[Collatz in Dafny]

Hardware & Software Verification

John Wickerson & Pete Harrod

Lecture 10: SAT and SMT solving

Automatic proof

Automatic proof

- We often rely on automatic provers:

Automatic proof

- We often rely on automatic provers:
  - e.g. in Dafny, to show that `invariant P` is preserved,

Automatic proof

- We often rely on automatic provers:
  - e.g. in Dafny, to show that `invariant P` is preserved,
  - e.g. in Isabelle methods like `by auto`.

Automatic proof

- We often rely on automatic provers:
  - e.g. in Dafny, to show that `invariant P` is preserved,
  - e.g. in Isabelle methods like `by auto`.
- How do these automatic provers work?

SAT queries

- Simple case: proofs about Boolean statements.

SAT queries

- Simple case: proofs about Boolean statements.
  - $f = ((A \wedge \neg B \wedge C) \Longrightarrow (C \vee (B \wedge A)))$

SAT queries

- Simple case: proofs about Boolean statements.
  - $f = (\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$

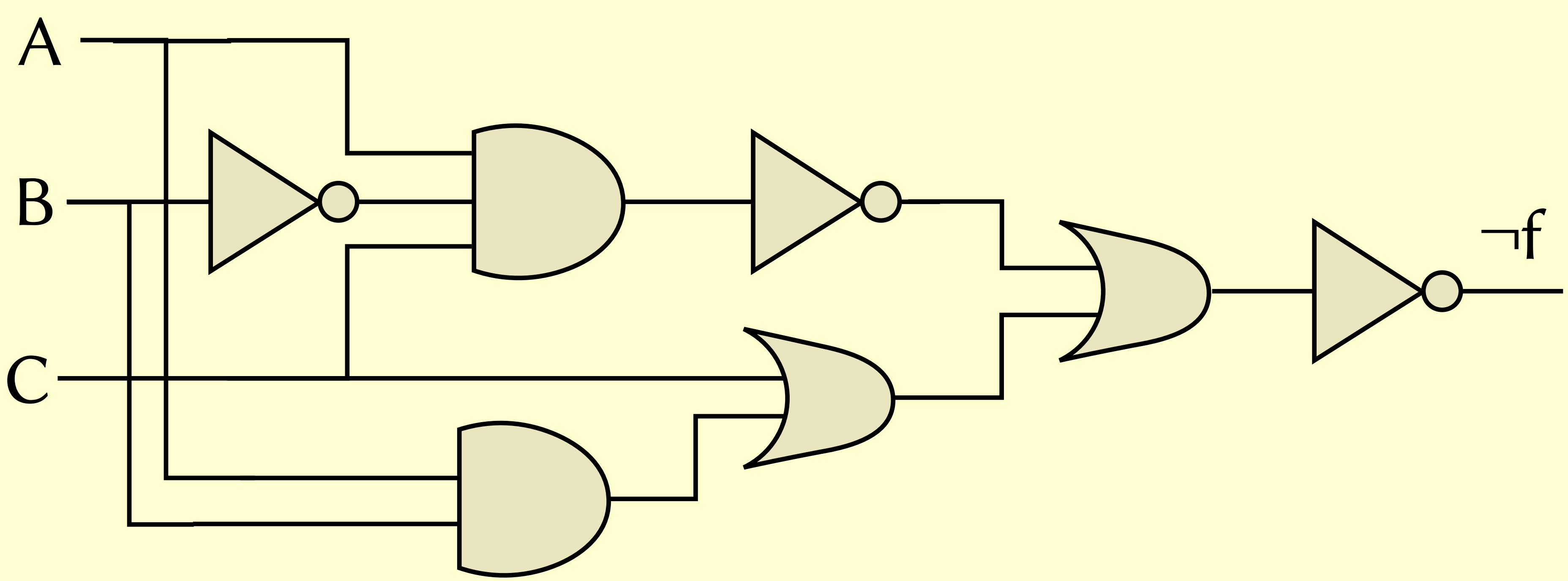
SAT queries

- Simple case: proofs about Boolean statements.
  - $\neg f \;=\; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$

SAT queries

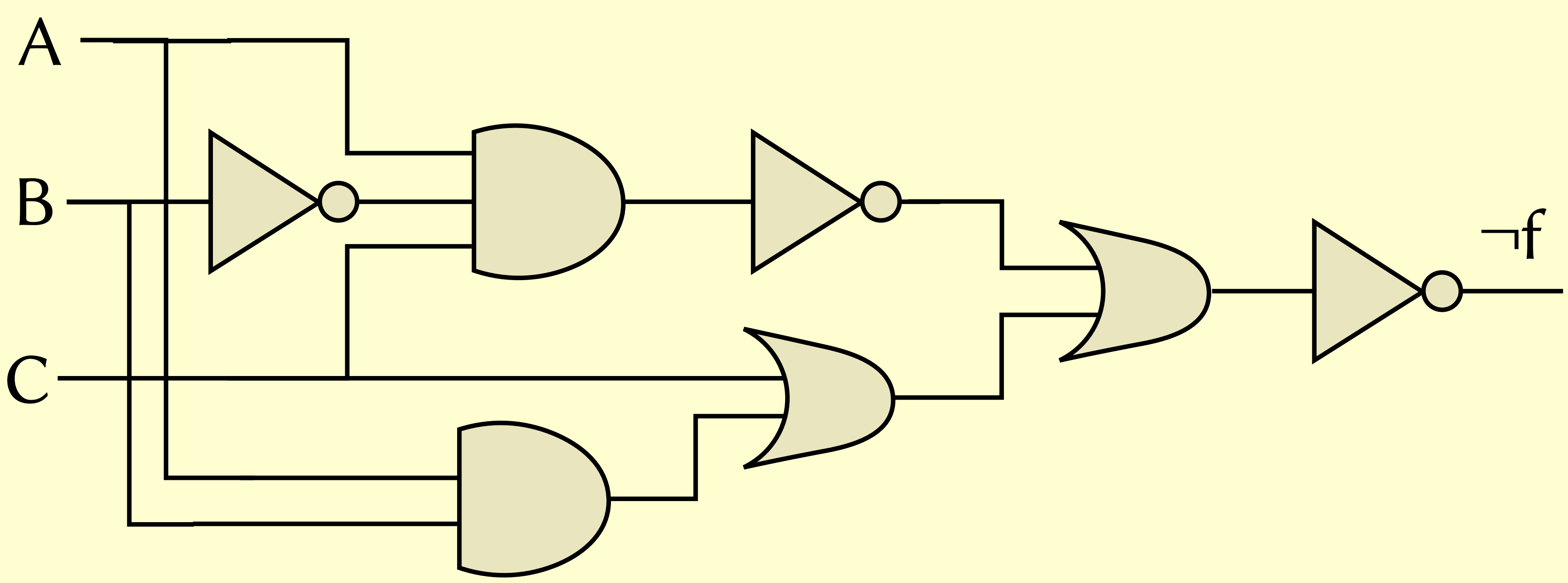
- Simple case: proofs about Boolean statements.

  - $\neg f \;=\; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$



SAT queries

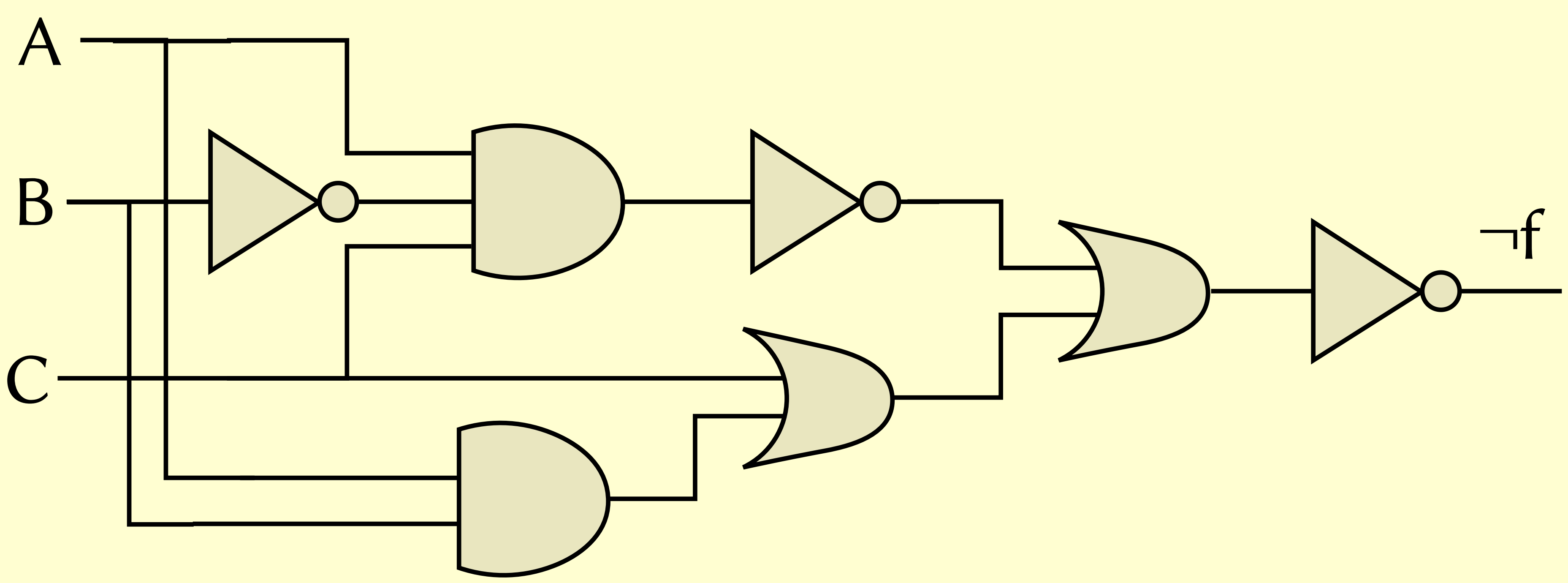
- Simple case: proofs about Boolean statements.
  - $\neg f \;=\; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$



