Notes on Models of Computation
Chapter 1

H. Conrad Cunningham

06 April 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.

Note: These notes were written primarily to accompany my use of Chapter 1 of the Linz textbook *An Introduction to Formal Languages and Automata* [[1].

1 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?

1.1 Mathematical Preliminaries and Notation

The mathematical concepts used in the Linz textbook include:

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

1.1.1 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:

- $S_1 \bar{\cup} S_2 = \bar{S}_1 \cap \bar{S}_2$
- $S_1 \bar{\cap} S_2 = \bar{S}_1 \cup \bar{S}_2$

1.1.2 Functions

Function $f : D \rightarrow R$ 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

1.1.3 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 $x \equiv z$

1.1.4 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 $E = \{(v_1 v_3), (v_3, v_1), (v_3, v_2), (v_3, v_3)\}$.

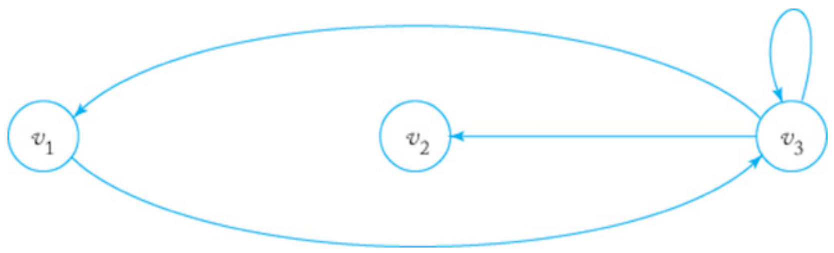


Figure 1: **Linz Fig. 1.1: Diagram of a Digraph**

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.

1.1.5 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.

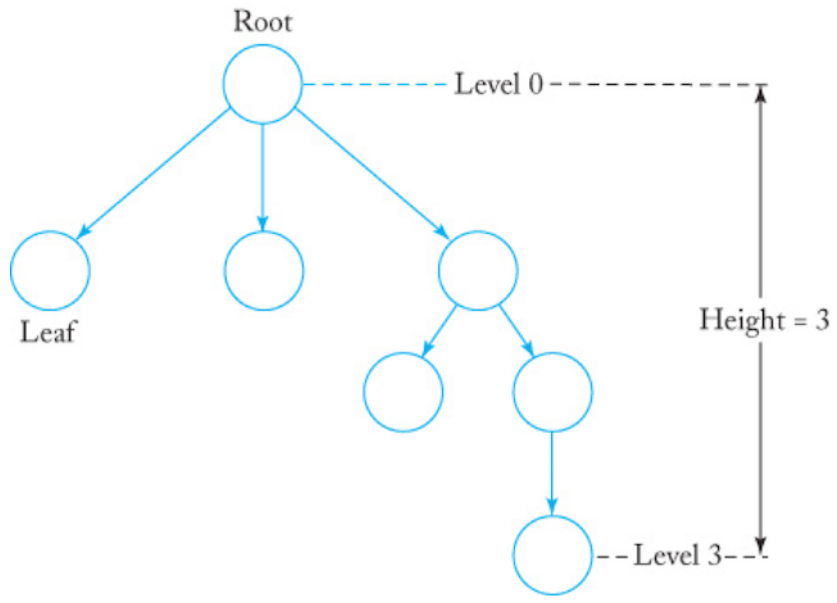


Figure 2: **Linz Fig. 1.2: A Tree**

1.1.6 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.

1.2 Three Basic Concepts

Three fundamental ideas are the major themes of this course:

1. languages
2. grammars
3. automata

1.2.1 Languages

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

1.2.1.1 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, $\lambda w = w\lambda = w$.

1.2.1.2 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)

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

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

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.

1.2.1.3 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^*).

1.2.1.4 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*.

1.2.1.5 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 $L^+ = L^1 \cup L^2 \cup \cdots$.

1.2.1.6 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.

1.2.2 Grammars

1.2.2.1 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 $x \to y$ 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 $w = uxv$:

- $x \to y$ 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.

