B.3 Test tube programming language

¶1. **Test Tube Programming Language (TTPL):** These ideas can be extended to a *Test Tube Programming Language (TTPL).*

¶2. Developed in the mid 90s by Lipton and Adleman.

B.3.a Basic Operations

¶1. DNA algorithms operate on "test tubes," which are multi-sets of strings over $\Sigma = \{\text{A, C, T, G}\}$.

¶2. There are four basic operations (all implementable):

¶3. **Extract (or separate):** There are two complementary *extraction* (or *separation*) operations.
Given a test tube t and a string w, $+(t, w)$ returns all strings in t that have w as a subsequence:

$$+(t,w) \stackrel{\text{def}}{=} \{s \in t \mid \exists u, v \in \Sigma^* : s = uwv\}.$$

Likewise, $-(t, w)$ returns a test tube of all the remaining strings:

$$-(t,w) \stackrel{\text{def}}{=} t \; - \; +(t,w) \quad \text{(multi-set difference)}.$$

¶4. **Merge:** The *merge* operation combines several test tubes into one test tube:

$$\cup(t_1, t_2, \ldots, t_n) \stackrel{\text{def}}{=} t_1 \cup t_2 \cup \cdots \cup t_n.$$

¶5. **Detect:** The detect operation determines if any DNA strings remain in a test tube:

$$\text{detect}(t) \stackrel{\text{def}}{=} \begin{cases} \textbf{true}, & \text{if } t \neq \emptyset \\ \textbf{false}, & \text{otherwise} \end{cases}.$$

¶6. **Amplify:** Given a test tube t, the *amplify* operation produces two copies of it: $t', t'' \leftarrow \text{amplify}(t)$.

¶7. **Restricted model:** Amplification is a problematic operation, which depends on the special properties of DNA and RNA.
Also it may be error prone.
Therefore it is useful to consider a *restricted model* of DNA computing that avoids or minimizes the use of amplification.

¶8. The following additional operations have been proposed:

¶9. **Length-separate:** Produces a test tube containing all the strands less than a specified length:

$$(t, \leq n) \stackrel{\text{def}}{=} \{s \in t \mid |s| \leq n\}.$$

¶10. **Position-separate:** There are two *position-separation* operations, one that selects for strings that begin with a given sequence, and one for sequences that end with it:

$$\begin{aligned} B(t, w) &\stackrel{\text{def}}{=} \{s \in t \mid \exists v \in \Sigma^* : s = wv\}, \\ E(t, w) &\stackrel{\text{def}}{=} \{s \in t \mid \exists u \in \Sigma^* : s = uw\}. \end{aligned}$$

B.3.b Examples

¶1. **AllC:** The following example algorithm detects if there are any sequences that contain only C:

```
procedure [out] = AllC(t, A, T, G)
  t ← −(t, A)
  t ← −(t, T)
  t ← −(t, G)
  out ← detect (t)
end procedure
```

¶2. **HPP:** Adelman's solution of the HPP can be expressed in TTPL:

```
procedure [out] = HPP(t, vin, vout)
  t ← B(t, vin)                  //begin with vin
  t ← E(t, vout)                 //end with vout
  t ← (t, ≤ 140)                 //correct length
  for i=1 to 5 do
    t ← +(t, s[i])               //contain vertex i
  end for
  out ← detect(t)                //any HP left?
end procedure
```

¶3. **SAT:** Programming Lipton's solution to SAT requires another primitive operation, which extracts all sequences for which the jth bit is $a \in \mathbf{2}$: $E(t, j, a)$.
Recall that these are represented by the sequences x_j and x'_j. Therefore:

$$\begin{aligned} E(t, j, 1) &= +(t, x_j), \\ E(t, j, 0) &= +(t, x'_j). \end{aligned}$$

¶4.
```
procedure [out] = SAT(t)
  for k = 1 to m do                  // for each clause
    for i = 1 to n do                // for each literal
      if C[k][i] = x_j               // i-th literal in clause k
        then t[i] ← E(t,j,1)
        else t[i] ← E(t,j,0)
      end if
    end for
    t ← ∪(t[1], t[2], ..., t[n]) // solutions for clauses 1..k
  end for
  out ← detect(t)
end procedure
```