A	B	C	¬f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

SAT queries

- Simple case: proofs about Boolean statements.
  - $\neg f \; = \; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$

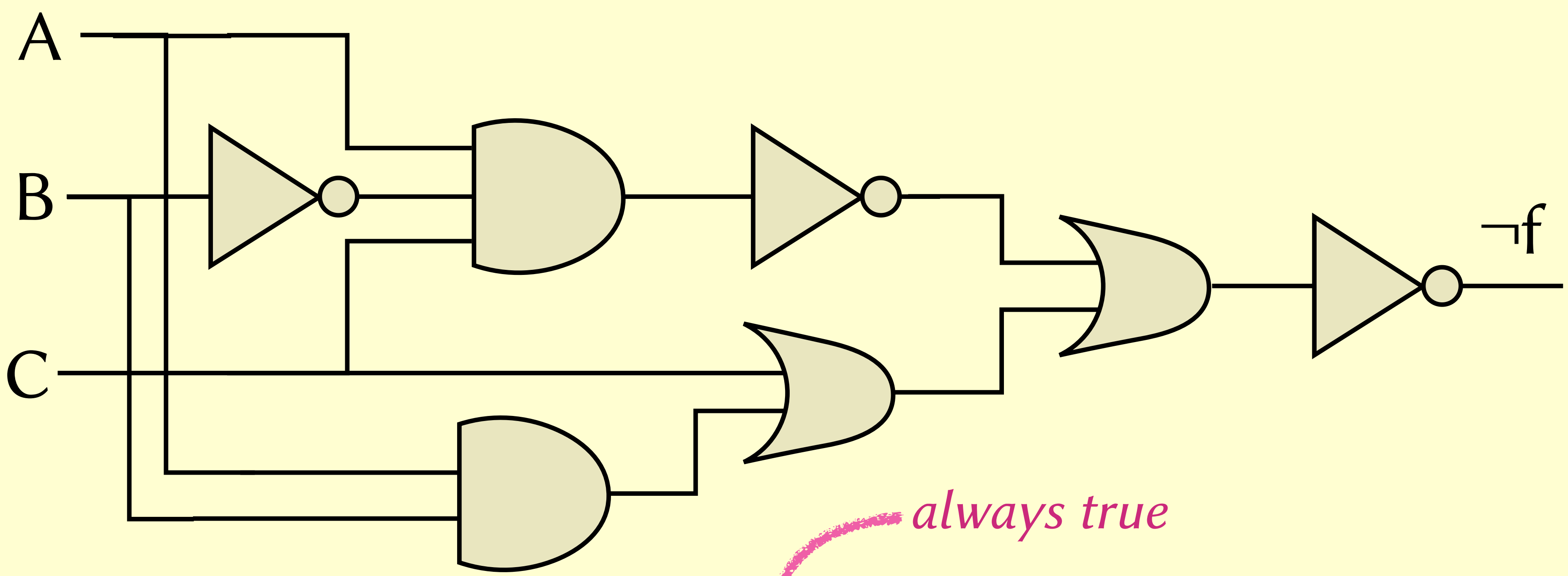


A	B	C	¬f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

A formula can be VALID, SATISFIABLE, UNSATISFIABLE, or INVALID.

SAT queries

- Simple case: proofs about Boolean statements.
  - $\neg f \;=\; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$



A	B	C	¬f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

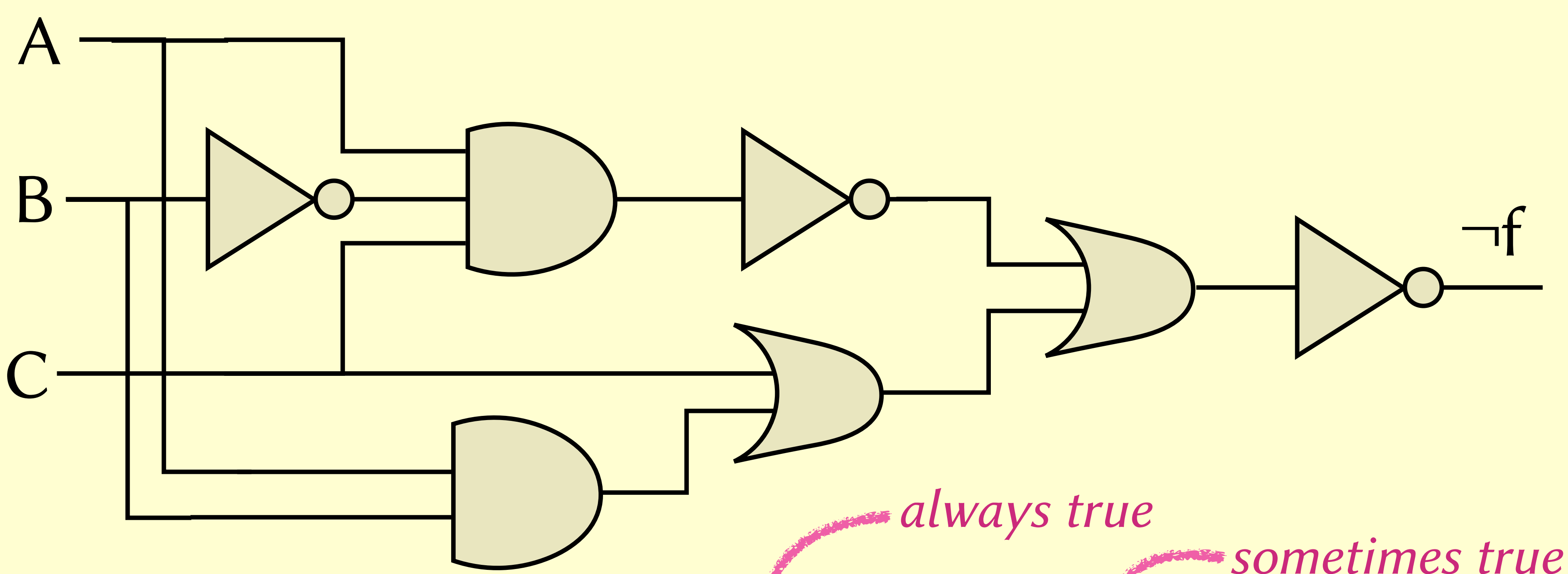
always true

A formula can be VALID, SATISFIABLE, UNSATISFIABLE, or INVALID.

SAT queries

- Simple case: proofs about Boolean statements.

  - $\neg f \;=\; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$



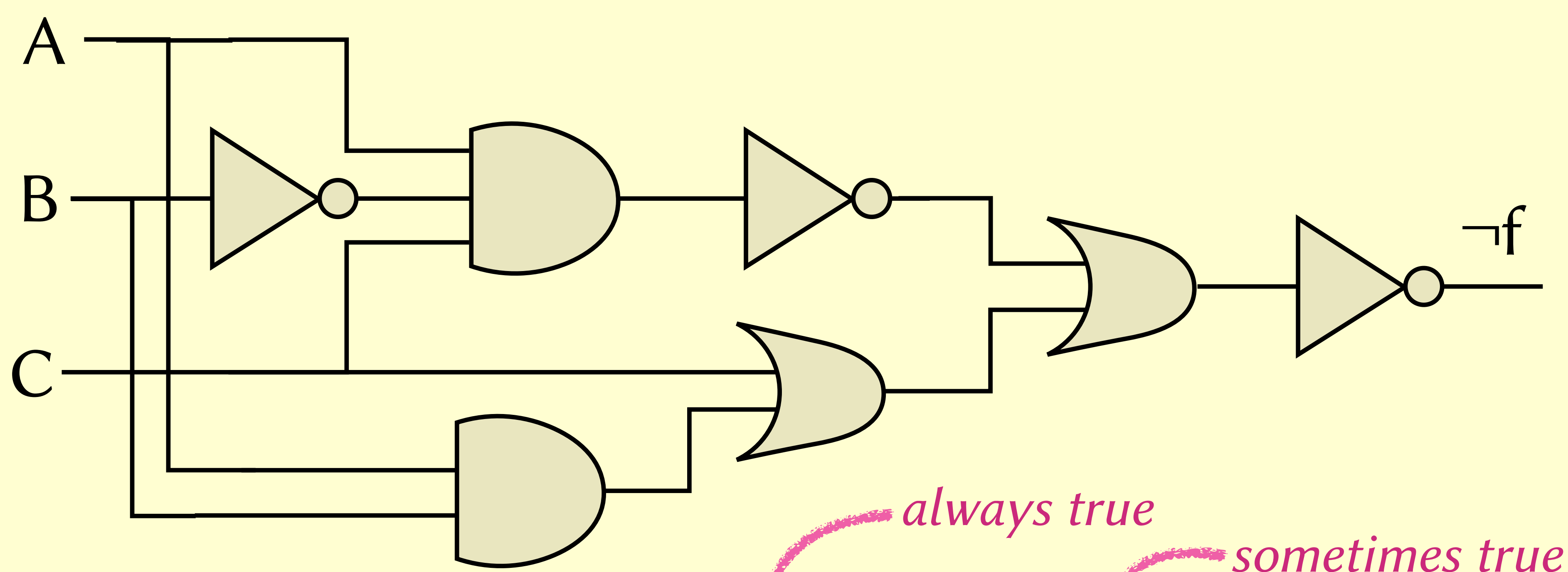
A	B	C	¬f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

always true *sometimes true*

A formula can be VALID, SATISFIABLE, UNSATISFIABLE, or INVALID.

