $L_3 = \emptyset$ if and only if $L_1 = L_2$. QED.

3.3 Identifying Nonregular Languages

A regular languages may be *infinite*

- but it is accepted by an automaton with *finite* "memory"
- which imposes restrictions on the language.

In processing a string, the amount of information that the automaton must "remember" is strictly limited (finite and bounded).

3.3.1 Using the Pigeonhole Principle

In mathematics, the *pigeonhole principle* refers to the following simple observation:

> If we put n objects into m boxes (pigeonholes), and, if $n > m$, at least one box must hold more than one item.

This is obvious, but it has deep implications.

3.3.2 Linz Example 4.6

Is the language $L = \{a^n b^n : n \geq 0\}$ regular?

The answer is *no*, as we show below.

Proof that L is not regular

Strategy: Use proof by contradiction. Assume that what we want to prove is false. Show that this introduces a contradiction. Hence, the original assumption must be true.

Assume L is regular.

Thus there exists a dfa $M = (Q, \{a, b\}, \delta, q_0, F)$ such that $L(M) = L$.

Machine M has a specific number of states. However, the number of a's in a string in $L(M)$ is finite but *unbounded* (i.e., no maximum value for the length). If n is larger than the number of states in M, then, according to the pigeonhole principle, there must be some state q such that

$$\delta^*(q_0, a^n) = q$$

and

$$\delta^*(q_0, a^m) = q$$

with $n \neq m$. But, because M accepts $a^n b^n$,

$$\delta^*(q, b^n) = q_f \in F$$

for some $q_f \in F$.

From this we reason as follows:

$$\begin{array}{ll} & \delta^*(q_0, a^m b^n) \\ = & \delta^*(\delta^*(q_0, a^m), b^n) \\ = & \delta^*(q, b^n) \\ = & q_f \end{array}$$

But this contradicts the assumption that M accepts $a^m b^n$ only if $n = m$. Therefore, L cannot be regular. QED

We can use the pigeonhole principle to make "finite memory" precise.

3.3.3 Pumping Lemma for Regular Languages

Linz Theorem 4.8 (Pumping Lemma for Regular Languages): Let L be an infinite regular language. There exists some $m > 0$ such that any $w \in L$ with $|w| \geq m$ can be decomposed as

$$w = xyz$$

with

$$|xy| \leq m$$

and

$$|y| \geq 1$$

such that

$$w_i = xy^i z$$

is also in L for all $i \geq 0$.

That is, we can break every sufficiently long string from L into three parts in such a way that an arbitrary number of repetitions of the middle part yields another string in L.

We can "pump" the middle string, which gives us the name *pumping lemma* for this theorem.

Proof

Let L be an infinite regular language. Thus there exists a dfa M that accepts L. Let M have states $q_0, q_1, q_2, \cdots\ q_n$.

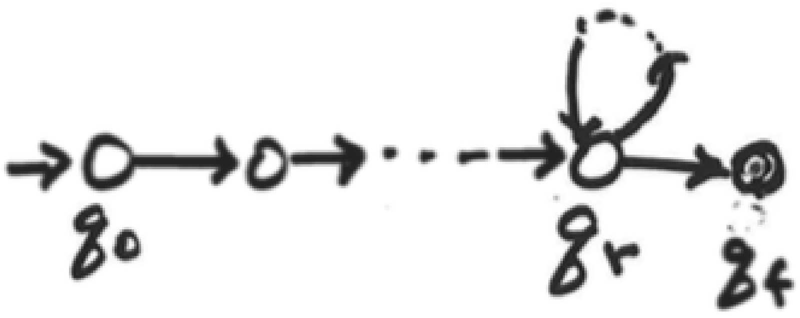
Consider a string $w \in L$ such that $|w| \geq m = n+1$. Such a string exists because L is infinite.

Consider the set of states $q_0, q_i, q_j, \cdots\ q_f$ that M traverses as it processes w.

The size of this set is exactly $|w|+1$. Thus, according to the pigeonhole principle, at least one state must be repeated, and such a repetition must start no later than the nth move.

Thus the sequence is of the form

$$q_0, q_i, q_j, \cdots, q_r, \cdots, q_r, \cdots, q_f.$$



Then there are substrings x, y, and z of w such that

$$\delta^*(q_0, x) = q_r$$

$$\delta^*(q_r, y) = q_r$$

$$\delta^*(q_r, z) = q_f$$

with $|xy| \leq n+1 = m$ and $|y| \geq 1$. Thus, for any $k \geq 0$,

$$\delta^*(q_0, xy^k z) = q_f$$

QED.

We can use the pumping lemma to show that languages are *not regular*. Each of these is a proof by contradiction.

3.3.4 Linz Example 4.7

Show that $L = \{a^n b^n : n \geq 0\}$ is not regular.

Assume that L is regular, so that the Pumping Lemma must hold.

If, for some $n \geq 0$ and $i \geq 0$, $xyz = a^n b^n$ and $xy^i z$ are both in L, then y must be all a's or all b's.

We do not know what m is, but, whatever m is, the Pumping Lemma enables us to choose a string $w = a^m b^m$. Thus y must consist entirely of a's.

Suppose $k > 0$. We must decompose $w = xyz$ as follows for some $p + k \leq m$:

$$x = a^p$$

$$y = a^k$$

$$z = a^{m-p-k} b^m$$

From the Pumping Lemma

$$w_0 = a^{m-k} b^m.$$

Clearly, this is not in L. But this contradicts the Pumping Lemma.

Hence, the assumption that L is regular is false. Thus $\{a^n b^n : n \geq 0\}$ is not regular.

3.3.5 Using the Pumping Lemma (Viewed as a Game)

The Pumping Lemma guarantees the *existence* of m and decomposition xyz for any string in a regular language.

- But we *do not know* what m and xyz are.
- We *do not have contradiction* if the Pumping Lemma is violated for some specific m or xyz.

The Pumping Lemma holds for all $w \in L$ and for all $i \geq 0$ (i.e., $xy^i z \in L$ for all i).

- We *do have a contradiction* if the Pumping Lemma is violated for some w or i.

We can thus conceptualize a proof as a game against an opponent.

- Our goal: Establish a contradiction of the Pumping Lemma.
- Opponent's goal: Stop us.

- Moves:
    1. The opponent picks m.
    2. Given m, we pick a string w in L of length equal or greater than m. We are free to choose any w, subject to requirement $w \in L$ and $|w| \geq m$.
    3. The opponent chooses the decomposition xyz, subject to $|xy| \leq m$ and $|y| \geq 1$. We have to assume that the opponent makes the choice that will make it hardest for us to win the game.
    4. We try to pick i in such a way that the pumped string w_i, as defined in $w_i = xy^iz$, is not in L. If we can do so, we win the game.

Strategy:

- Choose w in step 2 carefully. So that, regardless of the xyz choice, contradiction can be established.

3.3.6 Linz Example 4.8

Let $\Sigma = \{a, b\}$. Show that

$$L = \{ww^R : w \in \Sigma^*\}$$

is not regular.

We use the Pumping Lemma and assume L is regular.

Whatever m the opponent picks in step 1 (of the "game"), we can choose a w as shown below in step 2.

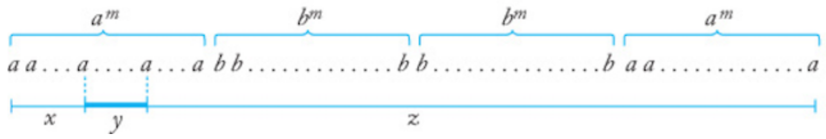

a^m
b^m
b^m
a^m
$a\,a\ldots a\ldots a\ldots a\ b\,b\ldots b\ b\ldots b\ a\,a\ldots a$
x
y
z

Figure 39: **Linz Fig. 4.5**

Because of this choice, and the requirement that $|xy| \leq m$, in step 3 the opponent must choose a y that consists entirely of a's. Consider

$$w_i = xy^iz$$

that must hold because of the Pumping Lemma.

In step 4, we use $i = 0$ in $w_i = xy^iz$. This string has fewer a's on the left than on the right and so cannot be of the form ww^R.

Therefore, the Pumping Lemma is violated. L is not regular.

Warning: Be careful! There are ways we can go wrong in applying the Pumping Lemma.

- If we choose w too short in step 2 of this example (i.e., where the first m symbols include two or more b's), then the opponent can choose a y having an even number of b's. In that case, we could not have reached a violation of the pumping lemma on the last stap.
- If we choose a string w consisting of all a's, say

  $w = a^{2m}$

  which is in L. To defeat us, the opponent need only pick

  $y = aa$

  Now w_i is in L for all i, and we lose. ˆ
- We must assume the opponent does not make mistakes. If, in the case where we pick $w = a^{2m}$, the opponent picks

  $y = a$

  then w_0 is a string of odd length and therefore not in L. But any argument is incorrect if it assumes the opponent fails to make the best possible choice (i.e., $y = aa$).

3.3.7 Linz Example 4.9

For $\Sigma = \{a, b\}$, show that the language

$$L = \{w \in \Sigma^* : n_a(w) < n_b(w)\}$$

is not regular.

We use the Pumping Lemma to show a contradiction. Assume L is regular.

Suppose the opponent gives us m. Because we have complete freedom in choosing $w \in L$, we pick $w = a^m b^{m+1}$. Now, because $|xy|$ cannot be greater than m, the opponent cannot do anything but pick a y with all a's, that is,

$$y = a^k \text{ for } 1 \leq k \leq m.$$

We now pump up, using $i = 2$. The resulting string

$$w_2 = a^{m+k} b^{m+1}$$

is not in L. Therefore, the Pumping Lemma is violated. L is not regular.

3.3.8 Linz Example 4.10

Show that

$$L = \{(ab)^n a^k : n > k, k \geq 0\}$$

is not regular

We use the Pumping Lemma to show a contradiction. Assume L is regular.

Given some m, we pick as our string

$$w = (ab)^{m+1}a^m$$

which is in L.

The opponent must decompose $w = xyz$ so that $|xy| \leq m$ and $|y| \geq 1$. Thus both x and y must be in the part of the string consisting of ab pairs. The choice of x does not affect the argument, so we can focus on the y part.

If our opponent picks $y = a$, we can choose $i = 0$ and get a string not in $L((ab)^*a^*)$ and, hence, not in L. (There is a similar argument for $y = b$.)

If the opponent picks $y = ab$, we can choose $i = 0$ again. Now we get the string $(ab)^m a^m$, which is not in L. (There is a similar argument for $y = ba$.)

In a similar manner, we can counter any possible choice by the opponent. Thus, because of the contradiction, L is not regular.

3.3.9 Linz Example (Factorial Length Strings)

Note: This example is adapted from an earlier edition of the Linz textbook.

Show that

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

is not regular.

We use the Pumping Lemma to show a contradiction. Assume L is regular.

Given the opponent's choice for m, we pick w to be the string $a^{m!}$ (unless the opponent picks $m < 3$, in which case we can use $a^{3!}$ as w).

The possible decompositions $w = xyz$ (such that $|xy| \leq m$) differ only in the lengths of x and y. Suppose the opponent picks y such that

$$|y| = k \leq m.$$

According to the Pumping Lemma, $xz = a^{m!-k} \in L$. But this string can only be in L if there exists a j such that

$$m! - k = j!.$$

But this is impossible, because for $m \geq 3$ and $k \leq m$ we know (see argument below) that

$$m! - k > (m - 1)!.$$

Therefore, L is not regular.

Aside: To see that $m! - k > (m-1)!$ for $m \geq 3$ and $k \leq m$, note that

$$m! - k \geq m! - m = m(m-1)! - m = m((m-1)! - 1) > (m-1)!.$$

3.3.10 Linz Example 4.12

Show that the language

$$L = \{a^n b^k c^{n+k} : n \geq 0, k \geq 0\}$$

is not regular.

Strategy: Instead of using the Pumping Lemma directly, we show that L is related to another language we already know is nonregular. This may be an easier argument.

In this example, we use the closure property under homomorphism (Linz Theorem 4.3).

Let h be defined such that

$$h(a) = a, h(b) = a, h(c) = c.$$

Then

$$\begin{array}{lcl} h(L) & = & \{a^{n+k} c^{n+k} : n+k \geq 0\} \\ & = & \{a^i c^i : i \geq 0\} \end{array}$$

But we proved this language was not regular in Linz Example 4.6. Therefore, because of closure under homomorphism, L cannot be regular either.

Alternative proof by contradiction

Assume L is regular.

Thus $h(L)$ is regular by closure under homomorphism (Linz Theorem 4.3).

But we know $h(L)$ is not regular, so there is a contradiction.

Thus, L is not regular.

3.3.11 Linz Example 4.13

Show that the language

$$L = \{a^n b^l : n \neq l\}$$

is not regular.

We use the Pumping Lemma, but this example requires more ingenuity to set up than previous examples.

Assume L is regular.

Choosing a string $w \in L$ with $m = n = l + 1$ or $m = n = l + 2$ will not lead to a contradiction.

In these cases, the opponent can always choose a decomposition $w = xyz$ (with $|xy| \leq m$ and $|y| \geq 1$) that will make it impossible to pump the string out of the language (that is, pump it so that it has an equal number of a's and b's). For $w = a^{l+1}b^l$, the opponent can chose y to be an even number of a's. For $w = a^{l+2}b^l$, the opponent can chose y to be an odd number of a's greater than 1.

We must be more creative. Suppose we choose $w \in L$ where $n = m!$ and $l = (m+1)!$.

If the opponent decomposes $w = xyz$ (with $|xy| \leq m$ and $|y| = k \geq 1$), then y must consist of all a's.

If we pump i times, we generate string xy^iz where the number of a's is $m! + (i-1)k$

We can contradict the Pumping Lemma if we can pick i such that

$$m! + (i-1)k = (m+1)!.$$

But we can do this, because it is always possible to choose

$$i = 1 + mm!/k.$$

For $1 \leq k \leq m$, the expression $1 + mm!/k$ is an integer.

Thus the generated string has $m! + ((1 + mm!/k) - 1)k$ occurrences of a.

$$\begin{array}{ll}
 & m! + ((1 + mm!/k) - 1)k \\
= & m! + mm! \\
= & m!(m+1) \\
= & (m+1)!
\end{array}$$

This introduces a contradiction of the Pumping Lemma. Thus L is not regular.

Alternative argument (more elegant)

Suppose $L = \{a^n b^l : n \neq l\}$ is regular.

Because of complementation closure, $\bar{L}$ is regular.

Let $L_1 = \bar{L} \cap L(a^*b^*)$.

But $L(a^*b^*)$ is regular and thus, by intersection closure, L_1 is also regular.

But $L_1 = \{a^n b^n : n \geq 0\}$, which we have shown to be nonregular. Thus we have a contradiction, so L is not regular.

3.3.12 Pitfalls in Using the Pumping Lemma

The Pumping Lemma is difficult to understand and, hence, difficult to *apply.*

Here are a few suggestions to avoid pitfalls in use of the Pumping Lemma.

- Do not attempt to use the Pumping Lemma to show a language is regular. Only use it to show a language is *not regular*.
- Make sure you start with a string that is in the language.
- Avoid invalid assumptions about the decomposition of a string w into xyz. Use only that $|xy| \leq m$ and $|y| \geq 1$.

Like most interesting "games", knowledge of the rules for use of the Pumping Lemma is necessary, but it is not sufficient to become a master "player". To master the use of the Pumping Lemma, one must work problems of various difficulties. Practice, practice, practice.

4 Context-Free Languages

In Linz Section 4.3, we saw that not all languages are regular. We examined the Pumping Lemma for Regular Languages as a means to prove that a specific language is not regular.

In Linz Example 4.6, we proved that

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

is not regular.

If we let $a =$ "(" and $b =$ ")", then L becomes a language of nested parenthesis.

This language is in a larger family of languages called the *context-free languages*.

Context-free languages are very important because many practical aspects of programming languages are in this family.

In this chapter, we explore the context-free languages, beginning with *context-free grammars*.

One key process we explore is *parsing*, which is the process of determining the grammatical structure of a sentence generated from a grammar.

4.1 Context-Free Grammars

4.1.1 Definition of Context-Free Grammars

Remember the restrictions we placed on *regular grammars* in Linz Section 3.3:

- The *left side* consists of a single variable.
- The *right side* has a special form.

To create more powerful grammars (i.e., that describe larger families of languages), we relax these restrictions.

For context-free grammars, we maintain the left-side restriction but relax the restriction on the right side.

Linz Definition 5.1 (Context-Free Grammar): A grammar $G = (V, T, S, P)$ is *context-free* if all productions in P have the form

$A \rightarrow x$

where $A \in V$ and $x \in (V \cup T)^*$. A language L is context-free if and only if there is a context-free grammar G such that $L = L(G)$.

The family of regular languages is a subset of the family of context-free languages!

Thus, *context-free grammars*

- enable the right side of a production to be substituted for a variable on the left side at any time in a sentential form

- with no dependencies on other symbols in the sentential form.

4.1.2 Linz Example 5.1

Consider the grammar $G = (\{S\}, \{a, b\}, S, P)$ with productions:

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow \lambda$

Note that this grammar satisfies the definition of context-free.

A possible derivation using this grammar is as follows:

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$$

From this derivation, we see that

$$L(G) = \{ww^R : w \in \{a, b\}^*\}.$$

The language is context-free, but, as we demonstrated in Linz Example 4.8, it is not regular.

This grammar is *linear* because it has at most one variable on the right.

