optimization of if-then-else chains

original C code

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

$\Downarrow$

```
if(!a) {
  if(!b) h();
  else g();
} else f();
```

$\Rightarrow$

optimized C code

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

$\Uparrow$

```
if(a) f();
else {
  if(!b) h();
  else g(); }
```

How to check that these two versions are equivalent?

SAT Example cont.

1. represent procedures as *independent* boolean variables

$$\mathit{original} := \quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h \quad \textbf{else if } \neg a \textbf{ then } g \quad \textbf{else } f$$

$$\mathit{optimized} := \quad \textbf{if } a \textbf{ then } f \quad \textbf{else if } b \textbf{ then } g \quad \textbf{else } h$$

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

$$\mathrm{compile}(\textbf{if } x \textbf{ then } y \textbf{ else } z) \quad \equiv \quad (x \wedge y) \vee (\neg x \wedge z)$$

3. check equivalence of boolean formulae

$$\mathrm{compile}(\mathit{original}) \quad \Leftrightarrow \quad \mathrm{compile}(\mathit{optimized})$$

$$
\begin{aligned}
\textit{original} \;&\equiv\; \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv\; (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv\; (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f)
\end{aligned}
$$

$$
\begin{aligned}
\textit{optimized} \;&\equiv\; \textbf{if } a \textbf{ then } f \textbf{ else } \textbf{ if } b \textbf{ then } g \textbf{ else } h \\
&\equiv\; a \wedge f \vee \neg a \wedge \textbf{ if } b \textbf{ then } g \textbf{ else } h \\
&\equiv\; a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h)
\end{aligned}
$$

$$
(\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f) \quad\Leftrightarrow\quad a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h)
$$

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:

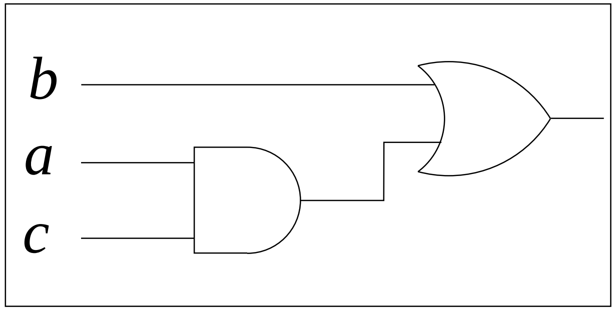
Is the boolean formula $\boxed{\text{compile}(\mathit{original}) \not\leftrightarrow \text{compile}(\mathit{optimized})}$ satisfiable?

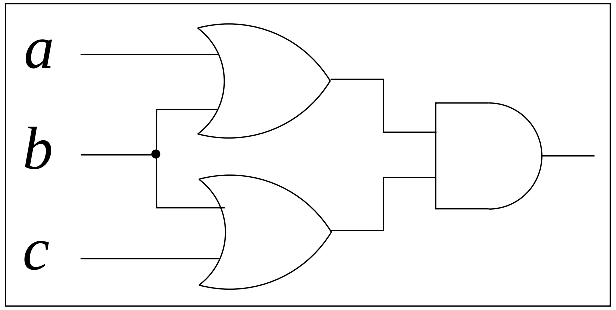
such an assignment would provide an easy to understand counterexample

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





$$b \vee a \wedge c \qquad\qquad (a \vee b) \wedge (b \vee c)$$

equivalent?

$$b \vee a \wedge c \quad \Leftrightarrow \quad (a \vee b) \wedge (b \vee c)$$

SAT (Satisfiability) the classical NP complete Problem:

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

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

SAT belongs to NP

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

guess the assignment σ (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

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

each clause C is a disjunction of literals

$$C = L_1 \vee \ldots \vee L_m$$

and each literal is either a plain variable x or a negated variable $\overline{x}$.

Example $(a \vee b \vee c) \wedge (\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee \overline{c})$

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

$\Rightarrow$ 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:

$$\text{NNF of} \quad f \leftrightarrow g \quad \text{is NNF of} \quad f \wedge g \vee \overline{f} \wedge \overline{g}$$

in this case the result can be exponentially larger (see parity example later).