1.2.2.2 Linz Example 1.11 (Grammar)

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

- $S \rightarrow aSb$
- $S \rightarrow \lambda$

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 $S \rightarrow aSb$, 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 $S \rightarrow \lambda$, 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 $S \rightarrow \lambda$ 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.

1.2.2.3 Linz Example 1.12: Finding a Grammar for a Language

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

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

A slightly different grammar might introduce nonterminal A as follows:

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

1.2.2.4 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.

1.2.2.5 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

$$
\begin{array}{l}
S \rightarrow SS \\
S \rightarrow \lambda \\
S \rightarrow aSb \\
S \rightarrow bSa
\end{array}
$$

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 $w \in L(G)$.

1.2.3 Automata

An *automaton* is an abstract model of a compute

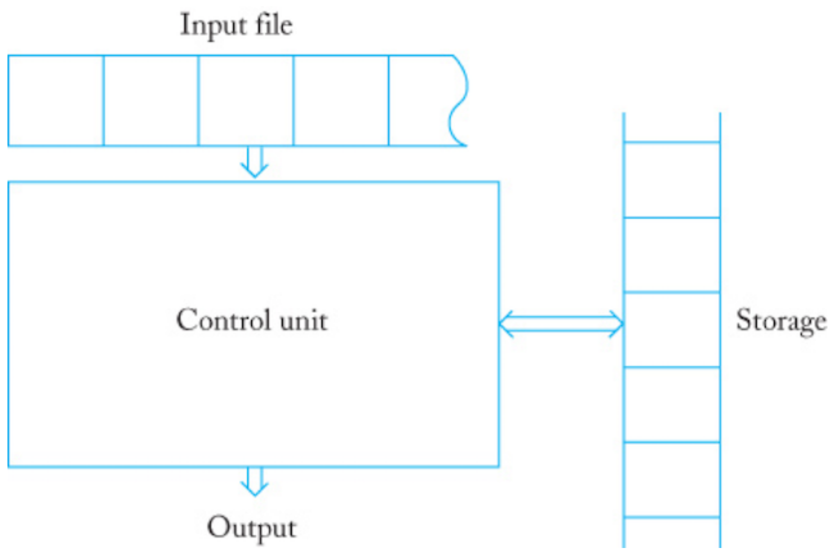


Figure 3: **Linz Fig. 1.4: Schematic Representation of a General Automaton**

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

1.3 Applications

1.3.1 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.

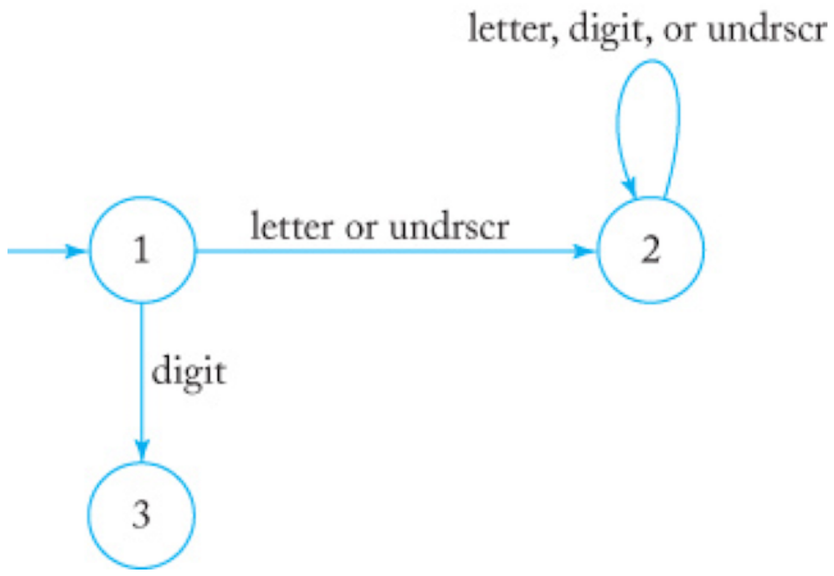


Figure 4: **Linz Fig. 1.7: Automaton to Accept C Identifiers**

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

1.3.2 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.