4.1.3 Linz Example 5.2

Consider the grammar G with productions:

$S \rightarrow abB$
$A \rightarrow aaBb$
$B \rightarrow bbAa$
$A \rightarrow \lambda$

Note that this grammar also satisfies the definition of context free.

A possible derivation using this grammar is:

$$\begin{aligned} S &\Rightarrow abB \Rightarrow abbbAa \Rightarrow abbbaaBba \Rightarrow abbbaabbAaba \\ &\Rightarrow abbbaabbaaBbaba \Rightarrow abbbaabbaabbAababa \Rightarrow abbbaabbaabbababa \end{aligned}$$

We can see that:

$$L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}$$

This grammar is also *linear* (as defined in Linz Section 3.3). Although linear grammars are context free, not all context free grammars are linear.

4.1.4 Linz Example 5.3

Consider the language

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

This language is context free. To show that this is the case, we must construct a context-free grammar that generates the language

First, consider the $n = m$ case. This can be generated by the productions:

$S \to aSb \mid \lambda$

Now, consider the $n > m$ case. We can modify the above to generate extra a's on the left.

$S \to AS_1$
$S_1 \to aS_1b \mid \lambda$
$A \to aA \mid a$

Finally, consider the $n < m$ case. We can further modify the grammar to generate extra b's on right.

$S \to AS_1 \mid S_1B$
$S_1 \to aS_1b \mid \lambda$
$A \to aA \mid a$
$B \to bB \mid b$

This grammar is context free, but it is *not linear* because the productions with S on the left are not in the required form.

Although this grammar is not linear, there exist other grammars for this language that are linear.

4.1.5 Linz Example 5.4

Consider the grammar with productions:

$S \to aSb \mid SS \mid \lambda$

This grammar is also context-free but not linear.

Example strings in $L(G)$ include *abaabb*, *aababb*, and *ababab*. Note that:

- a and b are always generated in pairs.
- a precedes the matching b.
- A prefix of a string may contain several more a's than b's.

We can see that $L(G)$ is

$\{\ w \in \{a, b\}^* : n_a(w) = n_b(w)$ and $n_a(v) \geq n_b(v)$ for any prefix v of $w\ \}$.

What is a programming language connection for this language?

- Let $a =$ "(" and $b =$ ")".
- This gives us a language of properly nested parentheses.

4.1.6 Leftmost and Rightmost Derivations

Consider grammar $G = (\{A, B, S\}, \{a, b\}, S, P)$ with productions:

$$\begin{aligned}&S \rightarrow AB\\&A \rightarrow aaA\\&A \rightarrow \lambda\\&B \rightarrow Bb\\&B \rightarrow \lambda\end{aligned}$$

This grammar generates the language $L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}$.

Now consider the two derivations:

$$\begin{aligned}&S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab\\&S \Rightarrow AB \Rightarrow ABb \Rightarrow aaABb \Rightarrow aaAb \Rightarrow aab\end{aligned}$$

These derivations yield the same sentence using exactly the same productions. However, the productions are applied in different orders.

To eliminate such incidental factors, we often require that the variables be replaced in a specific order.

Linz Definition 5.2 (Leftmost and Rightmost Derivations): A derivation is *leftmost* if, in each step, the leftmost variable in the sentential form is replaced. If, in each step, the rightmost variable is replaced, then the derivation is *rightmost*.

4.1.7 Linz Example 5.5

Consider the grammar with productions:

$$\begin{aligned}&S \rightarrow aAB\\&A \rightarrow bBb\\&B \rightarrow A \mid \lambda\end{aligned}$$

A leftmost derivation of the string *abbbb* is:

$$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$$

Similarly, a rightmost derivation of the string *abbbb* is:

$$S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb$$

4.1.8 Derivation Trees

We can also use a *derivation tree* to show derivations in a manner that is independent of the order in which productions are applied.

A derivation tree is an ordered tree in which we label the nodes with the left sides of productions and the children of a node represent its corresponding right sides.

The production

$A \to abABc$

is shown as a derivation tree in Linz Figure 5.1.

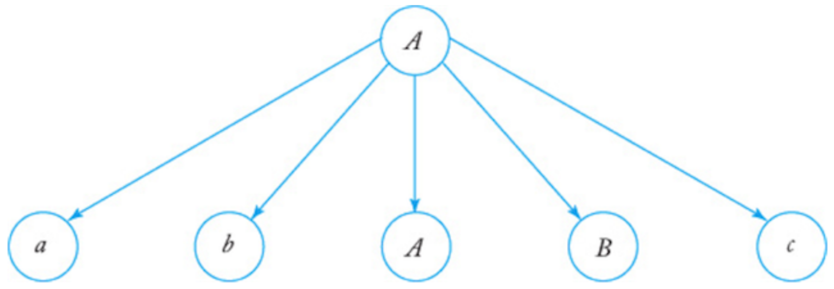


Figure 40: **Linz Fig. 5.1: Derivation Tree for Production** $A \to abABc$

Linz Definition 5.3 (Derivation Tree): Let $G = (V, T, S, P)$ be a context-free grammar. An ordered tree is a *derivation tree* for G if and only if it has the following properties:

1. The root is labeled S.
2. Every leaf has a label from $T \cup \{\lambda\}$.
3. Every interior vertex (i.e., a vertex that is not a leaf) has a label from V.
4. If a vertex has a label $A \in V$, and its children are labeled (from left to right) $a_1, a_2, \cdots, a_n$, then P must contain a production of the form $A \to a_1 a_2 \cdots\ a_n$.
5. A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ can have no other children.

If properties 3, 4, and 5 and modified property 2 (below) hold for a tree, then it is a *partial derivation tree*.

2. (modified) Every leaf has a label from $V \cup T \cup \{\lambda\}$

If we read the leaves of a tree from left to right, omitting any λ's encountered, we obtain a string called the *yield* of the tree.

The descriptive term from *left to right* means that we traverse the tree in a depth-first manner, always exploring the leftmost unexplored branch first. The *yield* is the string of terminals encountered in this traversal.

4.1.9 Linz Example 5.6

Consider the grammar G with productions:

$S \to aAB$
$A \to bBb$
$B \to A \mid \lambda$

Linz Figure 5.2 shows a partial derivation tree for G with the yield $abBbB$. This is a *sentential form* of the grammar G.

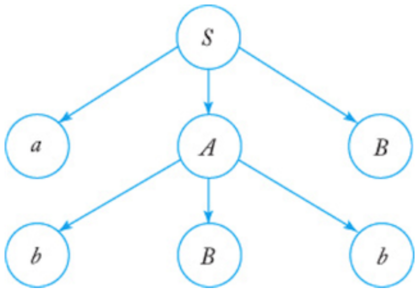


Figure 41: **Linz Fig. 5.2: Partial Derivation Tree**

Linz Figure 5.3 shows a derivation tree for G with the yield $abbbb$. This is a *sentence* of $L(G)$.

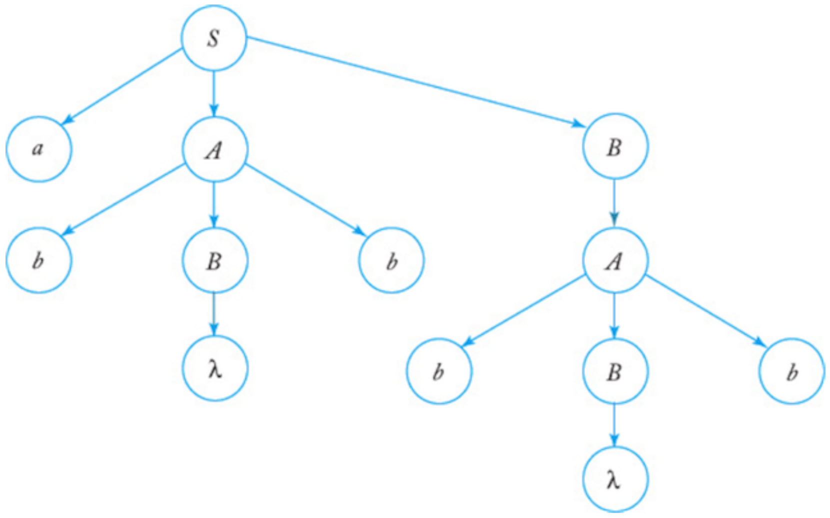


Figure 42: **Linz Fig. 5.3: Derivation Tree**

4.1.10 Relation Between Sentential Forms and Derivation Trees

Derivation trees give explicit (and visualizable) descriptions of derivations. They enable us to reason about context-free languages much as transition graphs enable use to reason about regular languages.

Linz Theorem 5.1 (Connection between Derivations and Derivation Trees): Let $G = (V, T, S, P)$ be a context-free grammar. Then the following properties hold:

- For every $w \in L(G)$, there exists a derivation tree of G whose yield is w.
- The yield of any derivation tree of G is in $L(G)$.
- If t_G is any partial derivation tree for G whose root is labeled S, then the yield of t_G is a sentential form of G.

Proof: See the proof in the Linz textbook.

Derivation trees:

- show which productions are used to generate a sentence
- abstract out the order in which individual productions are applied
- enable the construction of eiher a leftmost or rightmost derivation

4.2 Parsing and Ambiguity

4.2.1 Generation versus Parsing

The previous section concerns the generative aspects of grammars–using a grammar to generate strings in the language.

This section concerns the analytical aspects of grammars–processing strings from the language to determine their derivations. This process is called *parsing*.

For example, a compiler for a programming language must parse a program (i.e., a sentence in the language) to determine the derivation tree for the program.

- This verifies that the program is indeed in the language (syntactically).
- Construction of the derivation tree is needed to execute the program (e.g., to generate the machine-level code corresponding to the program).

4.2.2 Exhaustive Search Parsing

Given some $w \in L(G)$, we can parse w with respect to grammar G by:

- systematically constructing all derivations
- determining whether any derivation matches w

This is called *exhaustive search parsing* or *brute force parsing*. A more complete description of the algorithm is below.

This is a form of *top-down parsing* in which a derivation tree is constructed from the root downward.

Note: An alternative approach is *bottom-up parsing* in which the derivation tree is constructed from leaves upward. Bottom-up parsing techniques often have limitations in terms of the grammars supported but often give more efficient algorithms.

Exhaustive Search Parsing Algorithm

```
– Add root and 1st level of all derivation trees
$F \leftarrow \{x : s \rightarrow x$ in $P$ of $G\}$
while $F \neq \emptyset$ and $w \notin F$ do
    $F' \leftarrow \emptyset$
    – Add next level of all derivation trees
    for all $x \in F$ do
        if $x$ can generate $w$ then
            $V \leftarrow$ leftmost variable of $x$
            for all productions $V \rightarrow y$ in $G$ do
                $F' \leftarrow F' \cup \{x'\}$ where $x' = x$ with $V \leftarrow y$
    $F \leftarrow F'$
```

Note: The above algorithm determines whether a string w is in $L(G)$. It can be modified to build the actual derivation or derivation tree.

4.2.3 Linz Example 5.7

Note: The presentation here uses the algorithm above, rather than the approach in the Linz textbook.

Consider the grammar G with productions:

$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$

Parse the string $w = aabb$.

After initialization: $F = \{SS, aSb, bSa, \lambda\}$ (from the righthand sides of the grammar's four productions with S on the left).

First iteration: The loop test is true because F is nonempty and w is not present.

The algorithm does not need to consider the sentential forms bSa and λ in F because neither can generate w.

The inner loop thus adds $\{SSS, aSbS, bSaS, S\}$ from the leftmost derivations from sentential form SS and also adds $\{aSSb, aaSbb, abSab, ab\}$ from the leftmost derivations from sentential form aSb.

Thus $F = \{SSS, aSbS, bSaS, S, aSSb, aaSbb, abSab, ab\}$ at the end of the first iteration.

Second iteration: The algorithm enters the loop a second time because F is nonempty and does not contain w.

The algorithm does not need to consider any sentential form beginning with b or ab, thus eliminating $\{bSaS, abSab, ab\}$ and leaving $\{SSS, aSbS, S, aSSb, aaSbb\}$ of interest.

This iteration generates 20 new sentential forms from applying each of the 4 productions to each of the 5 remaining sentential forms.

In particular, note that that sentential form $aaSbb$ yields the target string $aabb$ when production $S \to \lambda$ is applied.

Third iteration: The loop terminates because w is present in F.

Thus we can conclude $w \in L(G)$.

4.2.4 Flaws in Exhaustive Search Parsing

Exhaustive search parsing has serious flaws:

- It is tedious and inefficient.
- It might not terminate when $w \notin L(G)$.

For example, if we choose $w = abb$ in the previous example, the algorithm goes into an infinite loop.

The fix for nontermination is to ensure sentential forms increase in length for each production. That is, we eliminate productions of forms:

$A \to \lambda$
$A \to B$

Chapter 6 of the Linz textbook (which we will not cover this semester) shows that this does not reduce the power of the grammar.

4.2.5 Linz Example 5.8

Consider the grammar with productions:

$S \to SS \mid aSb \mid bSA \mid ab \mid ba$

This grammar generates the same language as the one in Linz Example 5.7 above, but it satisfies the restrictions given in the previous subsection.

Given any nonempty string w, exhaustive search will terminate in no more than $|w|$ rounds for such grammars.

4.2.6 Toward Better Parsing Algorithms

Linz Theorem 5.2 (Exhaustive Search Parsing): Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of one of the forms

$A \to \lambda$
$A \to B$

where $A, B \in V$. Then the exhaustive search parsing method can be formulated as an algorithm which, for any $w \in T*$, either parses w or tells us that parsing is impossible.

Proof outline

- Each production must increase either the length or number of terminals.
- The maximum length of a sentential form is $|w|$, which is the maximum number of terminal symbols.
- Thus for some w, the number of loop iterations is at most $2|w|$.

But exhaustive search is *still inefficient.* The number of sentential forms to be generated is

$\sum_{i=1}^{2|w|} |P|^i$.

That is, it grows exponentially with the length of the string.

Linz Theorem 5.3 (Efficient Parsing): For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

- Construction of more efficient context-free parsing methods is left to compiler courses.
- $|w|^3$ is still inefficient.
- We would prefer linear ($|w|$) parsing.
- Again we must restrict the grammar in our search for more efficient parsing. The next subsection illustrates on such grammar.

4.2.7 Simple Grammar Definition

Linz Definition 5.4 (Simple Grammar): A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$A \rightarrow ax$

where $A \in V, a \in T, x \in V^*$, and any pair (A, a) occurs at most once in P.

4.2.8 Linz Example 5.9

The grammar

$S \rightarrow aS \mid bSS \mid c$

is an s-grammar.

The grammar

$S \rightarrow aS \mid bSS \mid aSS \mid c$

is *not* an s-grammar because (S, a) occurs twice.

4.2.9 Parsing Simple Grammars

Although s-grammars are quite restrictive, many features of programming languages can be described with s-grammars (e.g., grammars for arithmetic expressions).

If G is s-grammar, then $w \in L(G)$ can be parsed in linear time.

To see this, consider string $w = a_1 a_2 \cdots a_n$ and use the exhaustive search parsing algorithm.

1. The s-grammar has at most one rule with a_1 on left: $S \rightarrow a_1 A_1 A_2 \cdots$. *Choose it!*
2. Then the s-grammar has at most one rule with a_2 on left: $A_1 \rightarrow a_2 B_1 B_2 \cdots$. *Choose it!*
3. And so forth up to the nth terminal.

The number of steps is proportional to $|w|$ because each step consumes one symbol of w.

4.2.10 Ambiguity in Grammars and Languages

A derivation tree for some string generated by a context-free grammar may not be unique.

Linz Definition 5.5 (Ambiguity): A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

4.2.11 Linz Example 5.10

Again consider the grammar in Linz Example 5.4. Its productions are

$$S \rightarrow aSb \mid SS \mid \lambda.$$

The string $w = aabb$ has two derivation trees as shown in Linz Figure 5.4

The left tree corresponds to the leftmost derivation $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

The right tree corresponds to the leftmost derivation $S \Rightarrow SS \Rightarrow \lambda S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

Thus the grammar is ambiguous.

4.2.12 Linz Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\}$$
$$T = \{a, b, c, +, *, (,)\}$$

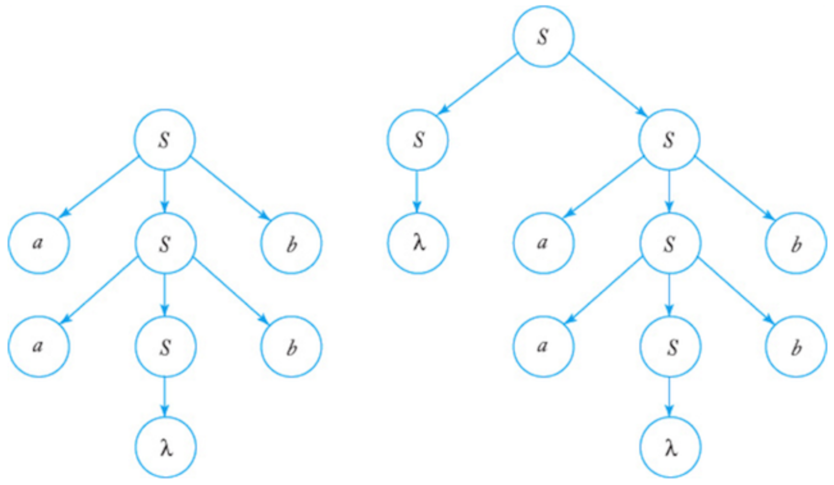


Figure 43: **Linz Fig. 5.4: Two Derivation Trees for** $aabb$

and P including the productions:

$E \to I$
$E \to E + E$
$E \to E * E$
$E \to (E)$
$I \to a \mid b \mid c$

This grammar generates a subset of the arithmetic expressions for a language like C or Java. It contains strings such as $(a + b) * c$ and $a * b + c$.