SAT queries

- Simple case: proofs about Boolean statements.
  - $\neg f = \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$



A	B	C	¬f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

always true *sometimes true*

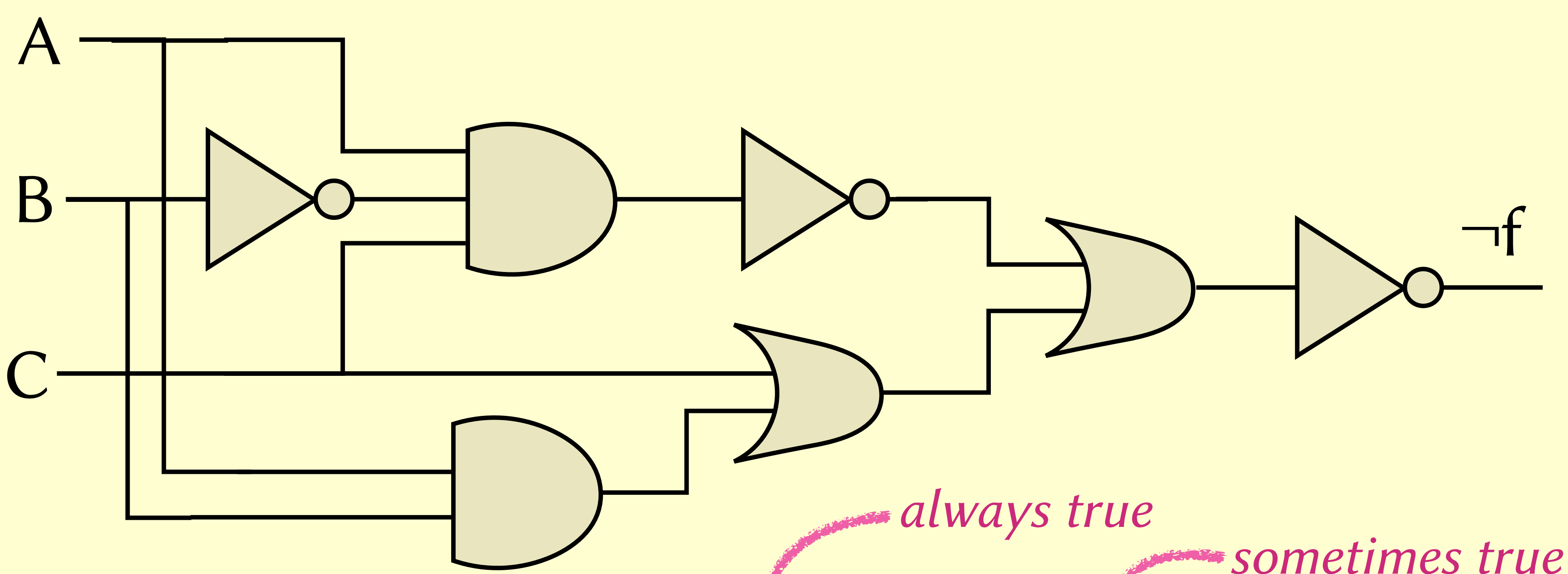
A formula can be VALID, SATISFIABLE, UNSATISFIABLE, or INVALID.

always false

SAT queries

- Simple case: proofs about Boolean statements.

  - $\neg f \;=\; \neg(\neg(A \wedge \neg B \wedge C) \vee (C \vee (B \wedge A)))$



A	B	C	¬f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

always true *sometimes true*

A formula can be VALID, SATISFIABLE, UNSATISFIABLE, or INVALID.

always false *sometimes false*

SAT solving

- A simple algorithm:

```
for A in {0, 1}:
  for B in {0, 1}:
    for C in {0, 1}:
      if f(A,B,C) = 1:
        return ("SAT", [A, B, C])
return ("UNSAT")
```

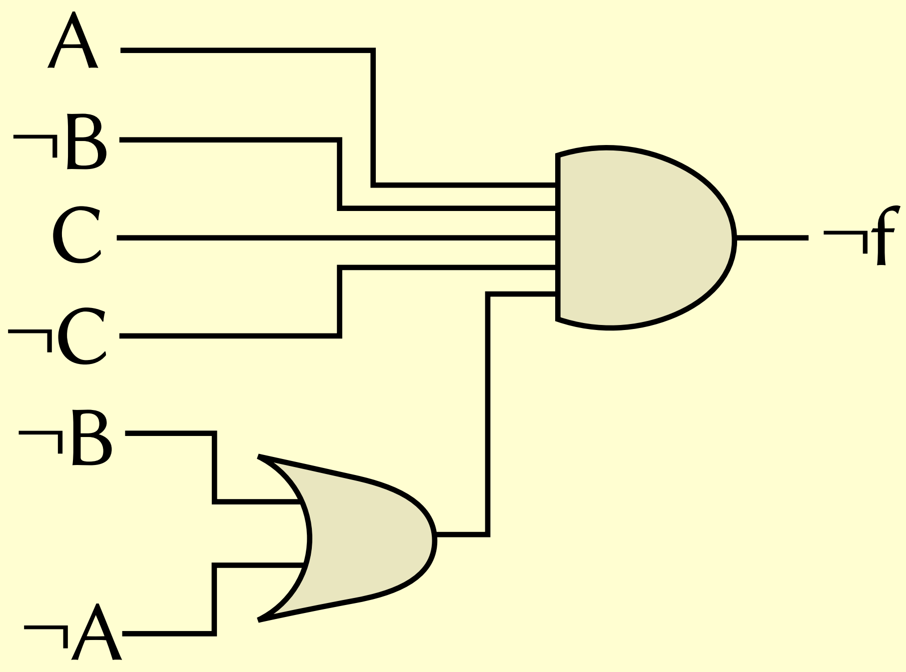
SAT solving

SAT solving

- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*

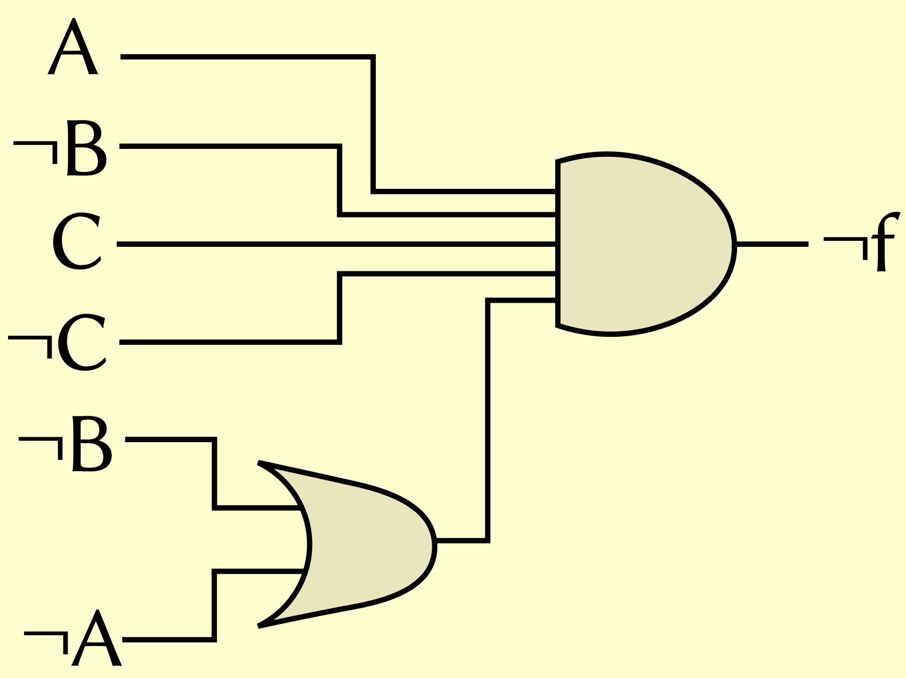
SAT solving

- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*



SAT solving

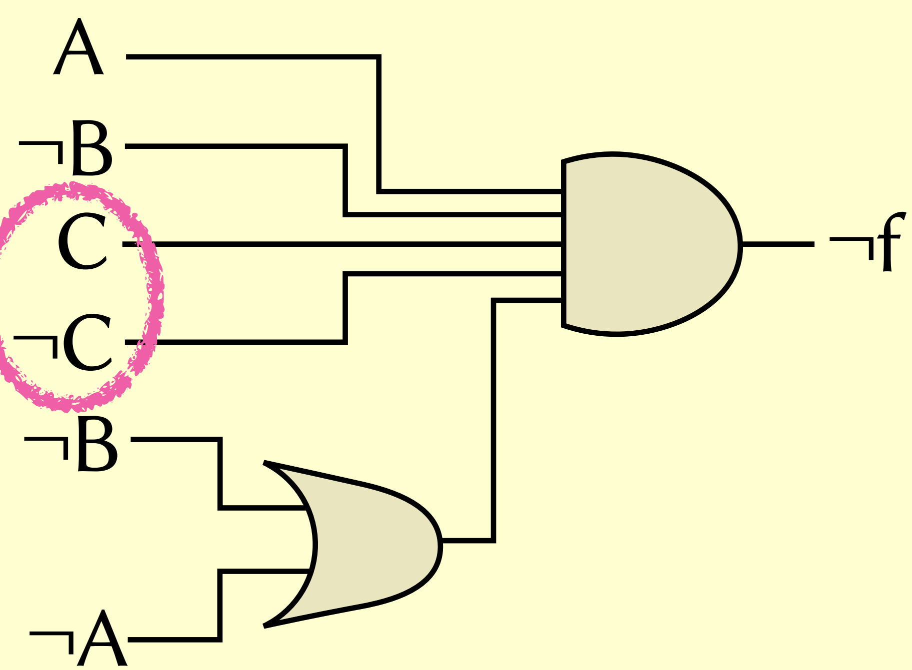
- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*



- It may then become obvious that the formula is UNSAT.