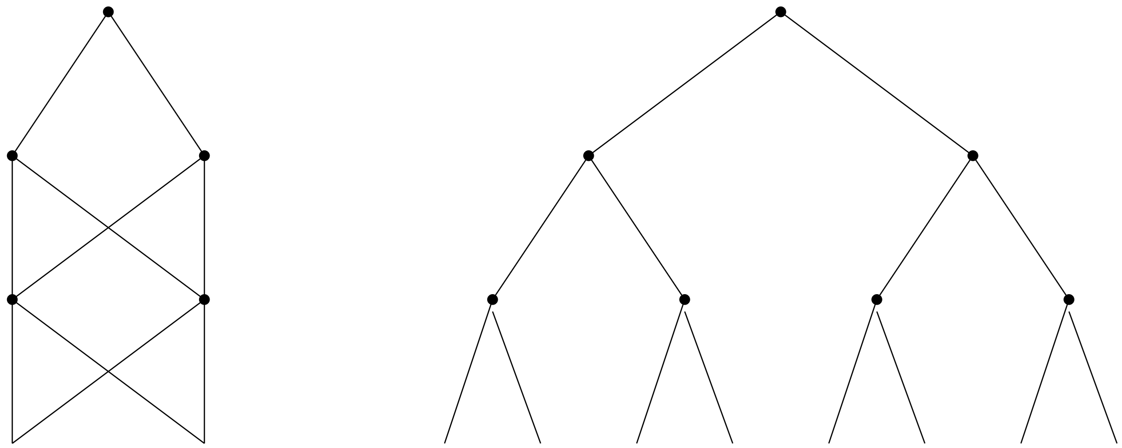
```
Formula
formula2nnf (Formula f, Boole sign)
{
  if (is_variable (f))
    return sign ? new_not_node (f) : f;
  if (op (f) == AND || op (f) == OR)
    {
      l = formula2nnf (left_child (f), sign);
      r = formula2nnf (right_child (f), sign);
      flipped_op = (op (f) == AND) ? OR : AND;
    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

```
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);
    }
}
```

```
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;
  return j ? !tmp7 : tmp7;
}
```

Eliminiate the `tmp...` variables through substitution.

What is the size of the DAG vs the Tree representation?

How to detect Sharing

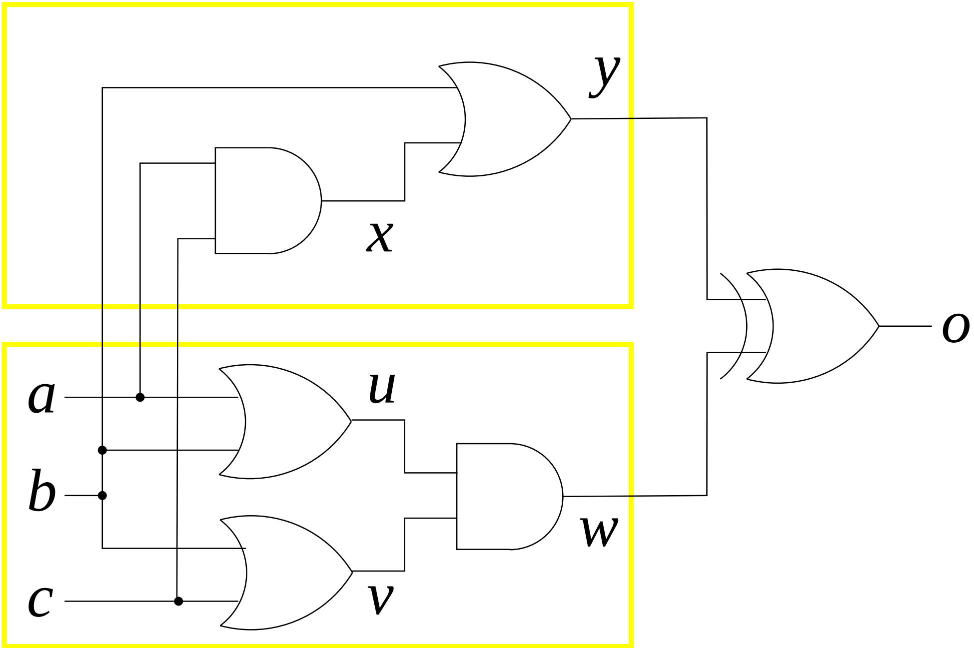
- through caching of results in algorithms operating on formulas
  (examples: substitution algorithm, generation of NNF for non-monotonic ops)

- 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)

CNF



$$
\begin{array}{l}
o \wedge \\
(x \leftrightarrow a \wedge c) \wedge \\
(y \leftrightarrow b \vee x) \wedge \\
(u \leftrightarrow a \vee b) \wedge \\
(v \leftrightarrow b \vee c) \wedge \\
(w \leftrightarrow u \wedge v) \wedge \\
(o \leftrightarrow y \oplus w)
\end{array}
$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$

Algorithmic Description of Tseitin Transformation

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:
$$x \leftrightarrow \overline{y} \;\Leftrightarrow\; (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x)$$
$$\Leftrightarrow\; (\overline{x} \vee \overline{y}) \wedge (y \vee x)$$

Disjunction:
$$x \leftrightarrow (y \vee z) \;\Leftrightarrow\; (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z))$$
$$\Leftrightarrow\; (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z)$$

Conjunction:
$$x \leftrightarrow (y \wedge z) \;\Leftrightarrow\; (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x)$$
$$\Leftrightarrow\; (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x)$$
$$\Leftrightarrow\; (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x)$$

Equivalence:
$$x \leftrightarrow (y \leftrightarrow z) \;\Leftrightarrow\; (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x)$$
$$\Leftrightarrow\; (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y))) \wedge ((y \leftrightarrow z) \rightarrow x)$$
$$\Leftrightarrow\; (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x)$$
$$\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x)$$
$$\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x)$$
$$\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x)$$
$$\Leftrightarrow\; (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x)$$

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)