Linz Figure 5.5 shows two derivation trees for the string $a + b * c$. Thus this grammar is ambiguous.

Why is ambiguity a problem?

Remember that the semantics (meaning) of the expression is also associated with the structure of the expression. The structure determines how the (machine language) code is generated to carry out the computation.

How do real programming languages resolve this ambiguity?

Often, they add *precedence rules* that give priority to "$*$" over "$+$". That is, the multiplication operator binds more tightly than addition.

This solution is totally outside the world of the context-free grammar. It is, in some sense, a hack.

A better solution is to rewrite the grammar (or sometimes redesign te language) to eliminate the ambiguity.

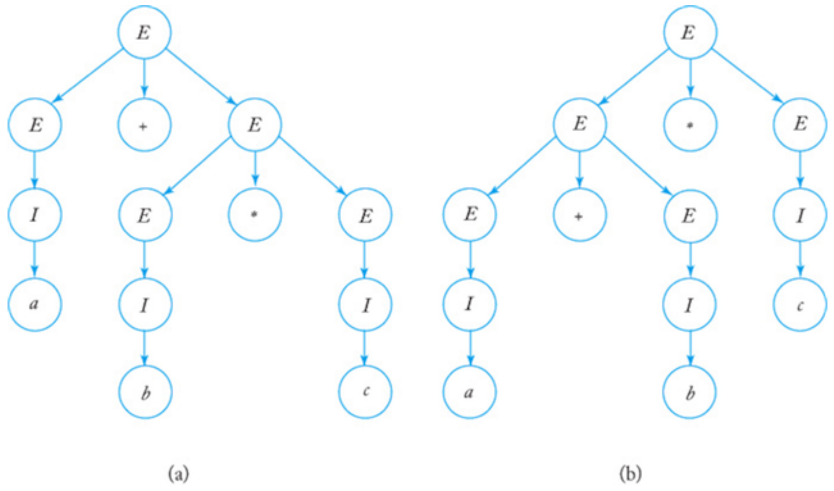


Figure 44: **Linz Fig. 5.5: Two Derivation Trees for** $a + b * c$

4.2.13 Linz Example 5.12

To rewrite the grammar in Linz Example 5.11, we introduce new variables, making V the set $\{E, T, F, I\}$, and replacing the productions with the following:

$E \rightarrow T$
$T \rightarrow F$
$F \rightarrow I$
$E \rightarrow E + T$
$T \rightarrow T * F$
$F \rightarrow (E)$
$I \rightarrow a \mid b \mid c$

Linz Figure 5.6 shows the only derivation tree for string $a + b * c$ in this revised grammar for arithmetic expressions.

4.2.14 Inherently Ambiguous

Linz Definition 5.6: If L is a context-free language for which there exists an unambiguous grammar, then L is said to be unambiguous. If every grammar that generates L is ambiguous, then language is called *inherently ambiguous*.

It is difficult to demonstrate that a grammar is inherently ambiguous. Often the best we can do is to give examples and argue informally that all grammars must be ambiguous.

4.2.15 Linz Example 5.13

The language

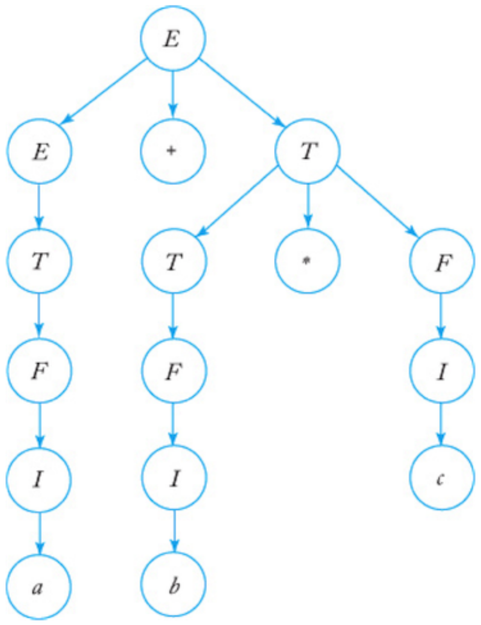


Figure 45: **Linz Fig. 5.6: Derivation Tree for $a{+}b{*}c$ in Revised Grammar**

$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$,

with n and m non-negative, is an inherently ambiguous context-free language.

Note that $L = L_1 \cup L_2$.

We can generate L_1 with the context-free grammar:

$S_1 = S_1c \mid A$
$A \rightarrow aAb \mid \lambda$

Similarly, we can generate L_2 with the context-free grammar:

$S_2 = aS_2 \mid B$
$B \rightarrow bBc \mid \lambda$

We can thus construct the union of these two sublanguages by adding a new production:

$S \rightarrow S_1 \mid S_2$

Thus this is a context-free language.

But consider a string of the form $a^n b^n c^n$ (i.e., $n = m$). It has two derivations, one starting with

$S \Rightarrow S_1$

and another starting with

$S \Rightarrow S_2$.

Thus the grammar is ambiguous.

L_1 and L_2 have conflicting requirements. L_1 places restrictions on the number of a's and b's while L_2 places restrictions on the number of b's and c's. It is imposible to find production rules that satisfy the $n = m$ case uniquely.

4.3 Context-Free Grammars and Programming Languages

The syntax for practical programming language syntax is usually expressed with context-free grammars. Compilers and interpreters must parse programs in these language to execute them.

The grammar for programming languages is often expressed using the *Backus-Naur Form (BNF)* to express productions.

For example, the language for arithmetic expressing in Linz Example 5.12 can be written in BNF as:

```
<expression> ::= <term>   | <expression> + <term>
      <term> ::= <factor> | <term> * <factor>
```

The items in angle brackets are variables, the symbols such as "+" and "-" are terminals, the "|" denotes alternatives, and "::=" separates the left and right sides of the productions.

Programming languages often use restricted grammars to get linear parsing: e.g., regular grammars, s-grammars, LL grammars, and LR grammars.

The aspects of programming languages that can be modeled by context-free grammars are called the the *syntax*.

Aspects such as type-checking are not context-free. Such issues are sometimes considered (incorrectly in your instructor's view) as part of the *semantics* of the language.

These are really still syntax, but they must be expressed in ways that are not context free.

5 OMIT Chapter 6

6 Pushdown Automata

Finite automata cannot recognize all context-free languages.

To recognize language $\{a^n b^n : n \geq 0\}$, an automaton must do more than just verify that all a's precede all b's; it must count an unbounded number of symbols. This cannot be done with the finite memory of a dfa.

To recognize language $\{ww^R : w \in \Sigma^*\}$, an automaton must do more than just count; it must remember a sequence of symbols and compare it to another sequence in reverse order.

Thus, to recognize context-free languages, an automaton needs unbounded memory. The second example above suggests using a stack as the "memory".

Hence, the class of machines we examine in this chapter are called *pushdown automata*.

In this chapter, we examine the connection between context-free languages and pushdown automata.

Unlike the situation with finite automata, deterministic and nondeterministic pushdown automata differ in the languages they accept.

- *Nondeterministic pushdown automata (npda)* accept precisely the class of context-free languages.
- *Deterministic pushdown automata (dpda)* just accept a subset–the *deterministic context-free languages*.

6.1 Nondeterministic Pushdown Automata

6.1.1 Schematic Drawing

Linz Figure 7.1 illustrates a pushdown automaton.

On each move, the control unit reads a symbol from the input file. Based on the input symbol and symbol on the top of the stack, the control unit changes the content of the stack and enters a new state.

6.1.2 Definition of a Pushdown Automaton

Linz Definition 7.1 (Nondeterministic Pushdown Accepter): A *nondeterministic pushdown accepter (npda)* is defined by the tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F),$$

where

Q is a finite set of internal states of the control unit,
Σ is the input alphabet,
Γ is a finite set of symbols called the *stack alphabet*,
$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$ is the transition

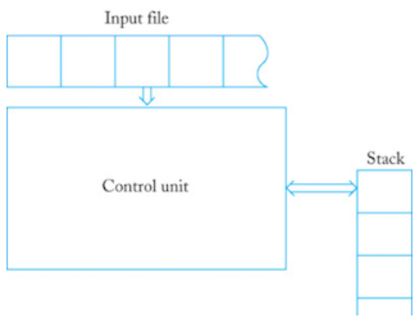


Figure 46: **Linz Fig. 7.1: Pushdown Automaton**

function,
$q_0 \in Q$ is the initial state of the control unit,
$z \in \Gamma$ is the *stack start symbol*,
$F \subseteq Q$ is the set of final states.

Note that the input and stack alphabets may differ and that start stack symbol z must be an element of Γ.

Consider the transition function $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$.

- 1st argument from Q is the current state.
- 2nd argument from $\Sigma \cup \{\lambda\})$ is either the next input symbol or λ for a move that does not consume input.
- 3rd argument from Γ is the current symbol at top of stack. (The stack cannot be empty! The stack start symbol represents the empty stack.)
- The result value is a finite set of pairs (q, w) where
  - q is the new state,
  - w is the (possibly empty) string that replaces the top symbol on the stack. (The first element of w will be the new top of the stack, second element under that, etc.)

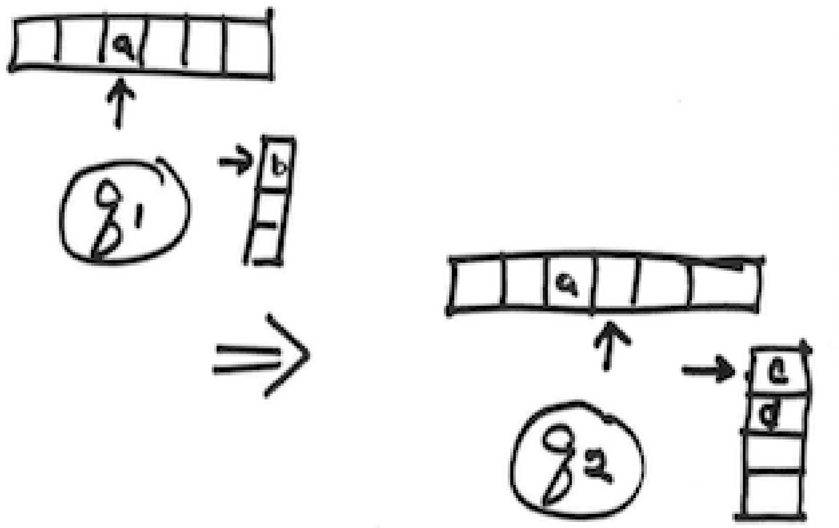
The machine is nondeterministic, but there are only a finite number of possible results for each step.

6.1.3 Linz Example 7.1

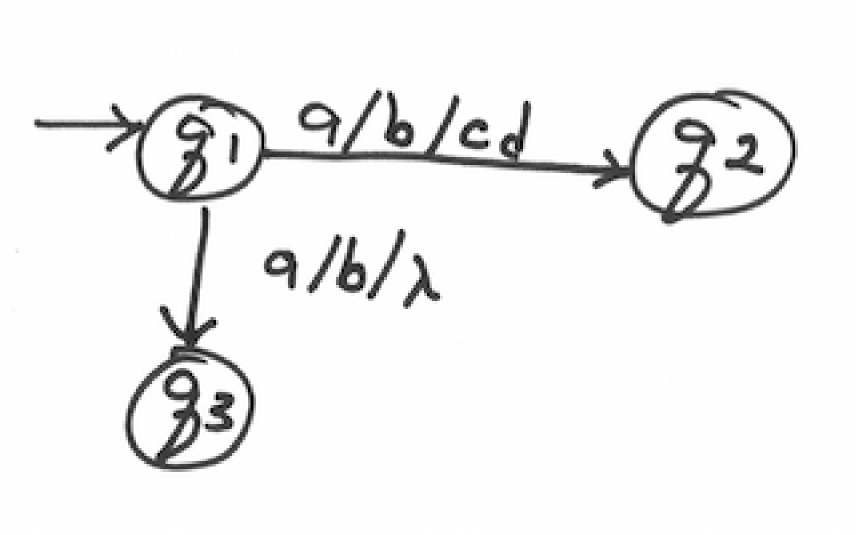
Suppose the set of transition rules of an npda contains

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}.$$

A possible change in the state of the automaton from q_1 to q_2 is shown in following diagram.



This transition function can be drawn as a transition graph as shown below. The triple marking the edges represent the input symbol, the symbol at the top of the stack, and the string pushed back on the stack. Here we use "/" to separate the elements of the triple on the edges; the book uses commas for this purpose.



6.1.4 Linz Example 7.2

Consider an npda $(Q, \Sigma, \Gamma, \delta, q_0, z, F)$ where

$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$

$\Gamma = \{0, 1\}$
$z = 0$
$F = \{q_3\}$

with the initial state q_0 and the transition function defined as follows:

1. $\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}$
2. $\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}$
3. $\delta(q_1, a, 1) = \{(q_1, 11)\}$
4. $\delta(q_1, b, 1) = \{(q_2, \lambda)\}$
5. $\delta(q_2, b, 1) = \{(q_2, \lambda)\}$
6. $\delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}$.

There are no transitions defined for final state q_3 or in the cases

$(q_0, b, 0)$, $(q_2, a, 1)$, $(q_0, a, 1)$, $(q_0, b, 1)$, $(q_1, a, 0)$, and $(q_1, b, 0)$.

These are *dead configurations* of the npda. In these cases, δ maps the arguments to the empty set.

The machine executes as follows.

1. The first transition rule is nondeterministic, with two choices for input symbol a and stack top 0.
   (a) The machine can push 1 on the stack and transition to state q_1. (This is the only choice that will allow the machine to accept additional input.)
   (b) The machine can pop the start symbol 0 and transition to final state q_3. (This is the only choice that will allow the machine to accept a single a.)
2. For stack top z, the machine can also transition from the initial state q_0 to final state q_3 without consuming any input. (This is only choice that will allow the machine to accept an empty string.) Note that rule 2 overlaps with rule 1, giving additional nondeterminism.
3. While the machine reads a's, it pushes a 1 on the stack and stays in state q_1.
4. When the machine reads a b (with stack top 1), it pops the 1 from the stack and transitions to state q_2.
5. While the machine reads b's (with stack top 1), it pops the 1 from the stack and stays in state q_2.
6. When the machine encounters the stack top 0, it pops the stack and transitions to final state q_3.

Acceptance or rejection?

- If the machine reaches final state q_3 with *no unprocessed input* using any possible sequence of transitions, then the machine *accepts* the input string.
- If every sequence of possible transitions reaches a configuration in which no move is defined or reaches the final state with unprocessed input remaining, then the machine *rejects* the input string.

The machine accepts:

- λ (via rule 2)
- singleton string a (via rule 1b)
- any string in which there are some number of a's followed by the same number of b's (via rules 1a-3-4-5-6 as applicable)

Other strings will always end in dead configurations. For example:

- b gets stuck in q_3 with unprocessed input (via rule 2)
- aa gets stuck in q_1 (via rules 1a-3) or in q_3 with unprocessed input (via rule 1b or rule 2)
- aab gets stuck in q_2 with stack top 1 (via rules 1a-3-4) or in q_3 with unprocessed input (via rule 1b or 2)
- abb gets stuck in q_3 with unprocessed input (via rule 1b or rule 2 or rules 1a-4-5-6)
- aba gets stuck in q_2 (via rules 1a-4) or in q_3 with unprocessed input (via rule 1b or rule 2 or rules 1a-4-6)

Thus, it is not difficult to see that $L = \{a^n b^n : n \geq 0\} \cup \{a\}$.

Linz Figure 7.2 shows a transition graph for this npda. The triples marking the edges represent the input symbol, the symbol at the top of the stack, and the string pushed back on the stack.

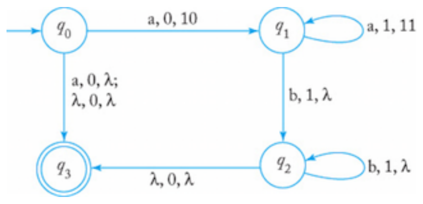


Figure 47: **Linz Fig. 7.2: Transition Graph for Example 7.2**

6.1.5 Instantaneous Descriptions of Pushdown Automata

Transition graphs are useful for describing pushdown automata, but they are not useful in formal reasoning about them. For that, we use a concise notation for describing configurations of pushdown automata with tuples.

The triple (q, w, u) where

q is the control unit state
w is the unread input string
u is the stack as string, beginning with the top symbol

is called an *instantaneous description* of a pushdown automaton.

We introduce the symbol $\vdash$ to denote a *move* from one instantaneous description to another such that

$$(q_1, aw, bx) \vdash (q_2, w, yx)$$

is possible if and only if

$$(q_2, y) \in \delta(q_1, a, b).$$

We also introduce the notation $\vdash^*$ to denote an arbitrary number of steps of the machine.

6.1.6 Language Accepted by an NPDA

Linz Definition 7.2 (Language Accepted by a Pushdown Automaton): Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ be a nondeterministic pushdown automaton. The language accepted by M is the set

$$L(M) = \{w \in \Sigma^* : (q_0, w, z) \vdash_M^* (p, \lambda, u), p \in F, u \in \Gamma^*\}.$$

In words, the language accepted by M is the set of all strings that can put M into a final state at the end of the string. The stack content u is irrelevant to this definition of acceptance.