SAT solving

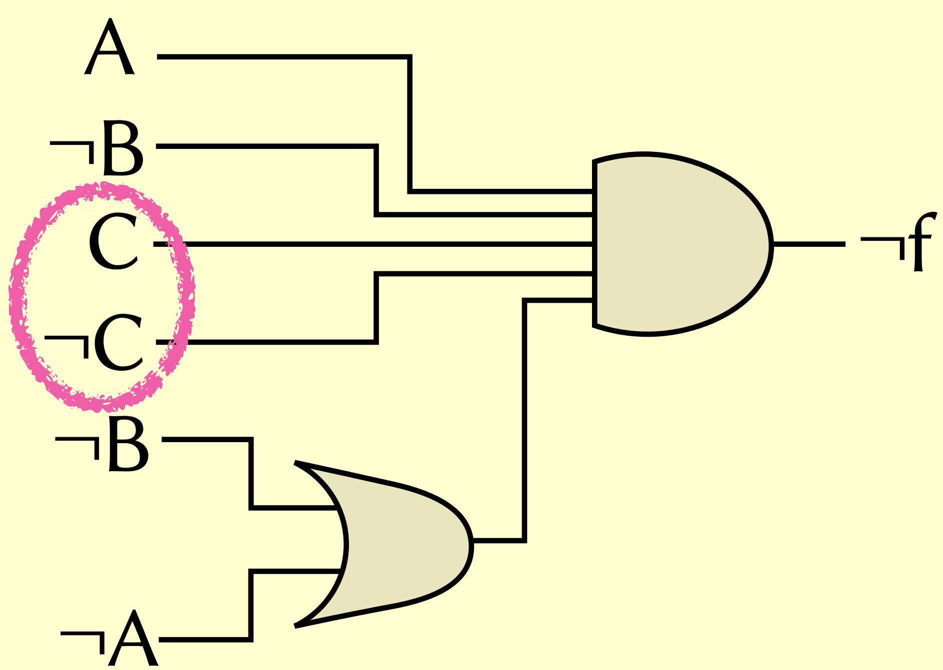
- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*



- It may then become obvious that the formula is UNSAT.

SAT solving

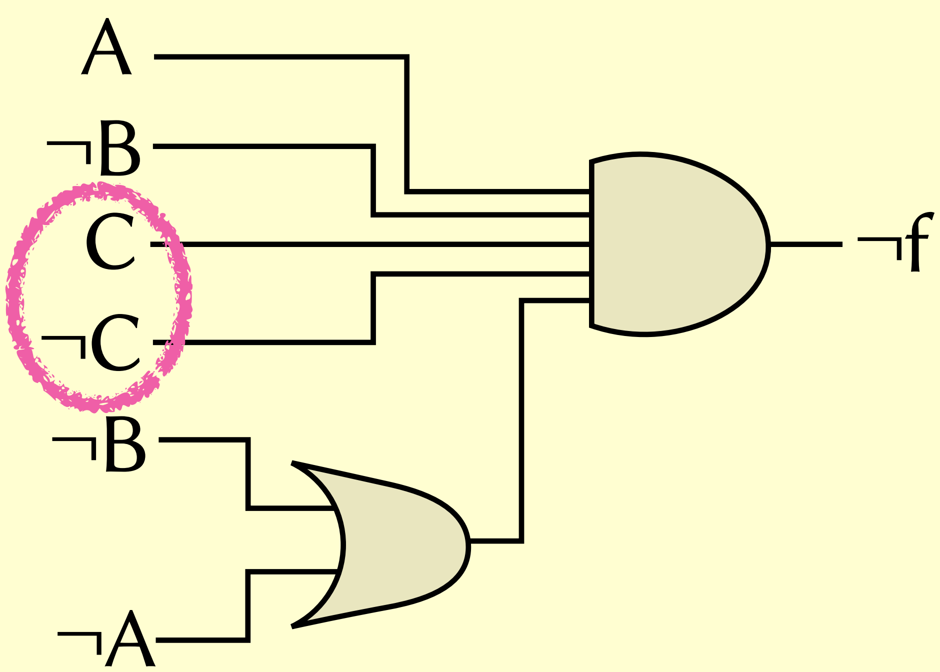
- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*



- It may then become obvious that the formula is UNSAT.
- If not, we can use the Davis–Putnam algorithm...

SAT solving

- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*

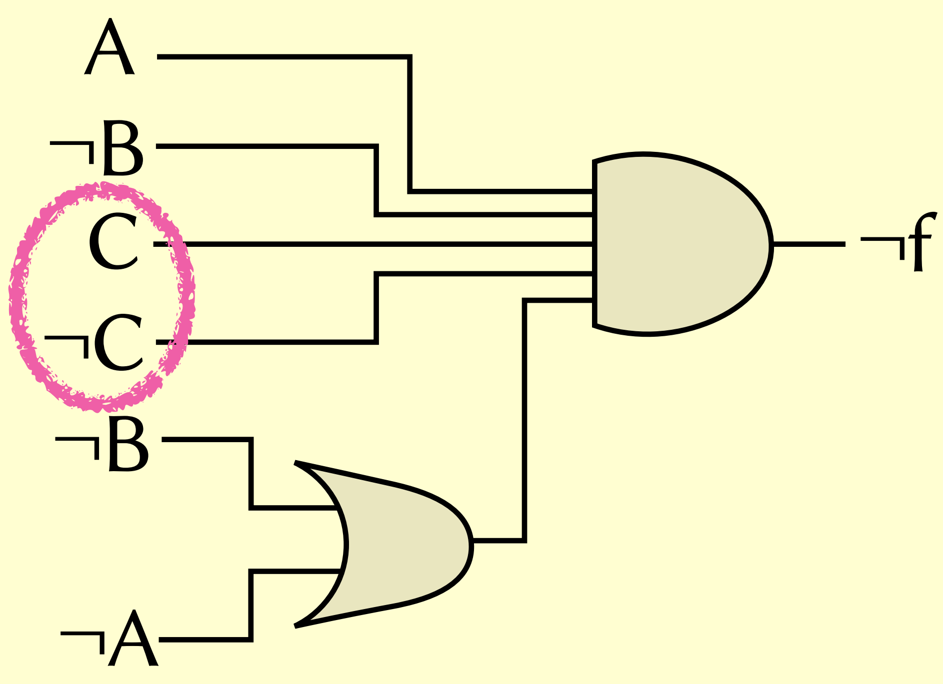


- It may then become obvious that the formula is UNSAT.
- If not, we can use the Davis–Putnam algorithm...

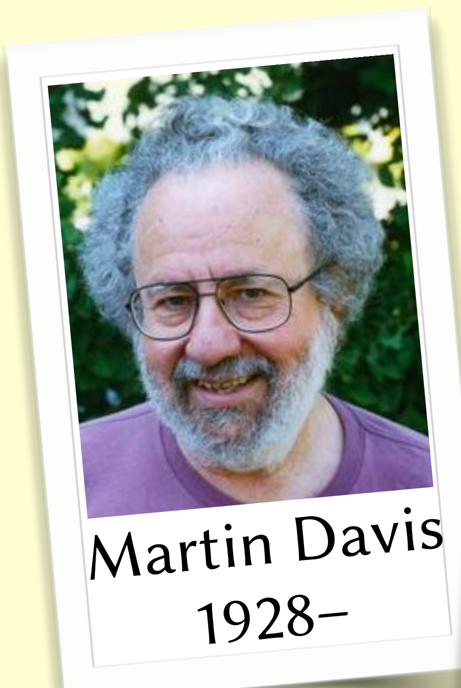
Martin Davis
1928–

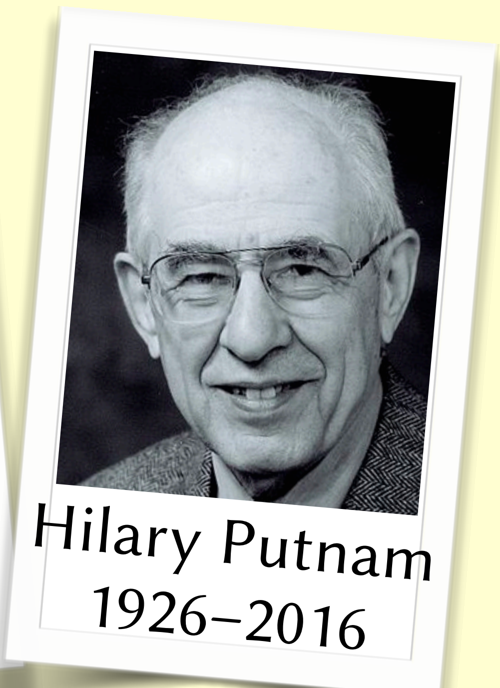
SAT solving

- A cleverer way: use de Morgan's rules to convert the formula to *conjunctive normal form.*



- It may then become obvious that the formula is UNSAT.
- If not, we can use the Davis–Putnam algorithm...


