1 Tree Grammars

1) Derivation tree for \( f(g(h(a), b), h(g(h(a), h(a)))) \):

\[
S \rightarrow f(g(A, B), A)
\]

\[
A \rightarrow h(a) \quad B \rightarrow b \quad A \rightarrow h(g(A, A))
\]

\[
A \rightarrow h(a) \quad A \rightarrow h(a)
\]

2) Another derivation tree for \( f(g(h(a), b), h(g(h(a), h(a)))) \):

\[
S \rightarrow f(C, h(B))
\]

\[
C \rightarrow g(h(a), b) \quad B \rightarrow g(h(D), A)
\]

\[
D \rightarrow a \quad A \rightarrow h(a)
\]
2 Tree Automata

a) DTFA $A = (\{q_t, q_f\}, \Sigma, \delta, \{q_t\})$ with $\Sigma = \{\text{true, false, and, or, xor, not}\}$, where $\rho(\text{true}) = \rho(\text{false}) = 0$, $\rho(\text{not}) = 1$ and $\rho(\text{and}) = \rho(\text{or}) = \rho(\text{xor}) = 2$.

The transition relation $\delta$ is the set of transitions:

$$
\delta = \{(q_f, \text{false}, \varepsilon),
(q_t, \text{true}, \varepsilon),
(q_f, \text{not}, q_t),
(q_t, \text{not}, q_f),
(q_f, \text{and}, (q_f, q_f)),
(q_f, \text{and}, (q_t, q_f)),
(q_f, \text{and}, (q_f, q_t)),
(q_t, \text{and}, (q_t, q_t)),
(q_f, \text{or}, (q_f, q_f)),
(q_t, \text{or}, (q_t, q_f)),
(q_t, \text{or}, (q_t, q_t)),
(q_f, \text{xor}, (q_f, q_f)),
(q_t, \text{xor}, (q_t, q_f)),
(q_t, \text{xor}, (q_t, q_t)),
(q_f, \text{xor}, (q_t, q_f)),
(q_f, \text{xor}, (q_t, q_t))\}
$$

The correctness is obvious (all operators behave as expected). If the final state (the state of the root node) is $q_t$ the expression evaluates to $\text{true}$ and the DTFA $A$ accepts the corresponding input tree.

b) A finite tree automaton is top down deterministic, if there is at most one $(q, a, q_1, \ldots, q_n) \in \delta$ for all $q$ and $a$ (see Page 30). This implies that the automaton must know the result of a subexpression before it can evaluate it. Since the result of a logical formual such as xor depends on both subexpression, these subexpressions must be evaluated first. A deterministic top-down automaton, however, tries to evaluate the expressions the other way around.

3 Code Generation

a) To achieve minimum register needs the result of an operation is stored in a temporary memory cell, if there is need to do so. This results in a register need of 1 and the following functions:

Node labels:

$$
\text{need(leaf}(a)) = 0,
\text{need(op}(a, b)) = 1
$$

Code generation:

```plaintext
static temp = 0;

gen (leaf(a, 0)) {
    emit ("R[0] = M[a]");  
}
```
gen (op(leaf(a, 0), leaf(b, 0))) {
    emit ("R[0] = M[a] op M[b]";)
}

gen (op(leaf(a, 0), I)) {
    gen (I);
    emit ("R[0] = M[a] op R[0]";)
}

gen (op(I, leaf(a, 0))) {
    gen (I);
    emit ("R[0] = M[a] op R[0]";)
}

gen (op(I, J)) {
    gen (I);
    emit ("M[temp] = R[0]";)
    ++temp;
    gen (J);
    --temp;
    emit ("R[0] = M[temp] op R[0]";)
}

b) For the given trees, the algorithm computes:

1) Node labels:

```
+ , 1
   /
  * , 1
 |   /
a, 0 b, 0 a, 0 / , 1
   |   |
   c, 0 d, 0
```

Code:

```c
/* gen(+(*(a, b), /(a, -(c, d)))) */
/* gen(*(a, b)) */
R[0] = M[a] + M[b]
M[t0] = R[0]

/* gen(/(a, -(c, d))) */
/* gen(-(c, d)) */
R[0] = M[c] - M[d]
R[0] = M[a] / R[0]

/* final result */
R[0] = M[t0] + R[0]
```
2) Node Labels:

```
/* gen(+(*(+(e,d), /(a,b)), /(*(e,f), -(c,d)))) */
/* gen(+(*(e,d), /(a,b))) */
/* gen(*+(e,d)) */
R[0] = M[e] + M[d]
M[t0] = R[0]

/* gen/(a,b)) */
R[0] = M[a]/M[b]

/* result */
R[0] = M[t0] * R[0]
M[t0] = R[0]

/* gen/(*(e,f), -(c,d))) */
/* gen(*(e,f)) */
R[0] = M[e] * M[f]
M[t1] = R[0]

/* gen(-(c,d)) */
R[0] = M[c] - M[d]

/* result */
R[0] = M[t1] / R[0]

/* final result */
R[0] = M[t0] + R[0]
```