6.1.7 Linz Example 7.4

Construct an npda for the language

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

We must count a's and b's, but their relative order is not important.

The basic idea is:

- on "a", push 0
- on "b", pop 0

But, what if $n_b > n_a$ at some point?

We must allow a "negative" count. So we modify the above solution as follows:

- on "a",

```
if top is $z$ or 0
    push 0
else if top is 1
    pop 1
```

- on "b",

```
if top is $z$ or 1
    push 1
else if top is 0
    pop 0
```

So the solution is an npda.

$M = (\{q_0, q_f\}, \{a, b\}, \{0, 1, z\}, \delta, q_0, z, \{q_f\})$, with δ given as

1. $\delta(q_0, \lambda, z) = \{(q_f, z)\}$
2. $\delta(q_0, a, z) = \{(q_0, 0z)\}$
3. $\delta(q_0, b, z) = \{(q_0, 1z)\}$
4. $\delta(q_0, a, 0) = \{(q_0, 00)\}$
5. $\delta(q_0, b, 0) = \{(q_0, \lambda)\}$
6. $\delta(q_0, a, 1) = \{(q_0, \lambda)\}$
7. $\delta(q_0, b, 1) = \{(q_0, 11)\}$.

Linz Figure 7.3 shows a transition graph for this npda.

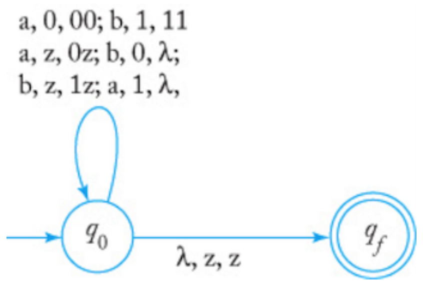


Figure 48: **Linz Fig. 7.3: Transition Graph for Example 7.4**

In processing the string *baab*, the npda makes the following moves (as indicated by transition rule number):

		$(q_0, baab, z)$
(3)	$\vdash$	$(q_0, aab, 1z)$
(6)	$\vdash$	(q_0, ab, z)
(2)	$\vdash$	$(q_0, b, 0z)$
(5)	$\vdash$	(q_0, λ, z)
(1)	$\vdash$	(q_f, λ, z)

Hence, the string is accepted.

6.1.8 Linz Example 7.5

Construct an npda for accepting the language $L = \{ww^R : w \in \{a, b\}^+\}$,

The basic idea is:

- push symbols from w on stack from left to right
- pop symbols from stack for w^R (which is w right-to-left)

Problem: How do we find the middle?

Solution: Use nondeterminism!

- Each symbol could be at middle.

- Automaton "guesses" when to switch.

For $L = \{ww^R : w \in \{a,b\}^+\}$, a solution to the problem is given by $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$, where:

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, b, z\}$ – which is Σ plus the staack start symbol $F = \{q_2\}$

The transition function can be visualized as having several parts.

- a set of transitions to push w on the stack (one for each element of $\Sigma \times \Gamma$):

  1. $\delta(q_0, a, a) = \{(q_0, aa)\}$
  2. $\delta(q_0, b, a) = \{(q_0, ba)\}$
  3. $\delta(q_0, a, b) = \{(q_0, ab)\}$
  4. $\delta(q_0, b, b) = \{(q_0, bb)\}$
  5. $\delta(q_0, a, z) = \{(q_0, az)\}$
  6. $\delta(q_0, b, z) = \{(q_0, bz)\}$

- a set of transitions to guess the middle of the string, where the npda switches from state q_0 to q_1 (any position is potentially the middle):

  7. $\delta(q_0, \lambda, a) = \{(q_1, a)\}$
  8. $\delta(q_0, \lambda, b) = \{(q_1, b)\}$

- a set of transitions to match w^R against the contents of the stack:

  9. $\delta(q_1, a, a) = \{(q_1, \lambda)\}$
  10. $\delta(q_1, b, b) = \{(q_1, \lambda)\}$

- a transition to recognize a successful match:

11. $\delta(q_1, \lambda, z) = \{(q_2, z)\}$

Remember that, to be accepted, a final state must be reached with no unprocessed input remaining.

The sequence of moves accepting *abba* is as follows, where the number in parenthesis gives the transition rule applied:

		$(q_0, abba, z)$
(5)	$\vdash$	(q_0, bba, az)
(2)	$\vdash$	(q_0, ba, baz)
(8)	$\vdash$	(q_1, ba, baz)
(10)	$\vdash$	(q_1, a, az)
(9)	$\vdash$	(q_1, λ, z)
(11)	$\vdash$	(q_2, z)

6.2 Pushdown Automata and Context-Free Languages

6.2.1 Pushdown Automata for CFGs

Underlying idea: Given a context-free language, construct an npda that simulates a leftmost derivation for any string in the language.

We assume the context-free language is represented as grammar in *Greibach Normal Form*, as defined in Linz Chapter 6. We did not cover that chapter, but the definition and key theorem are shown below.

Greibach Normal Form restricts the positions at which terminals and variables can appear in a grammar's productions.

Linz Definition 6.5 (Greibach Normal Form): A context-free grammar is said to be in *Greibach Normal Form* if all productions have the form

$$A \rightarrow ax,$$

where $a \in T$ and $x \in V^*$.

The structure of a grammar in Greibach Normal Form is similar to that of an s-grammar except that, unlike s-grammars, the grammar does not restrict pairs (A, a) to single occurrences within the set of productions.

Linz Theorem 6.7 (Existence of Greibach Normal Form Grammars): For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar $\hat{G}$ in Greibach normal form.

Underlying idea, continued: Consider a sentential form, for example,

$$x_1x_2x_3x_4x_5x_6$$

where $x_1x_2x_3$ are the *terminals* read from the input and $x_4x_5x_6$ are the variables on the stack.

Consider a production $A \rightarrow ax$.

If variable A is on top of stack and terminal a is the next input symbol, then remove A from the stack and push back x.

An npda transition function δ for $A \rightarrow ax$ must be defined to have the move

$$(q, aw, Ay) \vdash (q, w, xy)$$

for some state q, input string suffix w, and stack y. This, we define δ such that

$$\delta(q, a, A) = \{(q, x)\}.$$

6.2.2 Linz Example 7.6

Construct a pda to accept the language generated by grammar with productions

$$S \rightarrow aSbb \mid a.$$

First, we transform this grammar into Greibach Normal Form:

$S \to aSA \mid a$
$A \to bB$
$B \to b$

We define the pda to have three states – an initial state q_0, a final state q_2, and an intermediate state q_1.

We define the initial transition rule to push the start symbol S on the stack:

$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$

We simulate the production $S \to aSA$ with a transition that reads a from the input and replaces S on the top of the stack by SA.

Similarly, we simulate the production $S \to a$ with a transition that reads a while simply removing S from the top of the stack. We represent these two productions in the pda as the nondeterministic transition rule:

$\delta(q_1, a, S) = \{(q_1, SA), (q_1, \lambda)\}$

Doing the same for the other productions, we get transition rules:

$\delta(q_1, b, A) = \{(q_1, B)\}$
$\delta(q_1, b, B) = \{(q_1, \lambda)\}$

When the stack start symbol appears at the top of the stack, the derivation is complete. We define a transition rule to move the pda into its final state:

$\delta(q_1, \lambda, z) = \{(q_2, \lambda)\}$

6.2.3 Constructing an NPDA for a CFG

Linz Theorem 7.1 (Existence of NPDA for Context-Free Language): For any context-free language L, there exists an npda M such that $L = L(M)$.

Proof: The proof partly follows from the following construction (algorithm).

Algorithm to construct an npda for a context-free grammar

- Let $G = (V, T, S, P)$ be a grammar for L in Greibach Normal Form.
- Construct npda $M = (\{q_0, q_1, q_f\}, T, V \cup \{z\}, \delta, q_0, z, \{q_f\})$ where:
  - $z \notin V$
  - T is the input alphabet for the npda
  - $V \cup \{z\}$ is the stack alphabet for the npda
- Define transition rule $\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$ to initialize the stack.
- For every $A \to au$ in P, define transition rules

  $(q_1, u) \in \delta(q_1, a, A)$

  that read a, pop A, and push u. (Note the possible nondeterminism.)
- Define transition rule $\delta(q_1, \lambda, z) = \{(q_f, z)\}$ to detect the end of processing.

6.2.4 Linz Example 7.7

Consider the grammar:

$S \to aA$
$A \to aABC \mid bB \mid a$
$B \to b$
$C \to c$

This grammar is already in Greibach Normal Form. So we can apply the algorithm above directly.

In addition to the transition rules for the startup and shutdown, i.e.,

1. $\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$
2. $\delta(q_1, \lambda, z) = \{(q_f, z)\}$

the npda has the following transition rules for the productions:

3. $\delta(q_1, a, S) = \{(q_1, A)\}$
4. $\delta(q_1, a, A) = \{(q_1, ABC), (q_1, \lambda)\}$
5. $\delta(q_1, b, A) = \{(q_1, B)\}$
6. $\delta(q_1, b, B) = \{(q_1, \lambda)\}$
7. $\delta(q_1, c, C) = \{(q_1, \lambda)\}$

The sequence of moves made by M in processing is $aaabc$ is

		$(q_0, aaabc, z)$
(1)	$\vdash$	$(q_1, aaabc, Sz)$
(3)	$\vdash$	$(q_1, aabc, Az)$
(4a)	$\vdash$	$(q_1, abc, ABCz)$
(4b)	$\vdash$	(q_1, bc, BCz)
(6)	$\vdash$	(q_1, c, Cz)
(7)	$\vdash$	(q_1, λ, z)
(2)	$\vdash$	(q_f, λ, z)

This corresponds to the derivation

$$S \Rightarrow aA \Rightarrow aaABC \Rightarrow aaaBC \Rightarrow aaabC \Rightarrow aaabc.$$

The previous construction assumed Greibach Normal Form. This is not necessary, but the needed construction technique is more complex, as sketched below.

$A \to Bx$

$(q_1, Bx) \in \delta(q_1, \lambda, A)$

$A \to abCx$

e.g.,
$(q_2, \lambda) \in \delta(q_1, a, a)$

$(q_3, \lambda) \in \delta(q_2, b, b)$
$(q_1, Cx) \in \delta(q_3, \lambda, A)$

etc.

6.2.5 Constructing a CFG for an NPDA

Linz Theorem 7.2 (Existence of a Context-Free Language for an NPDA): If $L = L(M)$ for some npda M, then L is a context-free language.

Basic idea: To construct a context-free grammar from an npda, reverse the previous construction.

That is, construct a grammar to simulate npda moves:

- The stack content becomes the variable part of the grammar.
- The processed input becomes the terminal prefix of sentential form.

This leads to a relatively complicated construction. This is described in the Linz textbook in more detail, but we will not cover it in this course.

6.3 Deterministic Pushdown Automata and Deterministic Context-Free Languages

6.3.1 Deterministic Pushdown Automata

A *deterministic pushdown accepter (dpda)* is a pushdown automaton that never has a choice in its move.

Linz Definition 7.3 (Deterministic Pushdown Automaton): A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is *deterministic* if it is an automaton as defined in Linz Definition 7.1, subject to the restrictions that, for every $q \in Q, a \in \Sigma \cup \{\lambda\}$, and $b \in \Gamma$,

1. $\delta(q, a, b)$ contains at most one element,
2. if $\delta(q, \lambda, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$.

Restriction 1 requires that for, any given input symbol and any stack top, at most one move can be made.

Restriction 2 requires that, when a λ-move is possible for some configuration, no input-consuming alternative is available.

Consider the difference between this dpda definition and the dfa definition:

- A dpda allows λ-moves, but the moves are deterministic.
- A dpda may have dead configurations.

Linz Definition 7.4 (Deterministic Context-Free Language): A language L is a *deterministic context-free language* if and only if there exists a dpda M such that $L = L(M)$.

6.3.2 Linz Example 7.10

The language $L = \{a^n b^n : n \geq 0\}$ is a deterministic context-free language.

The pda $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{0, 1\}, \delta, q_0, 0, \{q_0\})$ with transition rules

$\delta(q_0, a, 0) = \{(q_1, 10)\}$
$\delta(q_1, a, 1) = \{(q_1, 11)\}$
$\delta(q_1, b, 1) = \{(q_2, \lambda)\}$
$\delta(q_2, b, 1) = \{(q_2, \lambda)\}$
$\delta(q_2, \lambda, 0) = \{(q_0, \lambda)\}$

accepts the given language. This grammar satisfies the conditions of Linz Definition 7.4. Therefore, it is deterministic.

6.3.3 Linz Example 7.5 Revisited

Consider language

$L = \{ww^R : w \in \{a, b\}^+\}$

and machine

$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$

where:

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, b, z\}$
$F = \{q_2\}$

The transition function can be visualized as having several parts:

- a set of transition rules to push w on the stack

  $\delta(q_0, a, a) = \{(q_0, aa)\} \leftarrow$ **Restriction 2 violation**
  $\delta(q_0, b, a) = \{(q_0, ba)\}$
  $\delta(q_0, a, b) = \{(q_0, ab)\}$
  $\delta(q_0, b, b) = \{(q_0, bb)\}$
  $\delta(q_0, a, z) = \{(q_0, az)\}$
  $\delta(q_0, b, z) = \{(q_0, bz)\}$

- a set of transition rules to guess the middle of the string, where the npda switches from state q_0 to q_1

  $\delta(q_0, \lambda, a) = \{(q_1, a)\} \leftarrow$ **Restriction 2 violation**
  $\delta(q_0, \lambda, b) = \{(q_1, b)\}$

- a set of transition rules to match w^R against the contents of the stack

  $\delta(q_1, a, a) = \{(q_1, \lambda)\}$
  $\delta(q_1, b, b) = \{(q_1, \lambda)\}$

- a transition rule to recognize a successful match

  $\delta(q_1, \lambda, z) = \{(q_2, z)\}$

This machines violates *Restriciton 2* of Linz Definition 7.3 (Deterministic Pushdown Automaton) as indicated above. Thus, it is not deterministic.

Moreover, L is itself not deterministic (which is not proven here).

6.4 Grammars for Deterministic Context-Free Grammars

Deterministic context-free languages are important because they can be parsed efficiently.

- The dpda essentially defines a parsing machine.
- Because it is deterministic, there is no backtracking involved.
- We can thus easily write a reasonably efficient computer program to implement the parser.
- Thus deterministic context-free languages are important in the theory and design of compilers for programming languages.

An *LL-grammar* is a generalization of the concept of s-grammar. This family of grammars generates the deterministic context-free languages.

Compilers for practical programming languages may use top-down parsers based on LL-grammars to parse the languages efficiently.

7 Properties of Context-Free Languages

Chapter 4 examines the closure properties of the family of regular languages, algorithms for determining various properties of regular languages, and methods for proving languages are not regular (e.g., the Pumping Lemma).

Chapter 8 examines similar aspects of the family of context-free languages.

7.1 Two Pumping Lemmas

Because of insufficient time and extensive coverage of the Pumping Lemma for regular languages, we will not cover the Pumping Lemmas for Context-Free Languages in this course. See section 8.1 of the Linz textbook if you are interested in this topic.

7.1.1 Context-Free Languages

Linz Section 8.1 includes the following language examples. The results of these are used in the remainder of this chapter.

1. Linz Example 8.1 shows $L = \{a^n b^n c^n : n \geq 0\}$ is not context free.
2. Linz Example 8.2 shows $L = \{ww : w \in \{a, b\}^*\}$ is not context free.
3. Linz Example 8.3 shows $L = \{a^{n!} : n \geq 0\}$ is not context free.
4. Linz Example 8.4 shows $L = \{a^n b^j : n = j^2\}$ is not context free.

7.1.2 Linear Languages

Linz Section 8.1 includes the following definitions. (The definition of linear grammar is actually from Chapter 3.)

Definition (Linear Grammar): A *linear grammar* is a grammar in which at most one variable can appear on the right side of any production.

A *linear context-free grammar* is thus a context-free grammar that is also a linear grammar.

Linz Definition 8.5 (Linear Language): A context-free language L is *linear* if there exists a linear context-free grammar G such that $L = L(G)$.

Linz Section 8.1 also includes the following language examples.

5. Linz Example 8.5 shows $L = \{a^n b^n : n \geq 0\}$ is a linear language.
6. Linz Example 8.6 shows $L = \{w : n_a(w) = n_b(w)\}$ is not linear.

7.2 Closure Properties and Decision Algorithms for Context-Free Languages

In most cases, the proofs and algorithms for the properties of regular languages rely upon manipulation of transition graphs for finite automata. Hence, they are relatively straightforward.

When we consider similar properties for context-free languages, we encounter more difficulties.

- Some properties do not hold.
- Other properties require more complex arguments.
- Some intuitively simple questions cannot be answered.

Let's consider closure under the simple set operations as we did for regular languages in Linz Theorem 4.1.

7.2.1 Closure under Union, Concatenation, and Star-Closure

Linz Theorem 8.3 (Closure under Union, Concatenation, and Star-Closure): The family of context-free languages is *closed under* (a) *union,* (b) *concatenation, and* (c) *star-closure.*

(8.3a) Proof of Closure under Union:

Let L_1 and L_2 be context-free languages with the corresponding context-free grammars $G_1 = (V_1, T_1, S_1, P_1)$ and $G_2 = (V_2, T_2, S_2, P_2)$.