Martin Davis
1928–


Hilary Putnam
1926–2016

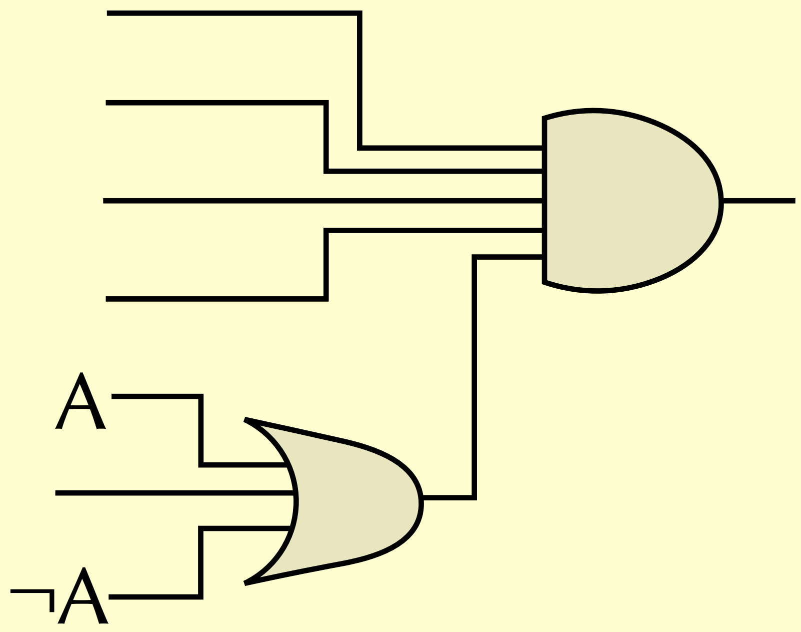
The DP method

The DP method

1. If an OR-gate takes both L and ¬L, delete it.

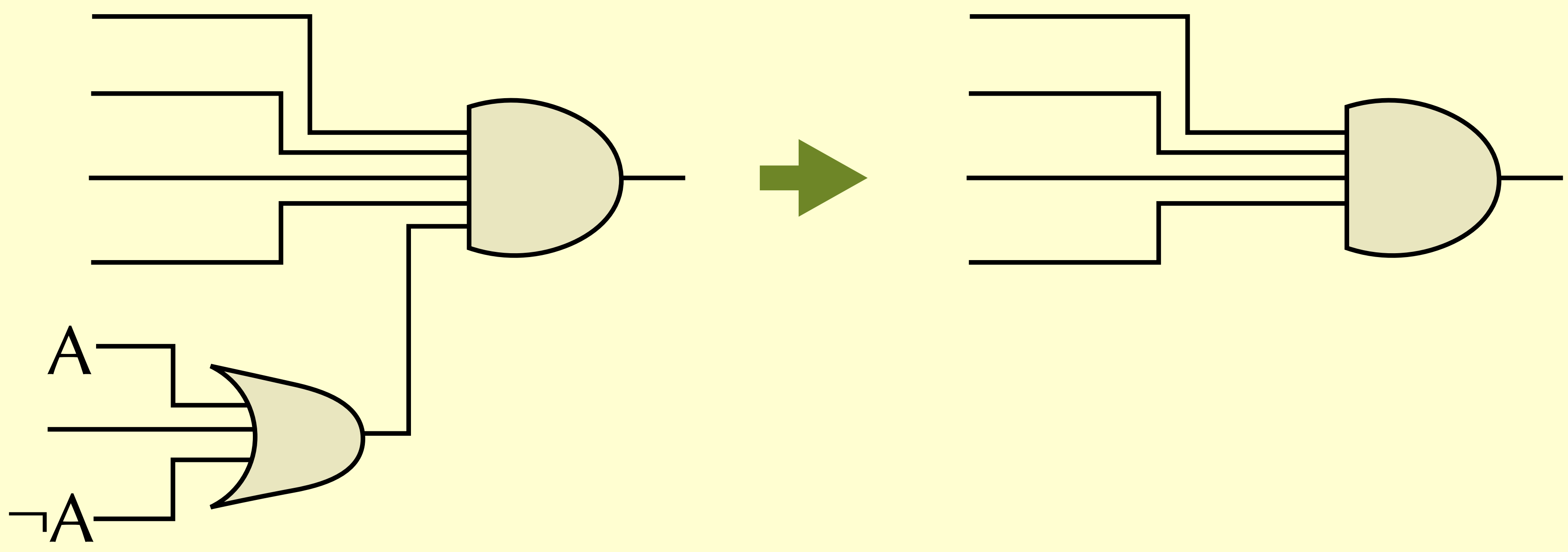
The DP method

1. If an OR-gate takes both L and ¬L, delete it.



The DP method

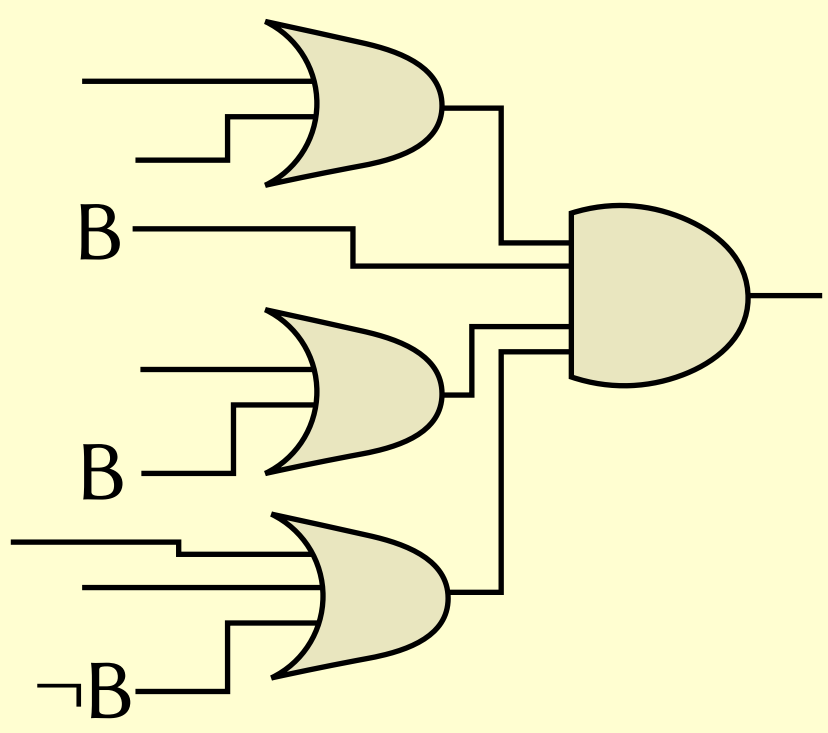
1. If an OR-gate takes both L and ¬L, delete it.



The DP method

1. If an OR-gate takes both L and ¬L, delete it.

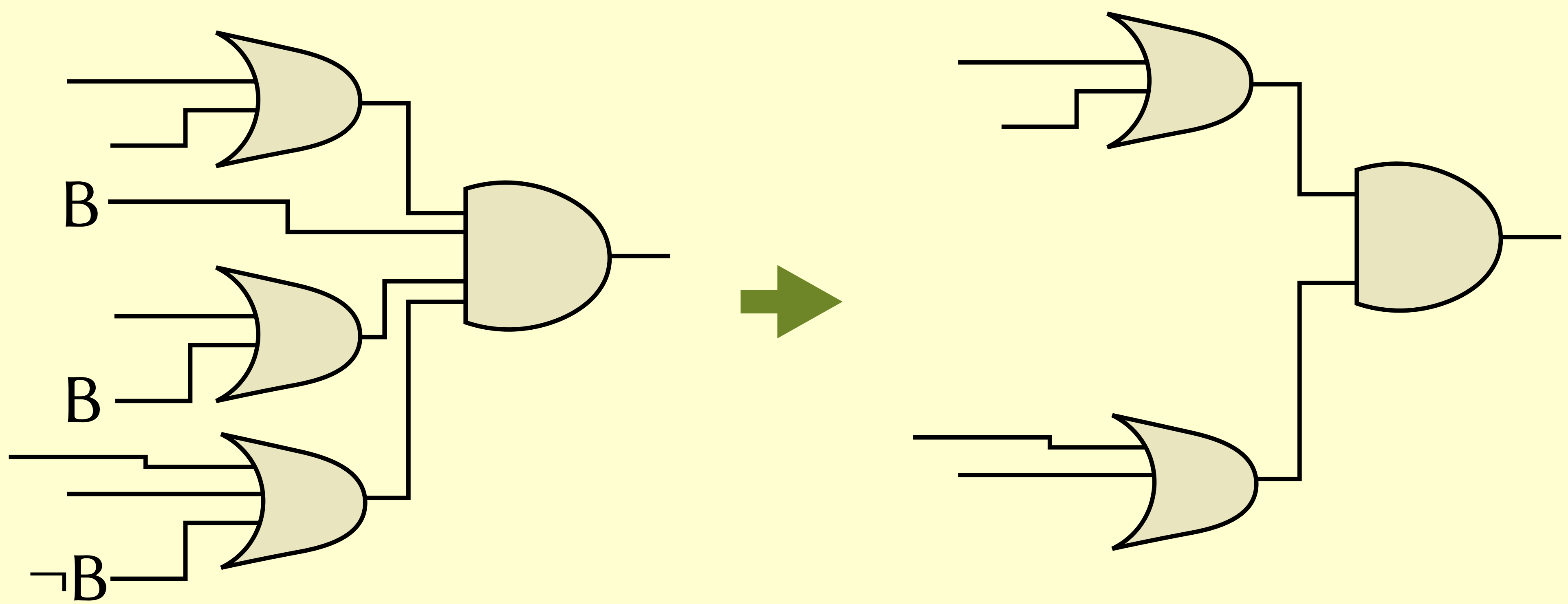
2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
(The solution, if it exists, will surely involve setting L=1.)



The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
(The solution, if it exists, will surely involve setting L=1.)



The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
   (The solution, if it exists, will surely involve setting L=1.)

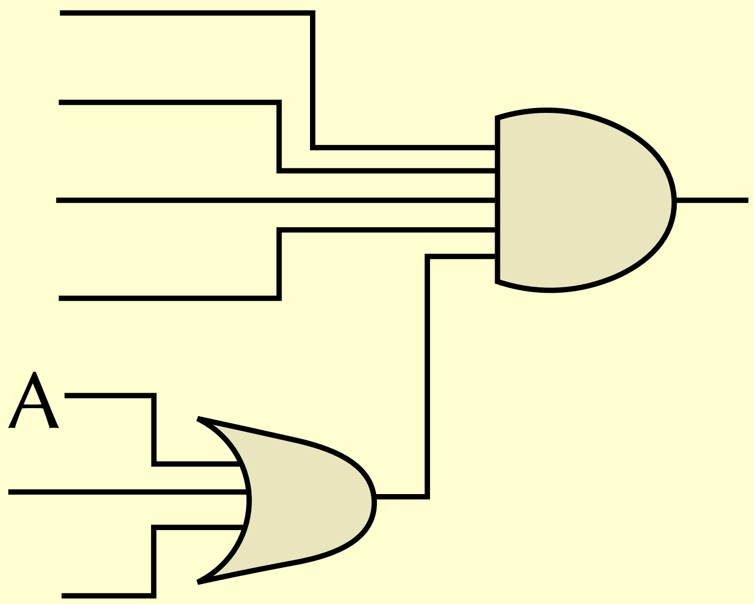
3. If L is unused, delete all OR-gates that take ¬L.
   (The solution, if it exists, will surely involve setting L=0.)

