optimization of if-then-else chains

original C code

```c
if(!a && !b) h();
else if(!a) g();
else f();
```

optimized C code

```c
if(a) f();
else if(b) g();
else h();
```

How to check that these two versions are equivalent?
1. represent procedures as independent boolean variables

original :=

\[
\begin{align*}
\text{if } \neg a \land \neg b \text{ then } h \\
\text{else if } \neg a \text{ then } g \\
\text{else } f
\end{align*}
\]

optimized :=

\[
\begin{align*}
\text{if } a \text{ then } f \\
\text{else if } b \text{ then } g \\
\text{else } h
\end{align*}
\]

2. compile if-then-else chains into boolean formulae

\[
\text{compile}(\text{if } x \text{ then } y \text{ else } z) \equiv (x \land y) \lor (\neg x \land z)
\]

3. check equivalence of boolean formulae

\[
\text{compile}(\text{original}) \iff \text{compile}(\text{optimized})
\]
original  ≡  \text{if } \neg a \land \neg b \text{ then } h \text{ else if } \neg a \text{ then } g \text{ else } f
\equiv (\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land \text{if } \neg a \text{ then } g \text{ else } f
\equiv (\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f)

optimized  ≡  \text{if } a \text{ then } f \text{ else if } b \text{ then } g \text{ else } h
\equiv a \land f \lor \neg a \land \text{if } b \text{ then } g \text{ else } h
\equiv a \land f \lor \neg a \land (b \land g \lor \neg b \land h)

(\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \iff a \land f \lor \neg a \land (b \land g \lor \neg b \land h)
How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

Is there an assignment to $a, b, f, g, h,$
which results in different evaluations of original and optimized?

or equivalently:

Is the boolean formula $\text{compile}(\text{original}) \not\leftrightarrow \text{compile}(\text{optimized})$ satisfiable?

such an assignment would provide an easy to understand counterexample

**Note:** by concentrating on counterexamples we moved from Co-NP to NP
(this is just a theoretical note and not really important for applications)
SAT Example: Circuit Equivalence Checking

\[ b \lor a \land c \]

\[ (a \lor b) \land (b \lor c) \]

equivalent?

\[ b \lor a \land c \iff (a \lor b) \land (b \lor c) \]
**SAT (Satisfiability)** the classical NP complete Problem:

Given a propositional formula $f$ over $n$ propositional variables $V = \{x, y, \ldots\}$.

Is there an assignment $\sigma : V \rightarrow \{0, 1\}$ with $\sigma(f) = 1$?

**SAT belongs to NP**

There is a *non-deterministic* Turing-machine deciding SAT in polynomial time:

*guess* the assignment $\sigma$ (linear in $n$), calculate $\sigma(f)$ (linear in $|f|$)

**Note:** on a *real* (deterministic) computer this would still require $2^n$ time

**SAT is complete for NP** (see complexity / theory class)

**Implications for us:**

general SAT algorithms are probably exponential in time (unless NP = P)
Conjunctive Normal Form

Definition

A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses

\[ C_1 \land C_2 \land \ldots \land C_n \]

Each clause \( C \) is a disjunction of literals

\[ C = L_1 \lor \ldots \lor L_m \]

And each literal is either a plain variable \( x \) or a negated variable \( \overline{x} \).

Example  \((a \lor b \lor c) \land (\overline{a} \lor \overline{b}) \land (\overline{a} \lor \overline{c})\)

Note 1: Two notions for negation: in \( \overline{x} \) and \( \neg \) as in \( \neg x \) for denoting negation.

Note 2: The original SAT problem is actually formulated for CNF.

Note 3: SAT solvers mostly also expect CNF as input.
**Assumption:** we only have conjunction, disjunction and negation as operators.

A formula is in Negation Normal Form (NNF), if negations only occur in front of variables.

⇒ all *internal* nodes in the formula tree are either ANDs or ORs.

Linear algorithms for generating NNF from an arbitrary formula.

Often NNF generations includes elimination of other non-monotonic operators:

\[
\text{NNF of } f \leftrightarrow g \text{ is NNF of } f \land g \lor \overline{f} \land \overline{g}
\]

In this case the result can be exponentially larger (see parity example later).
NNF Algorithm

Formula

formula2nnf (Formula f, Boole sign)
{
    if (is_variable (f))
        return sign ? new_not_node (f) : f;
    if (op (f) == AND || op (f) == OR)
    {
        l = formula2nnf (left_child (f), sign);
        r = formula2nnf (right_child (f), sign);
        flipped_op = (op (f) == AND) ? OR : AND;
        return new_node (sign ? flipped_op : op (f), l, r);
    }
    else
    {
        assert (op (f) == NOT);
        return formula2nnf (child (f), !sign);
    }
}
Simple Translation of Formula into CNF