Assume V_1 and V_2 are disjoint. (If not, we can make them so by renaming.)

Consider $L(G_3)$ where

$$G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, S_3, P_3)$$

with:

$S_3 \notin V_1 \cup V_2$ – i.e, S_3 is a fresh variable
$P_3 = P_1 \cup P_2 \cup \{\ S_3 \rightarrow S_1 \ |\ \ S_2\ \}$

Clearly, G_3 is a context-free grammar. So $L(G_3)$ is a context-free language.

Now, we need to show that $L(G_3) = L_1 \cup L_2$.

For $w \in L_1$, there is a derivation in G_3:

(1) $S_3 \Rightarrow S_1 \overset{*}{\Rightarrow} w$

Similarly, for $w \in L_2$, there is a derivation in G_3:

(2) $S_3 \Rightarrow S_2 \overset{*}{\Rightarrow} w$

Also, for $w \in L(G_3)$, the first step of the derivation must be either (1) $S_3 \Rightarrow S_1$ or (2) $S_3 \Rightarrow S_2$.

For choice 1, the sentential forms derived from S_1 only have variables from V_1. But V_1 is disjoint from V_2. Thus the derivation

$$S_1 \overset{*}{\Rightarrow} w$$

can only involve productions from from P_1. Hence, for choice 1, $w \in L_1$.

Using a similar argument for choice 2, we conclude $w \in L_2$.

Therefore, $L(G_3) = L_1 \cup L_2$.

QED.

(8.3b) Proof of Closure under Concatenation:

Consider $L(G_4)$ where

$$G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, S_4, P_4)$$

with:

$$S_4 \notin V_1 \cup V_2$$
$$P_4 = P_1 \cup P_2 \cup \{\ S_4 \rightarrow S_1 S_2\ \}$$

Then $L(G_4) = L_1 L_2$ follows from a similar argument to the one in part (a).

QED.

(8.3c) Proof of Closure under Star-Closure:

Consider $L(G_5)$ where

$$G_5 = (V_1 \cup \{S_5\}, T_1, S_5, P_5)$$

with:

$$S_5 \notin V_1$$
$$P_5 = P_1 \cup \{\ S_5 \rightarrow S_1 S_5 \mid \lambda\ \}$$

Then $L(G_5) = L_1^*$ follows from a similar argument to the one in part (a).

QED.

7.2.2 Non-Closure under Intersection and Complementation

Linz Theorem 8.4 (Non-closure under Intersection and Complementation): The family of context-free languages is *not closed under* (a) *intersection and* (b) *complementation.*

(8.4b) Proof of Non-closure under Intersection:

Assume the family of context-free languages is closed under intersection. Show that this leads to a contradiction.

It is sufficient to find two context-free languages whose intersection is not context-free.

Consider languages L_1 and L_2 defined as follows:

$$L_1 = \{a^n b^n c^m : n \geq 0, m \geq 0\}$$
$$L_2 = \{a^n b^m c^m : n \geq 0, m \geq 0\}$$

One way to show that a language is context-free is to find a context-free grammar that generates it. The following context-free grammar generates L_1:

$S \rightarrow S_1 S_2$
$S_1 \rightarrow a S_1 b \mid \lambda$
$S_2 \rightarrow c S_2 \mid \lambda$

Alternatively, we could observe that L_1 is the concatenation of two context-free languages and, hence, context-free by Linz Theorem 8.3 above.

Similarly, we can show that L_2 is context free.

From the assumption, we thus have that $L_1 \cap L_2$ is context free.

But

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\},$$

which is not context free. Linz proves this in Linz Example 8.1 (which is in the part of this chapter we did not cover in this course).

Thus we have a contradiction. Therefore, the family of context-free languages is not closed under intersection.

QED.

(8.4b) Proof of Non-closure under Complementation:

Assume the family of context-free languages is closed under complementation. Show that this leads to a contradiction.

Consider arbitrary context-free languages L_1 and L_2.

From set theory, we know that

$$L_1 \cap L_2 = \overline{\bar{L_1} \cup \bar{L_2}}.$$

From Linz Theorem 8.3 and the assumption that context-free languages are closed under complementation, we deduce that the right side ($\overline{\bar{L_1} \cup \bar{L_2}}$) is a context-free language for all L_1 and L_2.

However, we know from part (a) that the left side ($L_1 \cap L_2$) is not necessarily a context-free language for all L_1 and L_2.

Thus we have a contradiction. Therefore, the family of context-free languages is not closed under complementation.

QED.

7.2.3 Closure under Regular Intersection

Although context-free languages are not, in general, closed under intersection, there is a useful special case that is closed.

Linz Theorem 8.5 (Closure Under Regular Intersection): Let L_1 be a context-free language and L_2 be a regular language. Then $L_1 \cap L_2$ is *context free.*

Proof:

Let $M_1 = (Q, \Sigma, \Gamma, \delta_1, q_0, z, F_1)$ be an npda that accepts context-free language L_1.

Let $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$ be a dfa that accepts regular language L_2.

We construct an npda

$$\widehat{M} = (\widehat{Q}, \Sigma, \Gamma, \widehat{\delta}, \widehat{q_0}, \widehat{F})$$

that simulates M_1 and M_2 operating simultaneously (i.e., executes the moves of both machines for each input symbol).

We choose pairs of states from M_1 and M_2 to represent the states of $\widehat{M}$ as follows:

$$\widehat{Q} = Q \times P$$
$$\widehat{q_0} = (q_0, p_0)$$
$$\widehat{F} = F_1 \times F_2$$

We specify $\widehat{\delta}$ such that the moves of $\widehat{M}$ correspond to simultaneous moves of M_1 and M_2. That is,

$$((q_k, p_l), x) \in \widehat{\delta}((q_i, p_j), a, b)$$

if and only if

$$(q_k, x) \in \delta_1(q_i, a, b)$$

and

$$\delta_2(p_j, a) = p_l.$$

For moves (q_i, λ, b) in δ_1, we extend δ_2 so that $\delta_2(p_l, \lambda) = p_l$ for all l.

By induction on the length of the derivations, we can prove that

$$((q_0, p_0), w, z) \vdash^*_{\widehat{M}} ((q_r, p_s), \lambda, x),$$

with $q_r \in F_1$ and $p_s \in F_2$ if and only if

$$(q_0, w, z) \vdash^*_{M_1} (q_r, \lambda, x)$$

and

$$\delta^*(p_0, w) = p_s.$$

Therefore, a string is accepted by $\widehat{M}$ if and only if it is accepted by both M_1 and M_2. That is, the string is in $L(M_1) \cap L(M_2) = L_1 \cap L_2$.

QED.

7.2.4 Linz Example 8.7

Show that the language

$$L = \{a^n b^n : n \geq 0, n \neq 100\}$$

is context free.

We can construct an npda or context-free grammar for L, but this is tedious. Instead, we use closure of regular intersection (Linz Theorem 8.5).

Let $L_1 = \{a^{100}b^{100}\}$.

L_1 is finite, and thus also regular. Hence, $\bar{L_1}$ is regular because regular languages are closed under complementation.

From previous results, we know that $L = \{a^n b^n : n \geq 0\}$ is context free.

Clearly, $L = \{a^n b^n : n \geq 0\} \cap \bar{L_1}$.

By the closure of context-free languages under regular intersection, L is a context-free language.

7.2.5 Linz Example 8.8

Show that

$$L = \{w \in \{a, b, c\}^* : n_a(w) = n_b(w) = n_c(w)\}$$

is not context free.

Although we could use the Pumping Lemma for Context-Free Languages, we again use closure of regular intersection (Linz Theorem 8.5).

Assume that L is context free. Show that this leads to a contradiction.

Thus

$$L \cap L(a^* b^* c^*) = \{a^n b^n c^n : n \geq 0\}$$

is also context free. But we have previously proved that this language is not context free.

Thus we have a contradiction. Therefore, L is not context free.

7.2.6 Some Decidable Properties of Context Free Languages

There exist algorithms for determine whether a context-free language is empty or nonempty and finite or infinite.

These algorithms process the context-free grammars for the languages. They assume that the grammars are first transformed using various algorithms from Linz Chapter 6 (which we did not cover in this course).

The algorithms from Chapter 6 include the removal of

- *useless symbols and productions* (i.e., variables and productions that can never generate a sentence)
- *λ-productions* (i.e., productions with λ on the right side)
- *unit productions* (i.e., productions of the form $A \rightarrow B$)

Linz Theorem 8.6 (Determining Empty Context-Free Languages): Given a context-free grammar $G = (V, T, S, P)$, then there exists an algorithm for *determining whether* or not $L(G)$ is *empty*.

Basic idea of algorithm: Assuming $\lambda \notin L$, remove the useless productions. If the start symbol is useless, then L is empty. Otherwise, L is nonempty.

Linz Theorem 8.7 (Determining Infinite Context-Free Languages): Given a context-free grammar $G = (V, T, S, P)$, then there exists an algorithm for *determining whether* or not $L(G)$ is *infinite*.

Basic idea of algorithm: Remove useless symbols, λ-productions, and unit productions. If there are variables A that repeat as in

$$A \overset{*}{\Rightarrow} xAy$$

then the language is infinite. Otherwise, the language is finite. To determine repeated variables, we can build a graph of the dependencies of the variables on each other. If this graph has a cycle, then the variable at the base of the cycle is repeated.

Unfortunately, *other simple properties are not as easy* as the above.

For example, *there is no algorithm to determine whether two context-free grammars generate the same language.*

8 Turing Machines

A finite accepter (nfa, dfa)

- has no local storage
- accepts a regular language

A pushdown accepter (npda, dpda)

- has a stack for local storage
- accepts a language from a larger family
    - an npda accepts a context-free language
    - a dpda accepts a deterministic context-free language

The family of regular languages is a subset of the deterministic context-free languages, which is a subset of the context-free languages.

But, as we saw in Chapter 8, not all languages of interest are context-free. To accept languages like $\{a^n b^n c^n : n \geq 0\}$ and $\{ww : w \in \{a, b\}^*\}$, we need an automaton with a more flexible internal storage mechanism.

What kind of internal storage is needed to allow the machine to accept languages such as these? multiple stacks? a queue? some other mechanism?

More ambitiously, what is the most powerful automaton we can define? What are the limits of mechanical computation?

This chapter introduces the *Turing machine* to explore these theoretical questions. The Turing machine is a fundamental concept in the theoretical study of computation.

The Turing machine

- has a *tape*, a one-dimensional array of readable and writable cells that is unbounded in both directions
- accepts a language from the family of *recursively enumerable languages*, a larger family of languages than context-free

Although Turing machines are simple mechanisms, the *Turing thesis* (also known as the Church-Turing thesis) maintains that *any computation that can be carried out on present-day computers an be done on a Turing machine.*

Note: Much of the work on computability was published in the 1930's, before the advent of electronic computers a decade later. It included work by Austrian (and later American) logician Kurt Goedel on primitive recursive function theory, American mathematician Alonso Church on lambda calculus (a foundation of functional programming), British mathematician Alan Turing (also later a PhD student of Church's) on Turing machines, and American mathematician Emil Post on Post machines.

8.1 The Standard Turing Machine

8.1.1 What is a Turing Machine?

8.1.1.1 Schematic Drawing of Turing Machine Linz Figure 9.1 shows a schematic drawing of a standard Turing machine.

This deviates from the general scheme given in Chapter 1 in that the input file, internal storage, and output mechanism are all represented by a single mechanism, the tape. The input is on the tape at initiation and the output is on that tape at termination.

On each move, the tape's *read-write head* reads a symbol from the current tape cell, writes a symbol back to that cell, and moves one cell to the left or right.

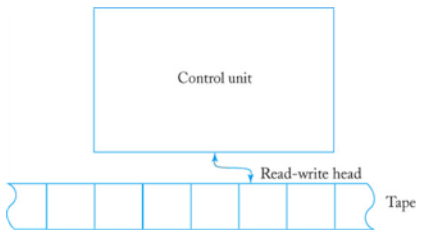


Figure 49: **Linz Fig. 9.1: Standard Turing Machine**

8.1.1.2 Definition of Turing Machine Turing machines were first defined by British mathematician Alan Turing in 1937, while he was a graduate student at Cambridge University.

Linz Definition 9.1 (Turing Machine): A *Turing machine* M is defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$$

where

1. Q is the set of internal states
2. Σ is the input alphabet
3. Γ is a finite set of symbols called the *tape alphabet*
4. δ is the transition function
5. $\Box \in \Gamma$ is a special symbol called the *blank*
6. $q_0 \in Q$ is the initial state
7. $F \subseteq Q$ is the set of final states

We also require

8. $\Sigma \subseteq \Gamma - \{\Box\}$

and define

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

Requirement 8 means that the blank symbol □ cannot be either an input or an output of a Turing machine. It is the default content for any cell that has no meaningful content.

From requirement 9, we see that the arguments of the transition function δ are:

- the current state of the control unit
- the current tape symbol

The result of the transition function δ gives:

- the new state of the control unit
- the symbol that replaces the current symbol on the tape
- a move symbol L or R, denoting a move of the read-write head to the *left* or the *right* on the tape

In general, δ is a partial function. That is, not all configurations have a next move defined.

8.1.1.3 Linz Example 9.1

Consider a Turing machine with a move defined as follows:

$$\delta(q_0, a) = (q_1, d, R)$$

Linz Figure 9.2 shows the situation (a) before the move and (b) after the move.

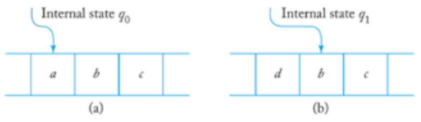


Figure 50: **Linz Fig. 9.2: One Move of a Turing Machine**

8.1.1.4 A Simple Computer

A Turing machine is a simple computer. It has

- a processing unit that has a finite memory
- a tape that provides unlimited secondary storage capacity
- a limited set of instructions

The Turing machine can

- sense the symbol under the tape's read-write head
- use the result to decide what to do next
- write a symbol back to the tape
- change the state of the control
- move the read-write head one position to the left or right on the tape

The transition function δ determines the behavior of the machine, i.e., it is the machine's *program*.

The Turing macine starts in initial state q_0 and then goes through a sequence of moves defined by δ. A cell on the tape may be read and written many times.

Eventually the Turing machine may enter a configuration for which δ is undefined. When it enters such a state, the machine *halts*. Hence, this state is called a *halt state*.

Typically, no transitions are defined on any final state.

8.1.1.5 Linz Example 9.2

Consider the Turing machine defined by

$$\begin{aligned}
&Q = \{q_0, q_1\},\\
&\Sigma = \{a, b\},\\
&\Gamma = \{a, b, \Box\},\\
&F = \{q_1\}
\end{aligned}$$

where δ is defined as follows:

1. $\delta(q_0, a) = (q_0, b, R)$,

2. $\delta(q_0, b) = (q_0, b, R)$,

3. $\delta(q_0, \Box) = (q_1, \Box, L)$.

Linz Figure 9 .3 shows a sequence of moves for this Turing machine:

- It begins in state q_0 with the input positioned over an a.
- When an a is read, transition rule 1 fires, replaces a by b on the tape, moves right, and stays in state q_0.
- When a b is read, transition rule 2 fires, leaves b on the tape, moves right, and stays in state q_0.
- It continues moving right, replacing each a by a b and leaving each b unchanged.
- When a blank ($\Box$) is read, transition rule 3 fires, leaves the blank on the tape, moves left, and enters final state q_1.

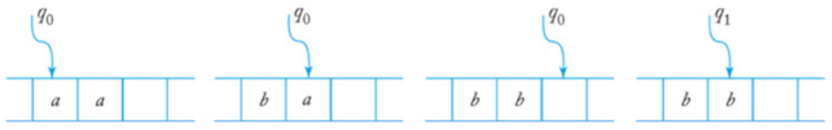


Figure 51: **Linz Fig. 9.3: A Sequence of Moves of a Turing Machine**

8.1.1.6 Transition Graph for Turing Machine

As with finite and push-down automata, we can use transition graphs to represent Turing machines. We label the edges of the graph with a triple giving (1) the current tape symbol, (2) the symbol that replaces it, and (3) the direction in which the read-write head moves.

Linz Figure 9.4 shows a transition graph for the Turing machine given in Linz Example 9.2.

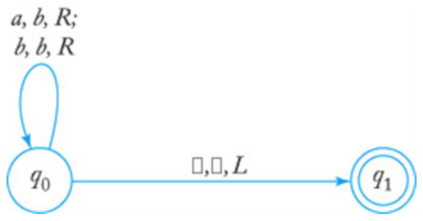


Figure 52: **Linz Fig. 9.4: Transition Graph for Example 9.2**

8.1.1.7 Linz Example 9.3 (Infinite Loop)

Consider the Turing machine defined in Linz Figure 9.5.

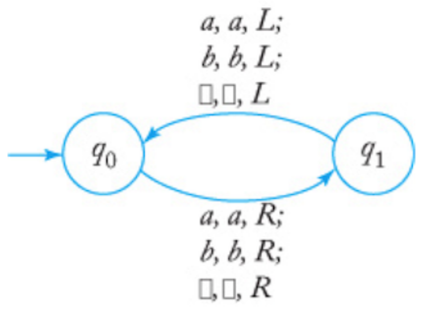


Figure 53: **Linz Fig. 9.5: Infinite Loop**

Suppose the tape initially contains $ab\ldots$ with the read-write head positioned over the a and in state q_0. Then the Turing machine executes the following sequence of moves:

1. The machine reads symbol a, leaves it unchanged, moves right (now over symbol b), and enters state q_1.
2. The machine reads b, leaves it unchanged, moves back left (now over a again), and enters state q_0 again.
3. The machine then repeats steps 1-3.