The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
(The solution, if it exists, will surely involve setting L=1.)

3. If L is unused, delete all OR-gates that take ¬L.
(The solution, if it exists, will surely involve setting L=0.)

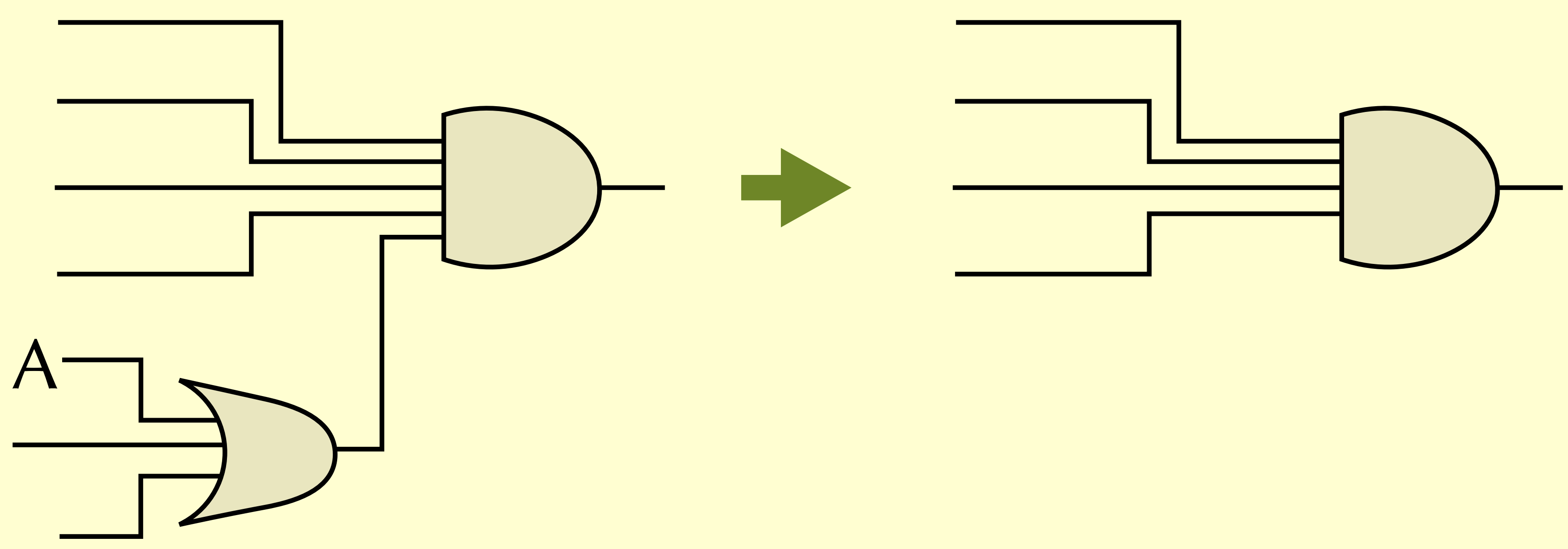


The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
(The solution, if it exists, will surely involve setting L=1.)

3. If L is unused, delete all OR-gates that take ¬L.
(The solution, if it exists, will surely involve setting L=0.)



The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
   (The solution, if it exists, will surely involve setting L=1.)

3. If L is unused, delete all OR-gates that take ¬L.
   (The solution, if it exists, will surely involve setting L=0.)

4. If any OR-gate has no inputs, the formula is false.

The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
(The solution, if it exists, will surely involve setting L=1.)

3. If L is unused, delete all OR-gates that take ¬L.
(The solution, if it exists, will surely involve setting L=0.)

4. If any OR-gate has no inputs, the formula is false.

5. If the AND-gate has no inputs, the formula is true.

The DP method

1. If an OR-gate takes both L and ¬L, delete it.

2. If L is connected directly to the AND-gate, delete it, delete all OR-gates that take L, and delete any connections to ¬L.
(The solution, if it exists, will surely involve setting L=1.)

3. If L is unused, delete all OR-gates that take ¬L.
(The solution, if it exists, will surely involve setting L=0.)

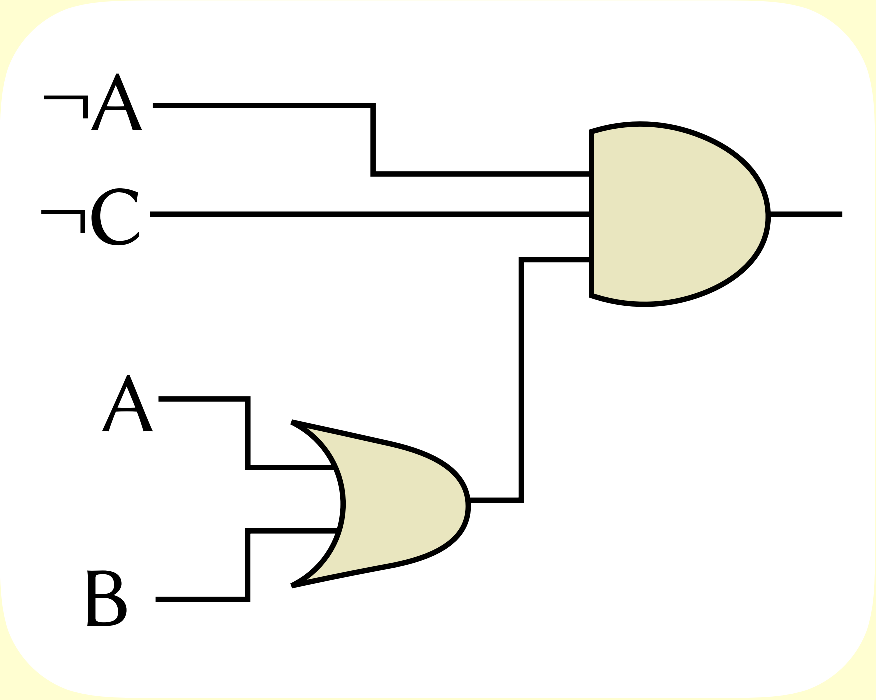
4. If any OR-gate has no inputs, the formula is false.

5. If the AND-gate has no inputs, the formula is true.

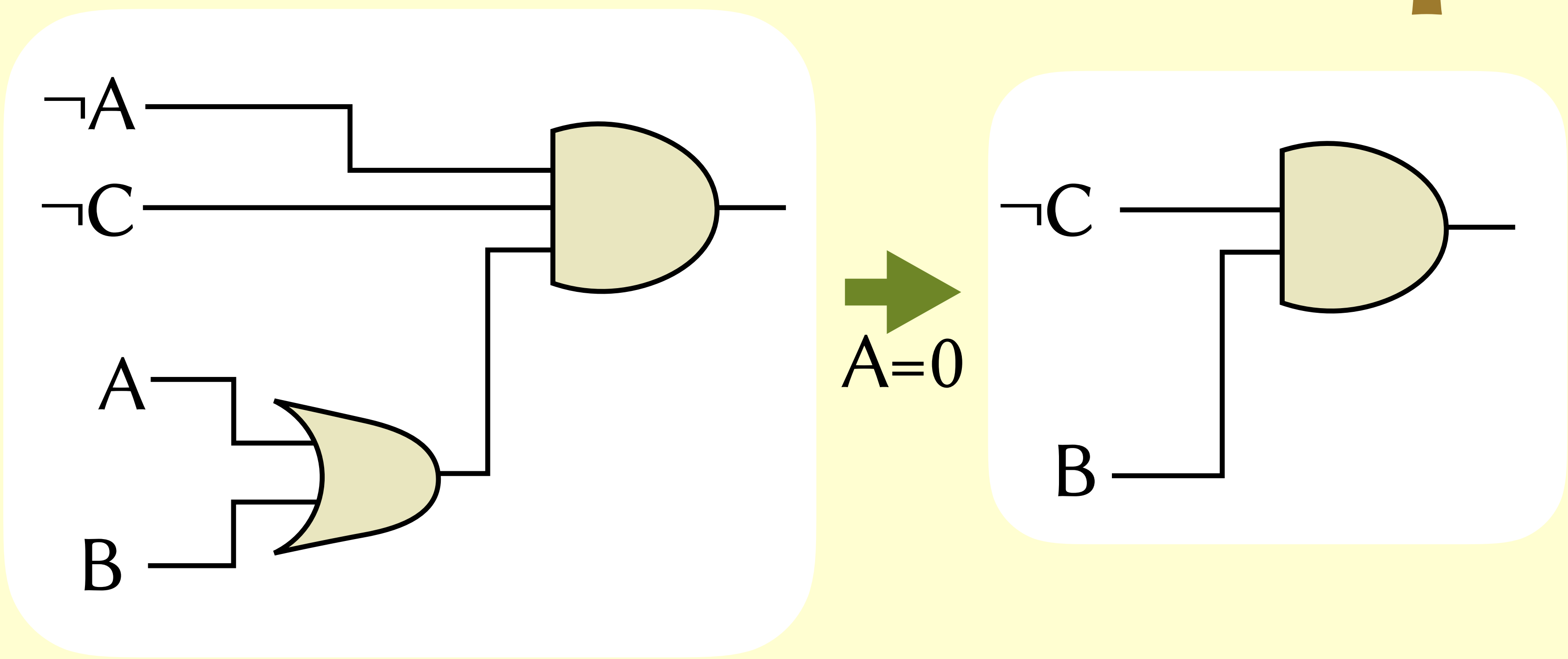
6. Pick a literal L and repeat the above for the cases L=0 and L=1.