Formula

formula2cnf_aux (Formula f)
{
    if (is_cnf (f))
        return f;
    if (op (f) == AND)
    {
        l = formula2cnf_aux (left_child (f));
        r = formula2cnf_aux (right_child (f));
        return new_node (AND, l, r);
    }
    else
    {
        assert (op (f) == OR);
        l = formula2cnf_aux (left_child (f));
        r = formula2cnf_aux (right_child (f));
        return merge_cnf (l, r);
    }
}
Merging two CNFs

Formula
formula2cnf (Formula f)
{
    return formula2cnf_aux (formula2nnf (f, 0));
}

Formula
merge_cnf (Formula f, Formula g)
{
    res = new_constant_node (TRUE);
    for (c = first_clause (f); c; c = next_clause (f, c))
        for (d = first_clause (g); d; d = next_clause (g, d))
            res = new_node (AND, res, new_node (OR, c, d));
    return res;
}
Why are Sharing / Circuits / DAGs important?

DAG may be exponentially more succinct than expanded Tree

Examples:  adder circuit, parity, mutual exclusion
Boole
parity (Boole a, Boole b, Boole c, Boole d, Boole e, 
Boole f, Boole g, Boole h, Boole i, Boole j)
{
  tmp0 = b ? !a : a;
  tmp1 = c ? !tmp0 : tmp0;
  tmp2 = d ? !tmp1 : tmp1;
  tmp3 = e ? !tmp2 : tmp2;
  tmp4 = f ? !tmp3 : tmp3;
  tmp5 = g ? !tmp4 : tmp4;
  tmp6 = h ? !tmp5 : tmp5;
  tmp7 = i ? !tmp6 : tmp6;
  \textbf{return} j ? !tmp7 : tmp7;
}

Eliminate the $\text{tmp}$... variables through substitution.

What is the size of the DAG vs the Tree representation?
• through caching of results in algorithms operating on formulas
  (examples: substitution algorithm, generation of NNF for non-monotonic ops)

• when modeling a system: variables are introduced for subformulae
  (then these variables are used multiple times in the toplevel formula)

• structural hashing: detects structural identical subformulae
  (see Signed And Graphs later)

• equivalence extraction: eg. BDD sweeping, Stålmarcks Method
  (we will look at both techniques in more detail later)
Example of Tseitin Transformation: Circuit to CNF

\[ o \land (x \leftrightarrow a \land c) \land (y \leftrightarrow b \lor x) \land (u \leftrightarrow a \lor b) \land (v \leftrightarrow b \lor c) \land (w \leftrightarrow u \land v) \land (o \leftrightarrow y \oplus w) \]

\[ o \land (x \rightarrow a) \land (x \rightarrow c) \land (x \leftarrow a \land c) \land \ldots \]

\[ o \land (\bar{x} \lor a) \land (\bar{x} \lor c) \land (x \lor \bar{a} \lor \bar{c}) \land \ldots \]
1. for each non input circuit signal $s$ generate a new variable $x_s$

2. for each gate produce complete input / output constraints as clauses

3. collect all constraints in a big conjunction

the transformation is *satisfiability equivalent*: the result is satisfiable iff and only the original formula is satisfiable

not equivalent in the classical sense to original formula: it has new variables

extract satisfying assignment for original formula, from one of the result (just project satisfying assignment onto the original variables)
Negation: \[ x \leftrightarrow \overline{y} \iff (x \rightarrow \overline{y}) \land (\overline{y} \rightarrow x) \]
\[ \iff (\overline{x} \lor \overline{y}) \land (y \lor x) \]

Disjunction: \[ x \leftrightarrow (y \lor z) \iff (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \]
\[ \iff (\overline{y} \lor x) \land (\overline{z} \lor x) \land (\overline{x} \lor y \lor z) \]

Conjunction: \[ x \leftrightarrow (y \land z) \iff (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \]
\[ \iff (\overline{x} \lor y) \land (\overline{x} \lor z) \land ((y \land z) \lor x) \]
\[ \iff (\overline{x} \lor y) \land (\overline{x} \lor z) \land (\overline{y} \lor \overline{z} \lor x) \]

Equivalence: \[ x \leftrightarrow (y \leftrightarrow z) \iff (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y))) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \lor (\overline{y} \land \overline{z})) \rightarrow x \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \lor (\overline{y} \lor \overline{z})) \rightarrow x \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (\overline{y} \lor \overline{z} \lor x) \land (y \lor z \lor x) \]
\[ \iff (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (\overline{y} \lor \overline{z} \lor x) \land (y \lor z \lor x) \]
Optimizations for Tseitin Transformation

- goal is smaller CNF (less variables, less clauses)

- extract multi argument operands (removes variables for intermediate nodes)

- half of AND, OR node constraints may be removed for *unnegated* nodes
  a node occurs negated if it has an ancestor which is a negation
  half of the constraints determine parent assignment from child assignment
  those are unnecessary if node is not used negated

- those have to be carefully applied to DAG structure
  (compare with the implementation of the BMC tool from CMU)