Clearly, regardless of the tape configuration, this machine does not halt. It goes into an *infinite loop*.

8.1.1.8 Standard Turing Machine

Because we can define a Turing machine in several different ways, it is useful to summarize the main features of our model.

A *standard Turing machine*:

1. has a tape that is unbounded in both directions, allowing any number of left and right moves
2. is deterministic in that δ defines at most one move for each configuration
3. has no special input or output files. At the initial time, the tape has some specified content, some of which is considered input. Whenever the machine halts, some or all of the contents of the tape is considered output.

These definitions are chosen for convenience in this chapter. Chapter 10 (which we do not cover in this course) examines alternative versions of the Turing machine concept.

8.1.1.9 Instantaneous Description of Turing Machine As with pushdown automata, we use *instantaneous descriptions* to examine the configurations in a sequence of moves. The notation (using strings)

$$x_1 q x_2$$

or (using individual symbols)

$$a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n$$

gives the instantaneous description of a Turing machine in state q with the tape as shown in Linz Figure 9.5.

By convention, the read-write head is positioned over the symbol to the right of the state (i.e., a_k above).

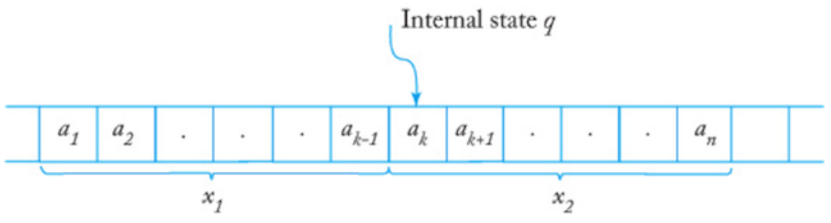


Figure 54: **Linz Fig. 9.6: Tape Configuration** $a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n$

A tape cell contains $\square$ if not otherwise defined to have a value.

Example: The diagrams in Linz Figure 9.3 (above) show the instantaneous descriptions q_0aa, bq_0a, $bbq_0\square$, and bq_1b.

As with pushdown automata, we use $\vdash$ to denote a *move*.

Thus, for transition rule

$$\delta(q_1, c) = (q_2, e, R)$$

we can have the move

$$abq_1cd \vdash abeq_2d.$$

As usual we denote the *transitive closure of move* (i.e., arbitrary number of moves) using:

$$\vdash^*$$

We also use subscripts to distinguish among machines:

$$\vdash_M$$

8.1.1.10 Computation of Turing Machine

Now let's summarize the above discussion with the following definitions.

Linz Definition 9.2 (Computation): Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing machine. Then any string $a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n$ with $a_i \in \Gamma$ and $q_1 \in Q$, is an *instantaneous description* of M.

A *move*

$$a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n \vdash a_1 \cdots a_{k-1} b q_2 a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta(q_1, a_k) = (q_2, b, R).$$

A *move*

$$a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n \vdash a_1 \cdots q_2 a_{k-1} b a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta(q_1, a_k) = (q_2, b, L).$$

M halts starting from some initial configuration $x_1 q_i x_2$ if

$$x_1 q_i x_2 \ \vdash^* \ y_1 q_j a y_2$$

for any q_j and a, for which $\delta(q_j, a)$ is undefined.

The sequence of configurations leading to a halt state is a *computation*.

If a Turing machine does not halt, we use the following *special notation* to describe its computation:

$$x_1 q x_2 \vdash^* \infty$$

8.1.2 Turing Machines as Language Acceptors

Can a Turing machine accept a string w?

Yes, using the following setup:

- Write w on the tape initially.
- Fill all the unused cells on the tape with blanks $\square$.
- Start the Turing machine with read-write head over leftmost symbol of w.
- If the machine halts in a final state, then it *accepts* string w.

Linz Definition 9.3 (Language Accepted by Turing Machine): Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ be a Turing machine. Then the *language accepted by* M is

$$L(M) = \{w \in \Sigma^+ : q_0 w \vdash^* x_1 q_f x_2, q_f \in F, x_1, x_2 \in \Gamma^*\}.$$

Note: The finite string w must be written to the tape with blanks on both sides. No blanks can are embedded within the input string w itself.

Question: What if $w \notin L(M)$?

The Turing machine might:

1. halt in nonfinal state
2. never halt

Any string for which the machine does not halt is, by definition, not in $L(M)$.

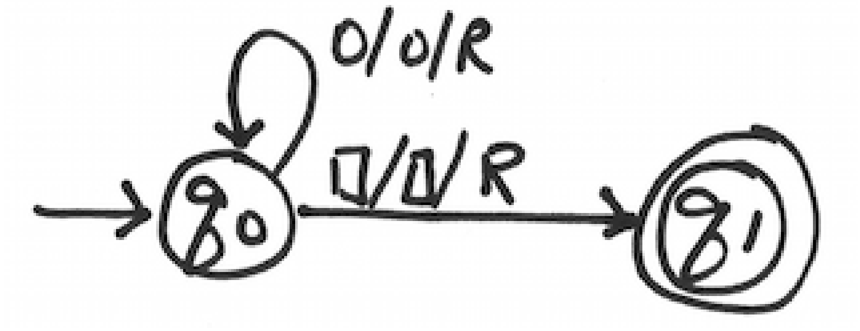
8.1.2.1 Linz Example 9.6

For $\Sigma = \{0, 1\}$, design a Turing machine that accepts the language denoted by the regular expression 00^*.

We use two internal states $Q = \{q_0, q_1\}$, one final state $F = \{q_1\}$, and transition function:

$$\delta(q_0, 0) = (q_0, 0, R),$$
$$\delta(q_0, \Box) = (q_1, \Box, R)$$

The transition graph shown below implements this machine.



- While a 0 appears under the read-write head, the head moves to the right.
- If a blank is read, the machine halts in final state q_1.
- If a 1 is read, the machine halts in the nonfinal state q_0 because $\delta(q_0, 1)$ is undefined.

The Turing machine also halts in a final state if started in state q_0 on a blank. We could interpret this as acceptance of λ, but for technical reasons the empty string is not included in Linz Definition 9.3.

8.1.2.2 Linz Example 9.7

For $\Sigma = \{a, b\}$, design a Turing machine that accepts

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

We can design a machine that incorporates the following algorithm:

```
While both $a$'s and $b$'s remain
    replace leftmost $a$ by $x$
    replace leftmost $b$ by $y$
If no $a$'s or $b$'s remain
    accept
else
    reject
```

Filling in the details, we get the following Turing machine for which:

$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$F = \{q_4\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, b, x, y, \Box\}$

The transitions can be broken into several sets.

The first set

1. $\delta(q_0, a) = (q_1, x, R)$
2. $\delta(q_1, a) = (q_1, a, R)$
3. $\delta(q_1, y) = (q_1, y, R)$
4. $\delta(q_1, b) = (q_2, y, L)$

replaces the leftmost a with an x, then causes the read-write head to travel right to the first b, replacing it with a y. The machine then enters a state q_2, indicating that an a has been successfully paired with a b.

The second set

5. $\delta(q_2, y) = (q_2, y, L)$
6. $\delta(q_2, a) = (q_2, a, L)$
7. $\delta(q_2, x) = (q_0, x, R)$

reverses the direction of movement until an x is encountered, repositions the read-write head over the leftmost a, and returns control to the initial state.

The machine is now back in the initial state q_0, ready to process the next a-b pair.

After one pass through this part of the computation, the machine has executed the partial computation:

$$q_0 aa \cdots abb \cdots b \vdash^* x q_0 a \cdots ayb \cdots b$$

So, it has matched a single a with a single b.

The machine continues this process until no a is found on leftward movement.

If all a's have been replaced, then state q_0 detects a y instead of an a and changes to state q_3. This state must verify that all b's have been processed as well.

8. $\delta(q_0, y) = (q_3, y, R)$
9. $\delta(q_3, y) = (q_3, y, R)$
10. $\delta(q_3, \Box) = (q_4, \Box, R)$

The input $aabb$ makes the moves shown below. (The bold number in parenthesis gives the rule applied in that step.)

		q_0aabb	– start at left end
(1)	$\vdash$	xq_1abb	– process 1st a-b pair
(2)	$\vdash$	xaq_1bb	– moving to right
(4)	$\vdash$	xq_1ayb	
(6)	$\vdash$	q_2xayb	– move back to left
(7)	$\vdash$	xq_0ayb	
(1)	$\vdash$	xxq_1yb	– process 2nd a-b pair
(3)	$\vdash$	$xxyq_1b$	– moving to right
(4)	$\vdash$	xxq_2yy	
(5)	$\vdash$	xq_2xyy	– move back to left
(7)	$\vdash$	xxq_0yy	
(8)	$\vdash$	$xxyq_3y$	– no a's
(9)	$\vdash$	$xxyyq_3\Box$	– check for extra b's
(10)	$\vdash$	$xxyy\Box q_4\Box$	– done, move to final

The Turing machine halts in final state q_4, thus accepting the string $aabb$.

If the input is not in the language, the Turing machine will halt in a nonfinal state.

For example, consider:

- a^nb^m for $n > m$?
  - halts in nonfinal state q_1 when $\Box$ found
- a^nb^m for $0 < n < m$?
  - halts in nonfinal state q_3 when b found
- aba?
  - halts in nonfinal state q_3 when a found
- b?
  - halts in nonfinal state q_0 when b found

8.1.3 Turing Machines as Transducers

Turing machines are more than just language accepters. They provide a simple abstract model for computers in general. Computers transform data. Hence, Turing machines are *transducers* (as we defined them in Chapter 1). For a computation, the

- *input* consists of all the nonblank symbols on the tape initially
- *output* consists of is whatever is on the tape when the machine halts in a final state

Thus, we can view a Turing machine transducer M as an implementation of a function f defined by

$$\hat{w} = f(w)$$

provided that

$$q_0 w \vdash_M^* q_f \hat{w},$$

for some final state q_f.

Linz Definition 9.4 (Turing Computable): A function f with domain D is said to be *Turing-computable*, or just *computable*, if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ such that

$$q_0 w \vdash_M^* q_f f(w),\ q_f \in F,$$

for all $w \in D$.

Note: A transducer Turing machine must start on the leftmost symbol of the input and stop on the leftmost symbol of the output.

8.1.3.1 Linz Example 9.9 Compute $x + y$ for positive integers x and y.

We use *unary notation* to represent the positive integers, i.e., a positive integer is represented by a sequence of 1's whose length is equal to the value of the integer. For example:

$$1111 = 4$$

The computation is

$$q_0 w(x) 0 w(y) \vdash^* q_f w(x+y) 0$$

where 0 separates the two numbers at initiation and after the result at termination.

Key idea: Move the separating 0 to the right end.

To achieve this, we construct $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ with

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$
$$F = \{q_4\}$$

$\delta(q_0, 1) = (q_0, 1, R)$
$\delta(q_0, 0) = (q_1, 1, R)$
$\delta(q_1, 1) = (q_1, 1, R)$
$\delta(q_1, \square) = (q_2, \square, L)$
$\delta(q_2, 1) = (q_3, 0, L)$
$\delta(q_3, 1) = (q_3, 1, L)$
$\delta(q_3, \square) = (q_4, \square, R)$

The sequence of instantaneous descriptions for adding 111 to 11 is shown below.

$$\begin{array}{ll}
q_0111011 & \vdash 1q_011011 \vdash 11q_01011 \vdash 111q_0011 \\
 & \vdash 1111q_1111 \vdash 11111q_11 \vdash 111111q_1\square \\
 & \vdash 11111q_21 \vdash 1111q_310 \vdash 111q_3110 \\
 & \vdash 11q_31110 \vdash 1q_311110 \vdash q_3111110 \\
 & \vdash q_3\square111110 \vdash q_4111110
\end{array}$$

8.1.3.2 Linz Example 9.10

Construct a Turing machine that copies strings of 1's. More precisely, find a machine that performs the computation

$$q_0w \vdash^* q_fww,$$

for any $w \in \{1\}^+$.

To solve the problem, we implement the following procedure:

1. Replace every 1 by an x.
2. Find the rightmost x and replace it with 1.
3. Travel to the right end of the current nonblank region and create a 1 there.
4. Repeat steps 2 and 3 until there are no more x's.

A Turing machine transition function for this procedure is as follows:

$\delta(q_0, 1) = (q_0, x, R)$
$\delta(q_0, \square) = (q_1\square, L)$
$\delta(q_1, x) = (q_2, 1, R)$
$\delta(q_2, 1) = (q_2, 1, R$
$\delta(q_2, \square) = (q_1, 1, L)$
$\delta(q_1, 1) = (q_1, 1, L)$
$\delta(q_1, \square) = (q_3, \square, R)$

where q_3 is the only final state.

Linz Figure 9.7 shows a transition graph for this Turing machine.

This is not easy to follow, so let us trace the program with the string 11. The computation performed is as shown below.

$$\begin{array}{ll}
q_011 & \vdash xq_01 \vdash xxq_0\square \vdash xq_1x \\
 & \vdash x1q_2\square \vdash xq_111 \vdash q_1x11
\end{array}$$

$$
\begin{array}{l}
\vdash\ 1q_211\ \vdash\ 11q_21\ \vdash\ 111q_2\square \\
\vdash\ 11q_111\ \vdash\ 1q_1111 \\
\vdash\ q_11111\ \vdash\ q_1\square 1111\ \vdash\ q_31111
\end{array}
$$

8.1.3.3 Linz Example 9.11 Suppose x and y are positive integers represented in unary notation.

Construct a Turing machine that halts in a final state q_y if $x \geq y$ and in a nonfinal state q_n if $x < y$.

That is, the machine must perform the computation:

$$
\begin{array}{l}
q_0w(x)0w(y) \vdash^* q_yw(x)0w(y), \text{ if } x \geq y \\
q_0w(x)0w(y) \vdash^* q_nw(x)0w(y), \text{ if } x < y
\end{array}
$$

We can adapt the approach from Linz Example 9.7. Instead of matching a's and b's, we match each 1 on the left of the dividing 0 with the 1 on the right. At the end of the matching, we will have on the tape either

$$xx \cdots 110xx \cdots x\square$$

or

$$xx \cdots xx0xx \cdots x11\square,$$

depending on whether $x > y$ or $y > x$.

A transition graph for machine is shown below.

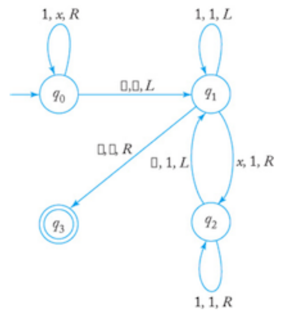
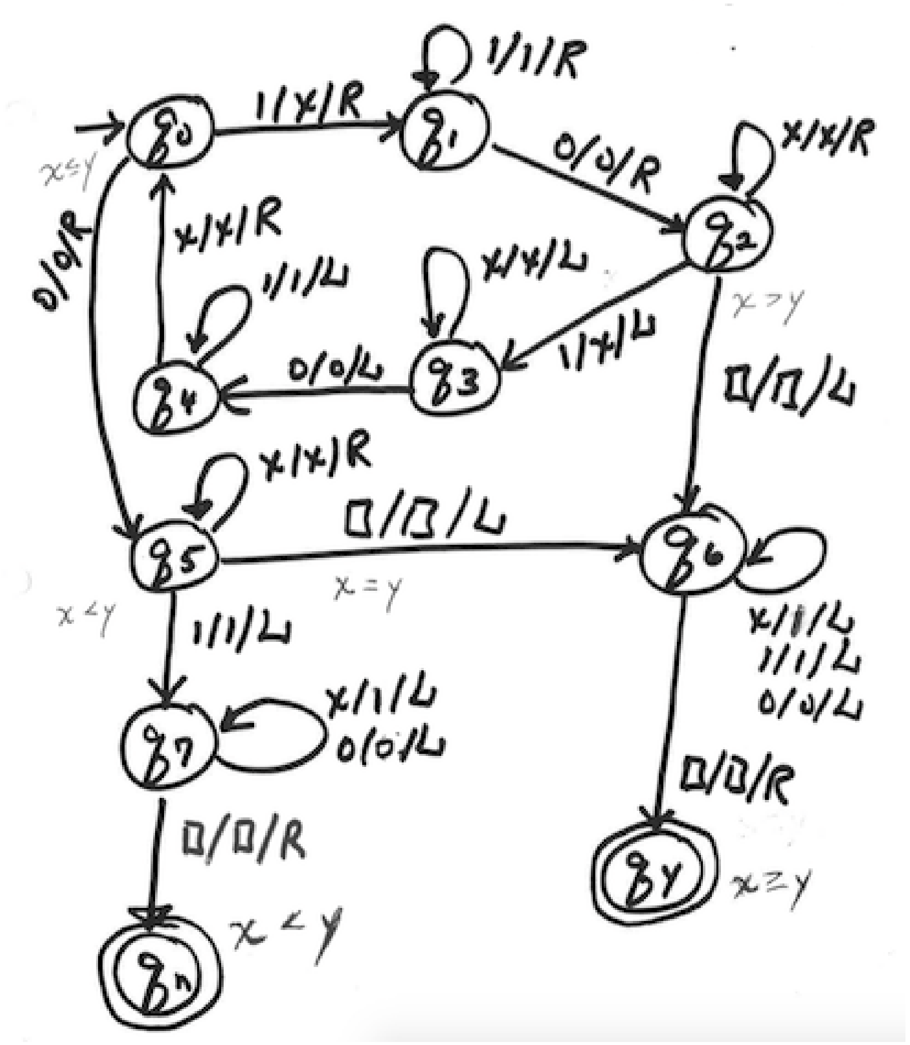


Figure 55: **Linz Fig. 9.7: Transition Graph for Example 9.10**



8.2 Combining Turing Machines for Complicated Tasks

8.2.1 Introduction

How can we compose simpler operations on Turing machines to form more complex operations?