DP example 1

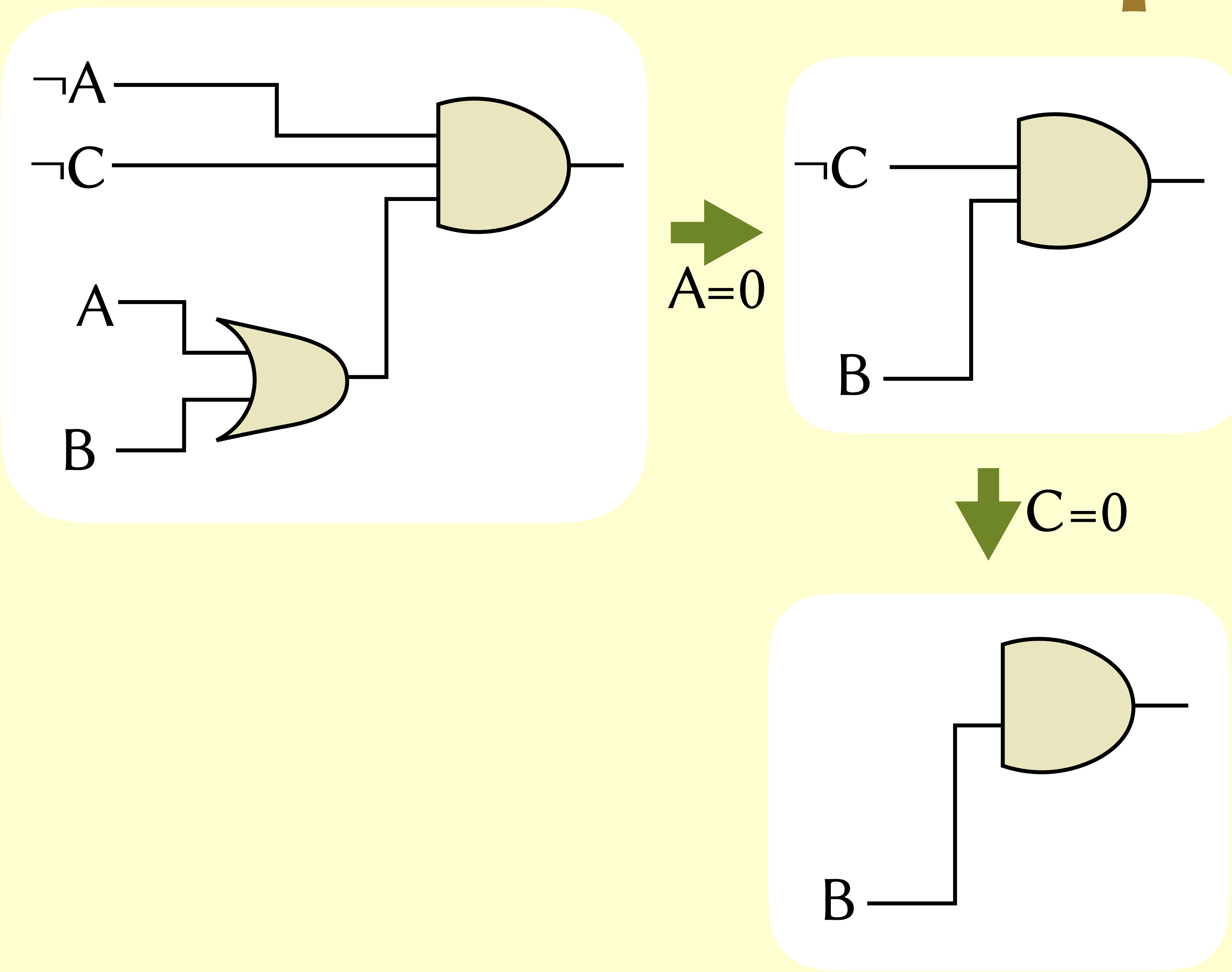
DP example 1



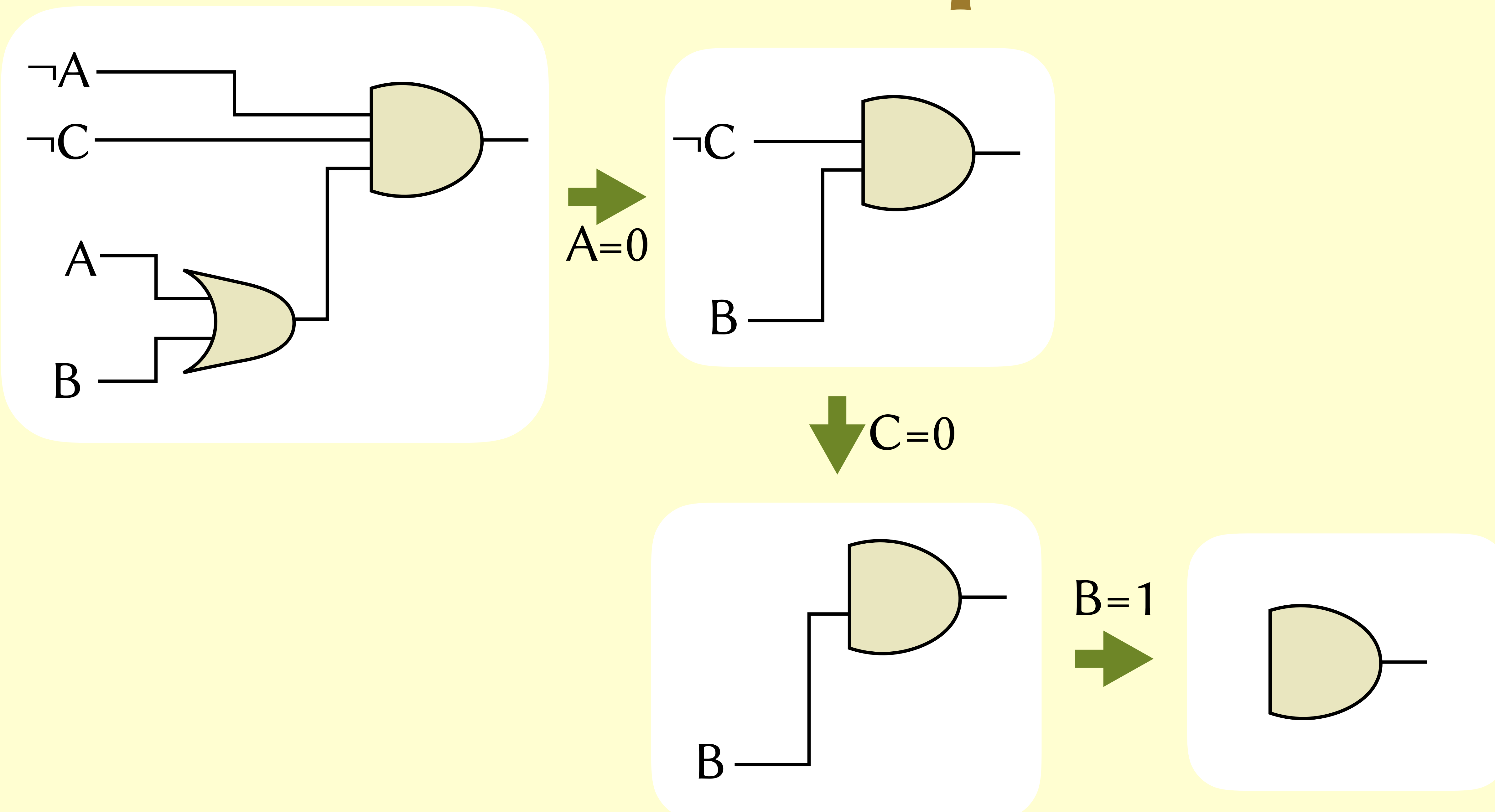
DP example 1



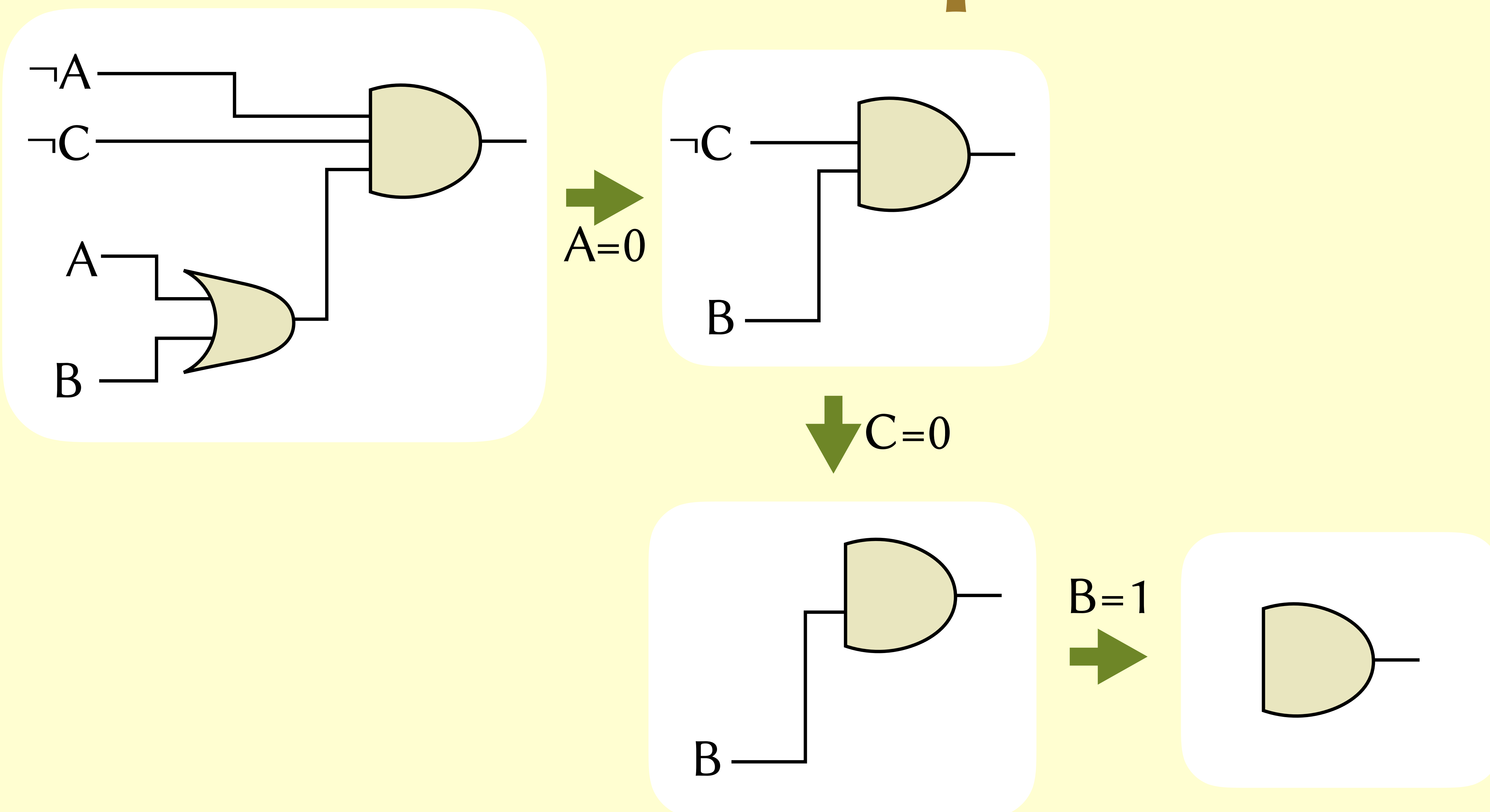