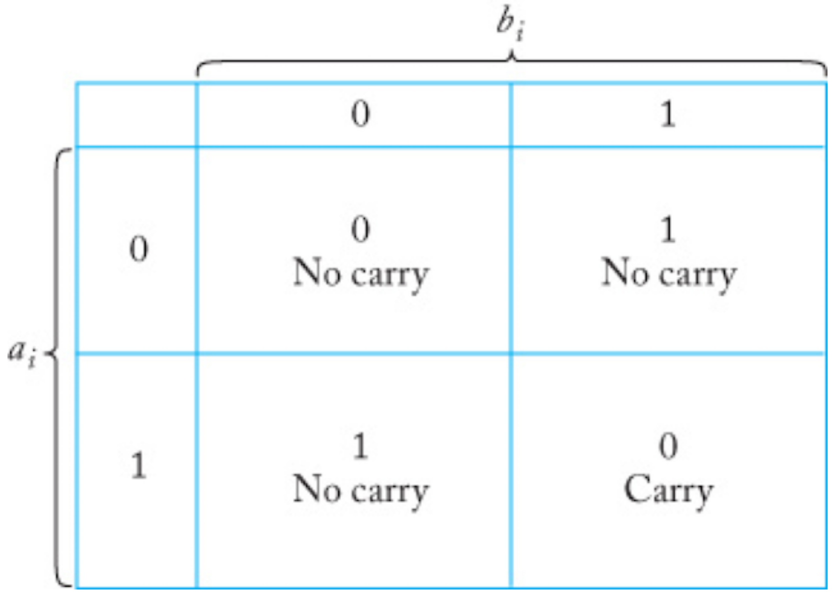
A *block diagram* for the machine is shown in Linz Figure 1.8.

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.

1.4 References

[1] Peter Linz. 2011. *Formal languages and automata* (Fifth ed.). Jones & Bartlett, Burlington, Massachusetts, USA.



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**

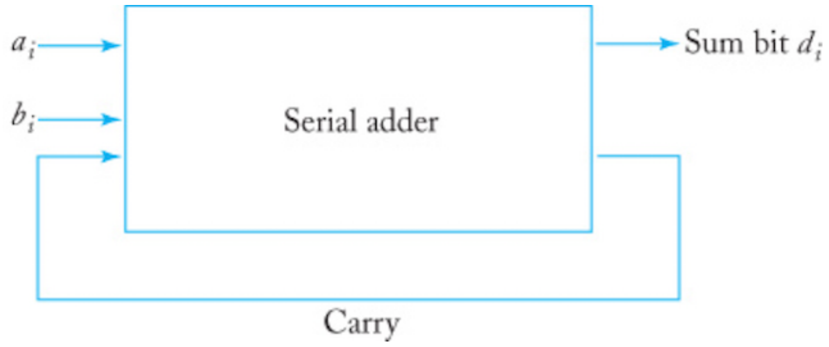


Figure 6: **Linz Fig. 1.8: Binary Adder Block Diagram**

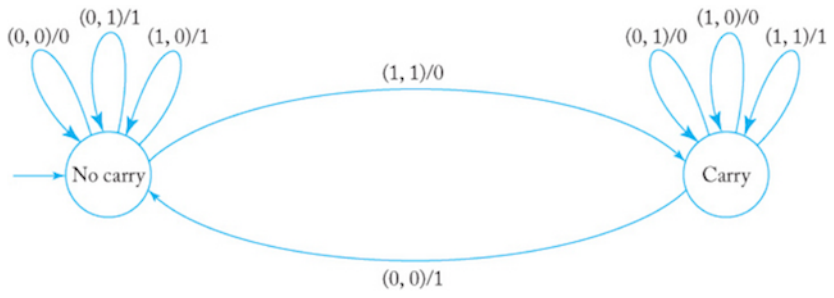


Figure 7: **Linz Fig. 1.9: Binary Adder Transducer Automaton**