Techniques discussed in this section include use of:

- *Top-down stepwise refinement*, i.e., starting with a high-level description and refining it incrementally until we obtain a description in the actual language
- Block diagrams or pseudocode to state high-level descriptions

8.2.2 Using Block Diagrams

In the *block diagram* technique, we define high-level computations in boxes without internal details on how computation is done. The details are filled in on a subsequent refinement.

To explore the use of block diagrams in the design of complex computations, consider Linz Example 9.12, which builds on Linz Examples 9.9 and 9.11 (above).

8.2.2.1 Linz Example 9.12 Design a Turing machine that computes the following function:

$f(x, y) = x + y$, if $x \geq y$,
$f(x, y) = 0$, if $x < y$.

For simplicity, we assume x and y are positive integers in unary representation and the value zero is represented by 0, with the rest of the tape blank.

Linz Figure 9.8 shows a high-level block diagram of this computation. This computation consists of a network of three simpler machines:

- a Comparer C to determine whether or not $x \geq y$
- an Adder A that computes $x + y$
- an Eraser E that changes every 1 to a blank

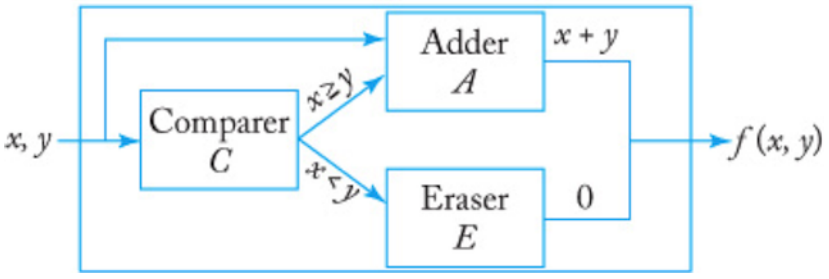


Figure 56: **Linz Fig. 9.8: Block Diagram**

We use such high-level diagrams in subsequent discussions of large computations. How can we justify that practice?

We can implement:

- the Comparer program C as suggested in Linz Example 9.11, using a Turing machine having states indexed with C
- the Adder program A as suggested in Linz Example 9.9, with states indexed with A
- the Eraser program E by constructing a Turing machine having states indexed with E

Comparer C carries out the computations

$$q_{C,0}w(x)0w(y) \vdash^* q_{A,0}w(x)0w(y), \text{ if } x \geq y,$$

and

$$q_{C,0}w(x)0w(y) \vdash^* q_{E,0}w(x)0w(y), \text{ if } x < y.$$

If $q_{A,0}$ and $q_{E,0}$ are the initial states of computations A and E, respectively, then C starts either A or E.

Adder A carries out the computation

$$q_{A,0}w(x)0w(y) \vdash^* q_{A,f}w(x+y)0.$$

And, Eraser E carries out the computation

$$q_{E,0}w(x)0w(y) \vdash^* q_{E,f}0.$$

The outer diagram in Linz Figure 9.8 thus represents a single Turing machine that combines the actions of machines C, A, and E as shown.

8.2.3 Using Pseudocode

In the *pseudocode* technique, we outline a computation using high-level descriptive phrases understandable to people. We refine and translate it to lower-level implementations later.

8.2.3.1 Macroinstructions

A simple kind of pseudocode is the macroinstruction. A *macroinstruction* is a single statement shorthand for a sequence of lower-level statements.

We first define the macroinstructions in terms of the lower-level language. Then we compose macroinstructions into a larger program, assuming the relevant substitutions will be done.

8.2.3.2 Linz Example 9.13

For this example, consider the macroinstruction

if a then q_j else q_k.

This means:

- If the Turing machine reads an a, then it, regardless of its current state, transitions into state q_j without changing the tape content or moving the read-write head.
- If the symbol read is not an a, then it transitions into state q_k without changing anything.

We can implement this macroinstruction with several steps of a Turing machine:

$\delta(q_i, a) = (q_{j0}, a, R)$ for all $q_i \in Q$
$\delta(q_{j0}, c) = (q_j, c, L)$ for all $c \in \Gamma$

$\delta(q_i, b) = (q_{k0}, b, R)$ for all $q_i \in Q$ and all $b \in \Gamma - \{a\}$
$\delta(q_{k0}, c) = (q_k, c, L)$ for all $c \in \Gamma$

States q_{j0} and q_{k0} just back up Turing machine tape position one place.

Macroinstructions are expanded at each occurrence.

8.2.3.3 Subprograms

While each occurrence of a macroinstruction is expanded into actual code, a *subprogram* is a single piece of code that is invoked repeatedly.

As in higher-level language programs, we must be able to call a subprogram and then, after execution, return to the calling point and resume execution without any unwanted effects.

How can we do this with Turing machines?

We must be able to:

- preserve information about the calling program's configuration (state, read-write head position, tape contents), so that it can be restored on return from the subprogram
- pass information from the calling program to the called subprogram and vice versa

We can do this by partitioning the tape into several regions. Linz Figure 9.9 illustrates this technique for a program A (a Turing machine) that calls a subprogram B (another Turing machine).

1. A executes in its own workspace.
2. Before transferring control to B, A writes information about its configuration and inputs for B into some separate region T.
3. After transfer, B finds its input in T.
4. B executes in its own separate workspace.
5. When B completes, it writes relevant results into T.
6. B transfers control back to A, which resumes and gets the needed results from T.

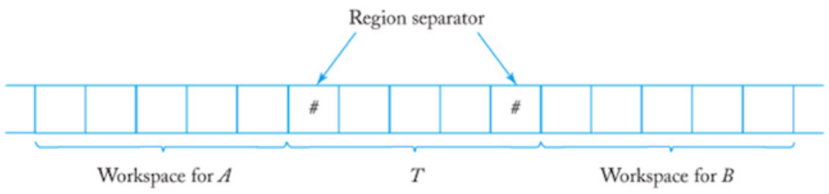


Figure 57: **Linz Fig. 9.9: Tape Regions for Subprograms**

Note: This is similar to what happens in an actual computer for a subprogram (function, procedure) call. The region T is normally a segment pushed onto the program's runtime stack or dynamically allocated from the heap memory.

8.2.3.4 Linz Example 9.14

Design a Turing machine that multiplies x and y, positive integers represented in unary notation.

Assume the initial and final tape configurations are as shown in Linz Figure 9.10.

We can multiply x by y by adding y to itself x times as described in the algorithm below.

```
Repeat until $x$ contains no more 1's\
   Find a 1 in $x$ and replace it with another symbol $a$\
   Replace the leftmost 0 by $0y$\
Replace all $a$'s with 1's
```

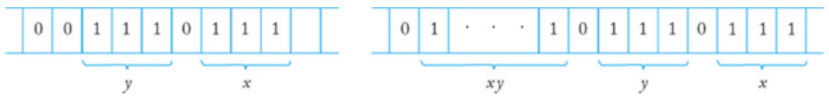


Figure 58: **Linz Fig. 9.10: Multiplication**

Although the above description of the pseudocode approach is imprecise, the idea is sufficiently simple that it is clear we can implement it.

We have not proved that the block diagram, macroinstruction, or subprogram approaches will work in all cases. But the discussion in this section shows that it is plausible to use Turing machines to express complex computations.

8.3 Turing's Thesis

The *Turing thesis* is an hypothesis that *any computation that can be carried out by mechanical means can be performed by some Turing machine.*

This is a broad assertion. It is not something we can prove!

The Turing thesis is actually a definition of mechanical computation: *a computation is mechanical if and only if it can be performed by some Turing machine.*

Some arguments for accepting the Turing thesis as the definition of mechanical computation include:

1. Anything that can be computed by any existing digital computer can also be computed by a Turing machine.
2. There are no known problems that are solvable by what we intuitively consider an algorithm for which a Turing machine program cannot be written.
3. No alternative model for mechanical computation is more powerful than the Turing machine model.

The Turing thesis is to computing science as, for example, classical Newtonian mechanics is to physics. Newton's "laws" of motion cannot be proved, but they could possibly be invalidated by observation. The "laws" are plausible models that have enabled humans to explain much of the physical world for several centuries.

Similarly, we accept the Turing thesis as a basic "law" of computing science. The conclusions we draw from it agree with what we know about real computers.

The Turing thesis enables us to formalize the concept of algorithm.

Linz Definition 9.5 (Algorithm): An *algorithm* for a function $f : D \to R$ is a Turing machine M, which given as input any $d \in D$ on its tape, eventually halts with the correct answer $f(d) \in R$ on its tape. Specifically, we can require that

$$q_0 d \vdash_M^* q_f f(d), q_f \in F,$$

for all $d \in D$.

To prove that "there exists an algorithm", we can construct a Turing machine that computes the result.

However, this is difficult in practice for such a low-level machine.

An alternative is, first, to appeal to the Turing thesis, arguing that anything that we can compute with a digital computer we can compute with a Turing machine. Thus a program in suitable high-level language or precise pseudocode can compute the result. If unsure, then we can validate this by actually implementing the computation on a computer.

Note: A higher-level language is *Turing-complete* if it can express any algorithm that can be expressed with a Turing machine. If we can write a Turing machine simulator in that language, we consider the language Turing complete.

9 OMIT Chapter 10

10 A Hierarchy of Formal Languages and Automata

The kinds of questions addressed in this chapter:

- What is the family of languages accepted by Turing machines?
- Are there any languages that are not accepted by any Turing machine?
- What is the relationship between Turing machines and various kinds of grammars?
- How can we classify the various families of languages and their relationships to one another?

Note: We assume the languages in this chapter are λ-free unless otherwise stated.

10.1 Recursive and Recursively Enumerable Languages

Here we make a distinction between languages accepted by Turing machines and languages for which there is a membership algorithm.

10.1.1 Aside: Countability

Definition (Countable and Countably Infinite): A set is *countable* if it has the same cardinality as a subset of the natural numbers. A set is *countably infinite* if it can be placed into one-to-one correspondence with the set of all natural numbers.

Thus there is some ordering on any countable set.

Also note that, for any finite set of symbols Σ, then Σ^* and any its subsets are countable. Similarly for Σ^+.

From Linz Section 10.4 (not covered in this course), we also have the following theorem about the set of Turing machines.

Linz Theorem 10.3 (Turing Machines are Countable): The set of all Turing machines is countably infinite.

10.1.2 Definition of Recursively Enumerable Language

Linz Definition 11.1 (Recursively Enumerable Language): A language L is *recursively enumerable* if there exists a Turing machine that accepts it.

This definition implies there is a Turing machine M such that for every $w \in L$

$$q_0 w \vdash_M^* x_1 q_f x_2$$

with the initial state q_0 and a final state q_f.

But what if $w \notin L$?

- M might halt in a nonfinal state.
- M might go into an infinite loop.

10.1.3 Definition of Recursive Language

Linz Definition 11.2 (Recursive Language): A language L on Σ is *recursive* if there exists a Turing machine M that accepts L and that halts on every w in Σ^*.

That is, a language is recursive if and only if there exists a *membership algorithm* for it.

10.1.4 Enumeration Procedure for Recursive Languages

If a language is recursive, then there exists an *enumeration procedure*, that is, a method for counting and ordering the strings in the language.

- Let M be a Turing machine that determines membership in a recursive language L on an alphabet Σ.
- Let M' be M modified to write the accepted strings to its tape.
- Σ^+ is countable, so there is some ordering of $w \in \Sigma^+$. Construct Turing machine $\hat{M}$ that generates all $w \in \Sigma^+$ in order, say $w_1, w_2, \cdots$.

Thus $\hat{M}$ generates the candidate strings w_i in order. M' writes the the accepted strings to its tape in order.

10.1.5 Enumeration Procedure for Recursively Enumerable Languages

Problem: A Turing machine M might not halt on some strings.

Solution: Construct $\hat{M}$ to advance "all" strings simultaneously, one move at a time. The order of string generation and moves is illustrated in Linz Figure 11.1.

Now machine $\hat{M}$ advances each candidate string w_i (columns of Linz Figure 11.1) one M-move at a time.

Because each string is generated by $\hat{M}$ and accepted by M in a finite number of steps, every string in L is eventually produced by M. The machine does not go into an infinite loop for a w_i that is not accepted.

Note: Turing machine $\hat{M}$ does not terminate and strings for which M does not halt will never complete processing, but any string that can be accepted by M will be accepted within a finite number of steps.

10.1.6 Languages That are Not Recursively Enumerable

Linz Theorem 11.1 (Powerset of Countable Set not Countable) Let S be an countably infinite set. Then its powerset 2^S is not countable.

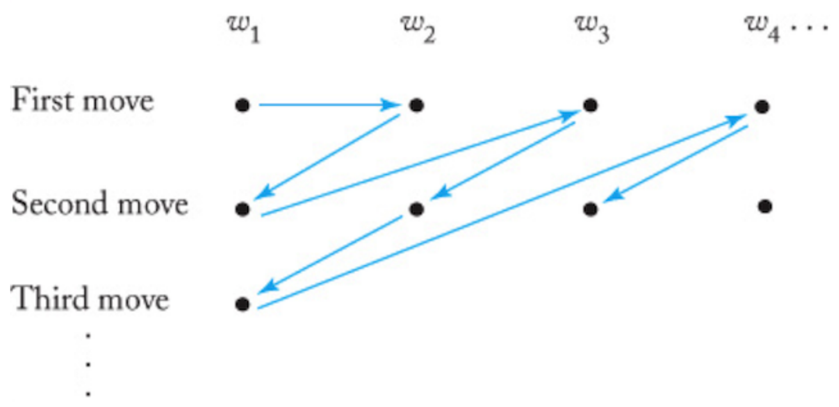
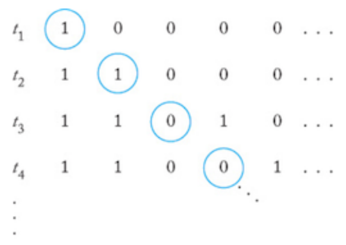


Figure 59: **Linz Fig. 11.1: Enumeration Procedure for Recursively Enumerable Languages**

Proof: Let $S = \{\ s_1, s_2, s_3, \cdots\ \}$ be an countably infinite set.

Let $t \in 2^S$. Then t can represented by a bit vector $b_1 b_2 \cdots$ such that $b_i = 1$ if and only if $s_i \in t$.

Assume 2^S is countable. Thus 2^S can be written in order $t_1, t_2, \cdots$ and put into a table as shown in Linz Figure 11.2.



$$\begin{array}{cccccccc}
t_1 & 1 & 0 & 0 & 0 & 0 & \cdots \\
t_2 & 1 & 1 & 0 & 0 & 0 & \cdots \\
t_3 & 1 & 1 & 0 & 1 & 0 & \cdots \\
t_4 & 1 & 1 & 0 & 0 & 1 & \cdots \\
\vdots & & & & & \ddots &
\end{array}$$

Figure 60: **Linz Fig. 11.2: Cantor's Diagonalization**

Consider the main diagonal of the table (circled in Linz Figure 11.2). Complement the bits along this diagonal and let t_d be a set represented by this bit vector.

Clearly $t_d \in 2^S$. But $t_d \neq t_i$ for any i, because they differ at least at s_i. This is a contradicts the assumption that 2^S is countable.

So the assumption is false. Therefore, 2^S is *not* countable. QED.

This is *Cantor's diagonalization* argument.

Linz Theorem 11.2 (Existence of Languages Not Recursively Enumerable): For any nonempty Σ, there exist languages that are not recursively enumerable.

Proof: Any $L \subseteq \Sigma^*$ is a language on Σ. Thus 2^{Σ^*} is the set of all languages on Σ.

Because Σ^* is infinite and countable, Linz Theorem 11.1 implies that the set of all languages on Σ is *not* countable. From Linz Theorem 10.3 (see above), we know the set of Turing machines can be enumerated. Hence, the recursively enumerable languages are countable.

Therefore, some languages on Σ are *not* recursively enumerable. QED.

10.1.7 A Language That is Not Recursively Enumerable

Linz Theorem 11.3: There exists a recursively enumerable language whose complement is not recursively enumerable.

Proof: Let $\Sigma = \{a\}$.

Consider the set of all Turing machines with input alphabet Σ, i.e., $\{M_1, M_2, M_3, \cdots\}$.

By Linz Theorem 10.3 (see above), we know that this set of is countable. So it has some order.

For each M_i there exists a recursively enumerable language $L(M_i)$.

Also, for each recursively enumerable languages on Σ, there is some Turing machine that accepts it.

Let $L = \{a^i : a^i \in L(M_i)\}$.

L is recursively enumerable because here is a Turing machine that accepts it. E.g., the Turing machine works as follows:

- Count a's in the input w to get i.
- Use Turing machine M_i to accept w.
- The combined Turing machine thus accepts L.

Now consider $\bar{L} = \{a^i : a^i \notin L(M_i)\}$.

Assume $\bar{L}$ is recursively enumerable.

There must be some Turing machine M_k, for some k, that accepts $\bar{L}$. Hence, $\bar{L} = L(M_k)$.

Consider a^k. Is it in L? Or in $\bar{L}$?

Consider the case $a^k \in \bar{L}$. Thus $a^k \in L(M_k)$. Hence, $a^k \in L$ by the definition of L. This is a contradiction.

Consider the case $a^k \in L$, i.e., $a^k \notin \bar{L}$. Thus $a^k \notin L(M_k)$ by definition of $\bar{L}$. But from the defintion of L, $a^k \in \bar{L}$. This is also be a contradiction.

In all cases, we have a contradiction, so the assumption is false. Therefore, $\bar{L}$ is not recursively enumerable. QED.

10.1.8 A Language That is Recursively Enumerable but Not Recursive

Linz Theorem 11.4: If a language L and its complement $\bar{L}$ are both recursively enumerable, then both languages are recursive. If L is recursive, then $\bar{L}$ is also recursive, and consequently both are recursively enumerable.

Proof: See Linz Section 11.2 for the details.

Linz Theorem 11.5: There exists a recursively enumerable language that is not recursive; that is, the family of recursive languages is a proper subset of the family of recursively enumerable languages.

Proof: Consider the language L of Linz Theorem 11.3.

This language is recursively enumerable, but its complement is not. Therefore, by Linz Theorem 11.4, it is not recursive, giving us the required example. QED.

There are well-defined languages that have no membership algorithms.

10.2 Unrestricted Grammars

Linz Definition 11.3 (Unrestricted Grammar): A grammar $G = (V, T, S, P)$ is an *unrestricted gramar* if all the productions are of the form

$$u \rightarrow v,$$

where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$.

Note: There is no λ on left, but otherwise the use of symbols is unrestricted.

Linz Theorem 11.6 (Recursively Enumerable Language for Unrestricted Grammar): Any language generated by an unrestricted grammar is recursively enumerable.

Proof: See Linz Section 11.2 for the details.

The grammar defines an enumeration procedure for all strings.

Linz Theorem 11.7 (Unrestricted Grammars for Recursively Enumerable Language): For every recursively enumerable language L, there exists an unrestricted grammar G, such that $L = L(G)$.

Proof: See Linz Section 11.2 for the details.

10.3 Context-Sensitive Grammars and Languages

Between the restricted context-free grammars and the unrestricted grammars, there are a number of kinds of "somewhat restricted" families of grammars.

Linz Definition 11.4 (Context-Sensitive Grammar): A grammar $G = (V, T, S, P)$ is said to be *context-sensitive* if all productions are of the form

$$x \rightarrow y,$$

where $x, y \in (V \cup T)^+$ and

$$|x| \leq |y|.$$

This type of grammar is *noncontracting* in that the length of successive sentential forms can never decrease.

All such grammars can be rewritten in a normal form in which all productions are of the form

$$xAy \rightarrow xvy.$$

This is equivalent to saying that the production

$$A \rightarrow v$$

can only be applied in a *context* where A occurs with string x on the left and string y on the right.

Linz Definition 11.5 (Context-Sensitive) : A language L is said to be *context-sensitive* if there exists a context-sensitive grammar G, such that $L = L(G)$ or $L = L(G) \cup \{\lambda\}$.

Note the special cases for λ. This enables us to say that the family of context-free languages is a subset of the family of context-sensitive languages.

10.3.1 Linz Example 11.2

The language $L = \{a^n b^n c^n : n \geq 1\}$ is a context-sensitive language. We show this by defining a context-sensitive grammar for the language, such as the following:

$$\begin{aligned}
&S \rightarrow abc \mid aAbc \\
&Ab \rightarrow bA \\
&Ac \rightarrow Bbcc \\
&bB \rightarrow Bb \\
&aB \rightarrow aa \mid aaA
\end{aligned}$$

Consider a derivation of $a^3b^3c^3$:

$$\begin{array}{ll}
S & \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \\
 & \Rightarrow aBbbcc \Rightarrow aaAbbcc \Rightarrow aabAbcc \\
 & \Rightarrow aabbAcc \Rightarrow aabbBbccc \Rightarrow aabBbbccc \\
 & \Rightarrow aaabbbccc
\end{array}$$

The grammar uses the variables A and B as messengers.

- An A is created on the left, travels to the right to the first c, where it creates another b and c.
- Messanger B is sent back to the left to create the corresponding a.

The process is similar to how a Turing machine would work to accept the language L.

L is not context-free.

10.3.2 Linear Bounded Automata (lba)

In Linz Section 10.5 (not covered in this course), a *linear-bounded automaton* is defined as a nondeterministic Turing machine that is restricted to the part of its tape occupied by its input (bounded on the left by [and right by]).

[_________].

Linz Theorem 11.8: For every context-sensitive language L not including λ, there exists some linear bounded automaton M such that $L = L(M)$:

Proof: See Linz Section 11.3 for the details.

Linz Theorem 11.9: If a language L is accepted by some linear bounded automaton M, then there exists a context-sensitive grammar that generates L.

Proof: See Linz Section 11.3 for the details.

10.3.3 Relation Between Recursive and Context-Sensitive Languages

Linz Theorem 11.10: Every context-sensitive language L is recursive.

Linz Theorem 11.11: There exists a recursive language that is not context-sensitive.

We have studied a number of automata in this course. Ordered by decreasing power these include:

- Turing machine (accept recursively enumerable languages)
- linear-bounded automata (accept context-sensitive languages)

- npda (accept context-free languages)
- dpda (accept deterministic context-free languages)
- nfa, dfa (accept regular languages)

10.4 The Chomsky Hierarchy

We have studied a number of types of languages in this course, including

0. recursively enumerable languages L_{RE}
1. context-sensitive languages L_{CS}

2. context-free languages L_{REG}
3. regular languages L_{REG}

One way of showing the relationship among these families of languages is to use the *Chomsky hierarchy*, where the types are numbered as above and as diagrams in Linz Figures 11.3 and 11.4.

This classification was first described in 1956 by American linguist Noam Chomsky, a founder of formal language theory.

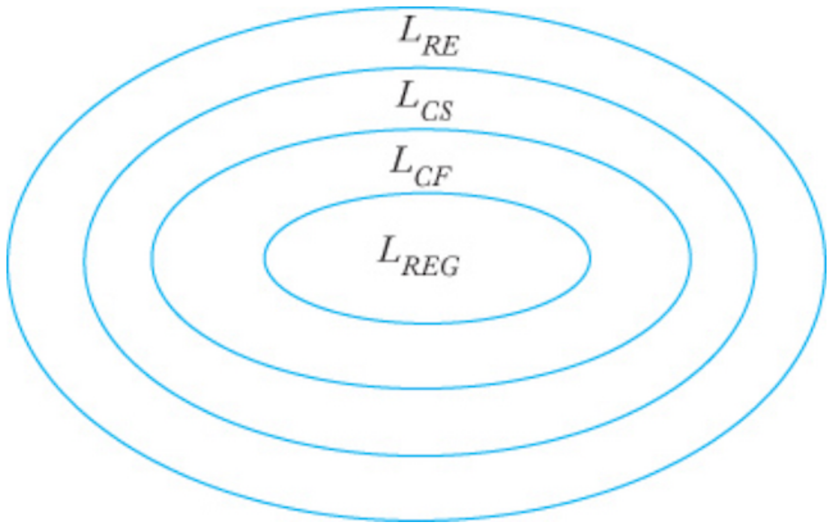


Figure 61: **Linz Fig 11.3: Original Chomsky Hierarchy**

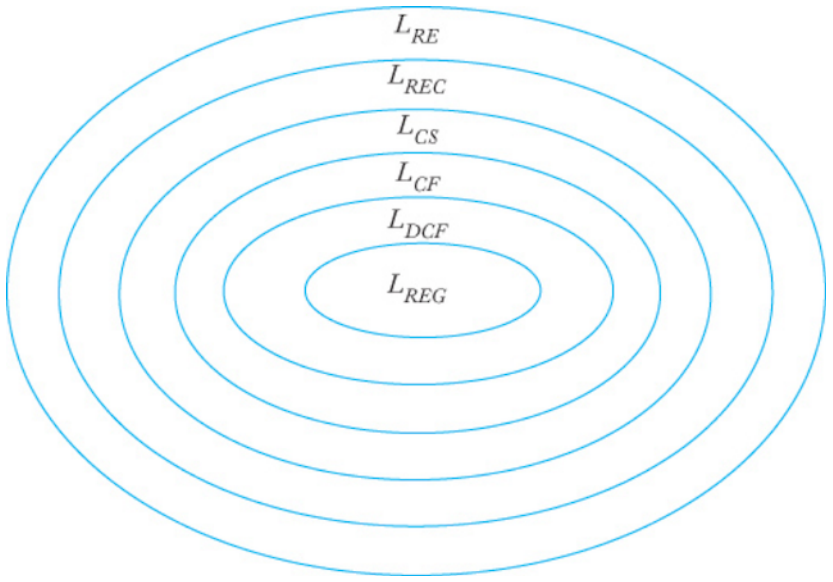


Figure 62: **Linz Fig 11.4: Extended Chomsky Hierarchy**

11 Limits of Algorithmic Computation

In Linz Chapter 9, we studied the *Turing thesis*, which concerned *what Turing machines can do*.

This chapter we study: What Turing machines *cannot* do.

This chapter considers the concepts:

- computability
- decidability

11.1 Some Problems That Cannot Be Solved with Turing Machines

11.1.1 Computability

Recall the following definition from Chapter 9.

Linz Definition 9.4 (Turing Computable): A function f with domain D is said to be *Turing-computable*, or just *computable*, if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ such that

$$q_0 w \vdash^*_M q_f f(w),\ q_f \in F,$$

for all $w \in D$.

Note:

- A function f can be computable only if it is defined on the entire domain D.
- Otherwise, f is *uncomputable.*
- So the domain of f is crucial to the issue of computability.

11.1.2 Decidability

Here we work in a simplified setting: the result of a computation is either "yes" or "no". In this context, the problem is considered either decidable or undecidable.

Problem: We have a set of related statements, each either true or false.

This problem is *decidable* if and only if there exists a Turing machine that gives the correct answer for every statement in the domain. Otherwise, the problem is *undecidable.*

Example problem statement: For a context-free grammar G, the language $L(G)$ is ambiguous. This is a true statement for some G and false for others.

If we can answer this question, with either the result true or false, for every context-free grammar, then the problem is decidable. If we cannot answer the question for some context-free grammar (i.e., the Turing machine does not halt), then the problem is undecidable.

(In Linz Theorem 12.8, we see that this question is actually undecidable.)

11.1.3 The Turing Machine Halting Problem

Given the description of a Turing machine M and input string w, does M, when started in the initial configuration q_0w, perform a computation that eventually halts?

What is the domain D?

- all Turing machines and all strings w on the Turing machine's alphabet

We cannot solve this problem by simulating M. That is an infinite computation if the Turing machine does not halt.

We must analyze the Turing machine description to get an answer for any machine M and string w. *But no such algorithm exists!*

Linz Definition 12.1 (Halting Problem): Let w_M be a string that describes a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ and let w be a string in M's alphabet. Assume that w_M and w are encoded as strings of 0's and 1's (as suggested in Linz Section 10.4). A solution to the *halting problem* is a Turing machine H, which for any w_M and w, performs the computation

$$q_0 w_M w \ \vdash^* \ x_1 q_y x_2$$

if M is applied to w halts, and

$$q_0 w_M w \ \vdash^* \ y_1 q_n y_2,$$

if M is applied to w does not halt. Here q_y and q_n are both final states of H.

Linz Theorem 12.1 (Halting Problem is Undecidable): There does not exist any Turing machine H that behaves as required by Linz Definition 12.1. Thus the *halting problem is undecidable.*

Proof: Assume there exists such a Turing machine H that solves the halting problem.

The input to H is $w_M w$, where w_M is a description of Turing machine M. H must halt with a "yes" or "no" answer as indicated in Linz Figure 12.1.

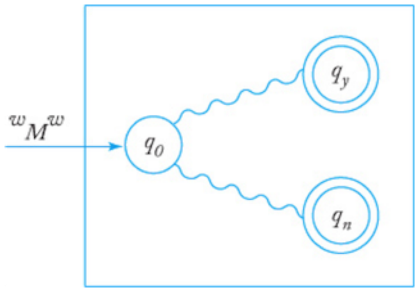


Figure 63: **Linz Fig. 12.1: Turing Machine H**

We next modify H to produce a Turing machine H' with the structure shown in Linz Figure 12.2.

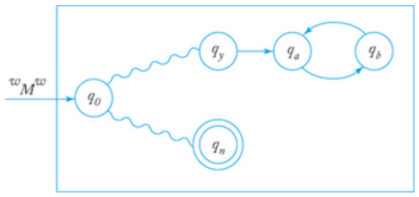


Figure 64: **Linz Fig. 12.2: Turing Machine H'**

When H' reaches a state where H halts, it enters an infinite loop.

From H' we construct Turing machine $\hat{H}$, which takes an input w_M and copies it, ending in initial state q_0 of H'. After that, it behaves the same as H'.

The behavior of $\hat{H}$ is

$$q_0 w_M \vdash^*_{\hat{H}} q_0 w_M w_M \vdash^*_{\hat{H}} \infty$$

if M applied to w_M halts, and

$$q_0 w_M \vdash^*_{\hat{H}} q_0 w_M w_M \vdash^*_{\hat{H}} y_1 q_n y_2$$

if M applied to w_M does not halt.

Now $\hat{H}$ is itself a Turing machine, which can be also be encoded as a string $\hat{w}$.

So, let's apply $\hat{H}$ to its own description $\hat{w}$. The behavior is

$$q_0\hat{w} \vdash^*_{\hat{H}} \infty$$

if $\hat{H}$ applied to $\hat{w}$ halts, and

$$q_0\hat{w} \vdash^*_{\hat{H}} y_1 q_n y_2$$

if M applied to $\hat{w}$ does not halt.

In the first case, $\hat{H}$ goes into an infinite loop (i.e., does not halt) if $\hat{H}$ halts. In the second case, $\hat{H}$ halts if $\hat{H}$ does not halt. This is clearly impossible!

Thus we have a contradiction. Therefore, there exists no Turing machine H. The halting problem is undecidable. QED.

It may be possible to determine whether a Turing machine halts in specific cases by analyzing the machine and its input.

However, this theorem says that there exists no algorithm to solve the halting problem for all Turing machines and all possible inputs.

Linz Theorem 12.2: If the halting problem were decidable, then every recursively enumerated language would be recursive. Consequently, the halting problem is undecidable.

Proof: Let L be a recursively enumerable language on Σ, M be a Turing machine that accepts L, and w_M be an encoding of M as a string.

Assume the halting problem is decidable and let H be a Turing machine that solves it.

Consider the following procedure.

1. Apply H to $w_M w$.
2. If H says "no", then $w \notin L$.
3. If H says "yes", then apply M to w, which will eventually tell us whether $w \in L$ or $w \notin L$.

The above is thus a membership algorithm, so L must be recursive. But we know that there are recursively enumerable languages that are not recursive. So this is a contradiction.

Therefore, H cannot exist and the halting problem is undecidable. QED.

11.1.4 Reducing One Undecidable Problem to Another

In the above, the halting problem is *reduced* to a membership algorithm for recursively enumerable languages.

A problem A is *reduced* to problem B if the decidability of B implies the decidability of A. We transform a new problem A into a problem B whose decidability is already known.

Note: The Linz textbook gives three example reductions in Section 12.1

11.2 Undecidable Problems for Recursively Enumerable Languages

Linz Theorem 12.3 (Empty Unrestricted Grammars Undecidable): Let G be an unrestricted grammar. Then the problem of determining whether or not

$$L(G) = \emptyset$$

is undecidable.

Proof: See Linz Section 12.2 for the details of this reduction argument. The decidability of membership problem for recursively enumerated languages implies the problem in this theorem.

Linz Theorem 12.4 (Finiteness of Turing Machine Languages is Undecided): Let M be a Turing Machine. Then the question of whether or not $L(M)$ is finite is undecidable.

Proof: See Linz Section 12.2 for the details of this proof.

Rice's theorem, a generalization of the above, states that any nontrivial property of a recursively enumerable language is undecidable. The adjective "nontrivial" refers to a property possessed by some but not all recursively enumerated languages.

11.3 The Post Correspondence Problem

This section is not covered in this course.

11.4 Undecidable Problems for Context-Free Languages

Linz Theorem 12.8: There exists no algorithm for deciding whether any given context-free grammar is ambiguous.

Proof: See Linz Section 12.4 for the details of this proof.

Linz Theorem 12.9: There exists no algorithm for deciding whether or not

$$L(G_1) \cap L(G_2) = \emptyset$$

for arbitrary context-free grammars G_1 and G_2.

Proof: See Linz Section 12.4 for the details of this proof.

Keep in mind that the above and other such decidability results do not eliminate the possibility that there may be specific cases–perhaps even many interesting and important cases–for which there exist decision algorithms.

However, these theorems do say that there are no general algorithms to decide these problems. There are always some cases in which specific algorithms will fail to work.

11.5 A Question of Efficiency

This section is not covered in this course.

11.6 References