DP example 1



DP example 1



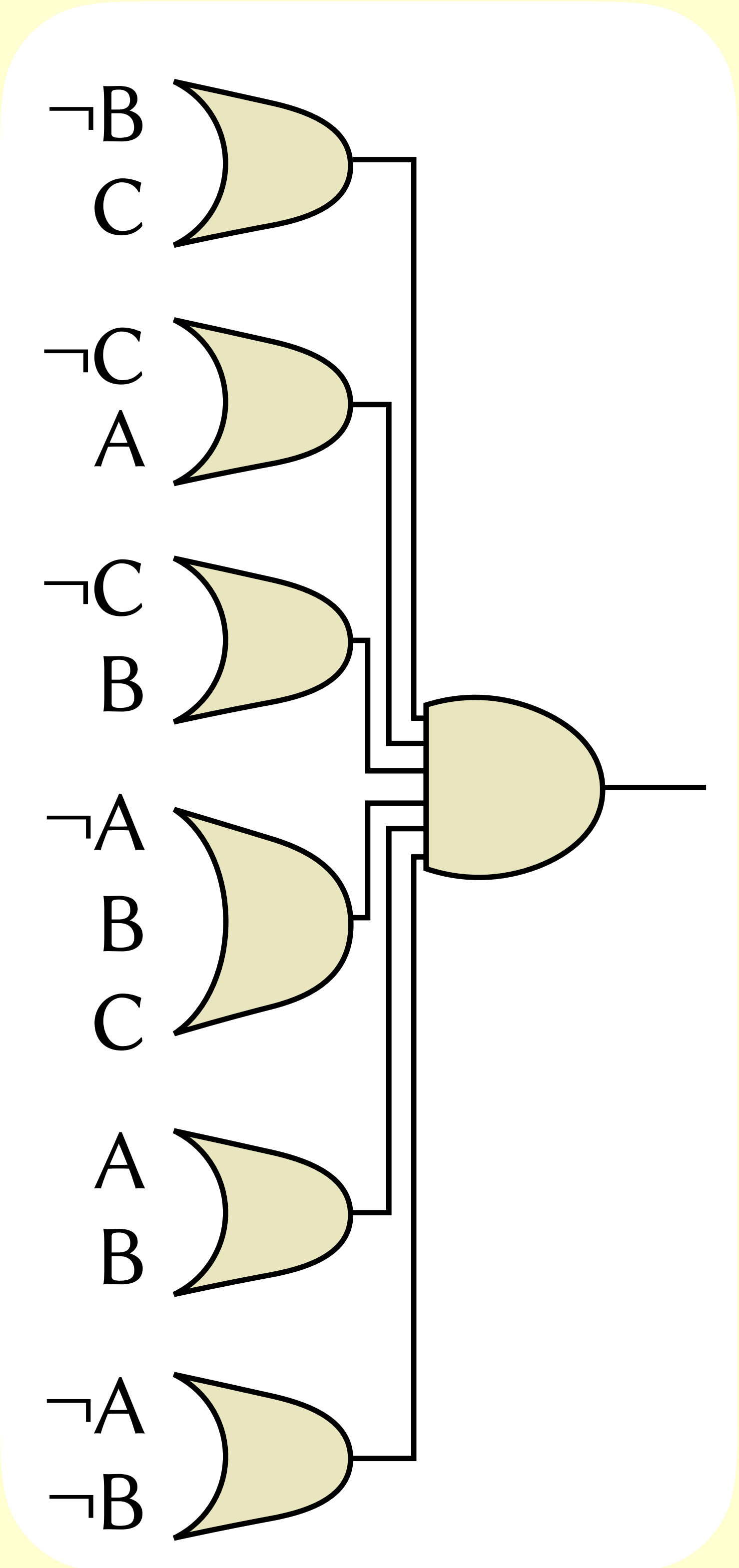
DP example 1



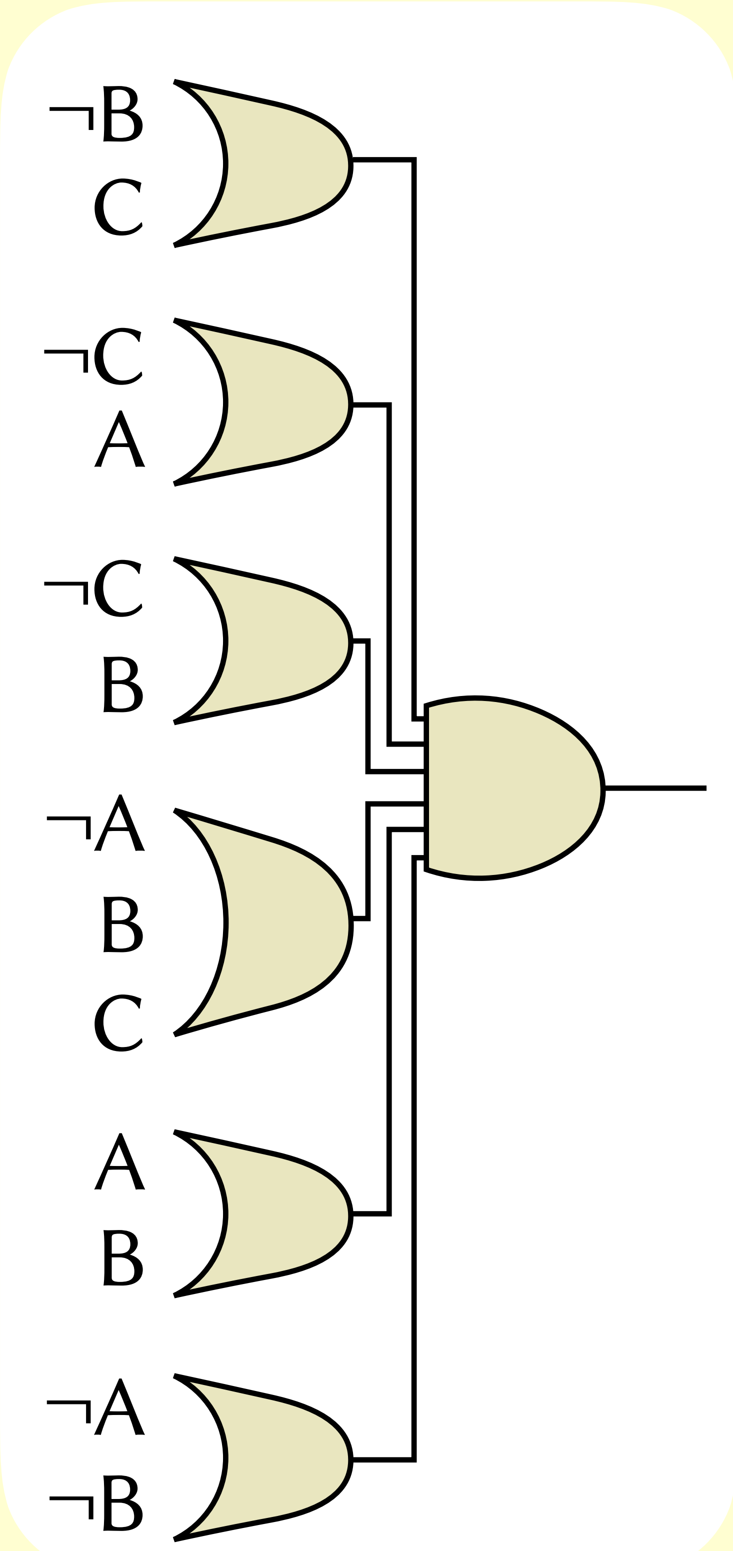
Satisfiable, e.g.
when A=0, B=1, C=0

DP example 2

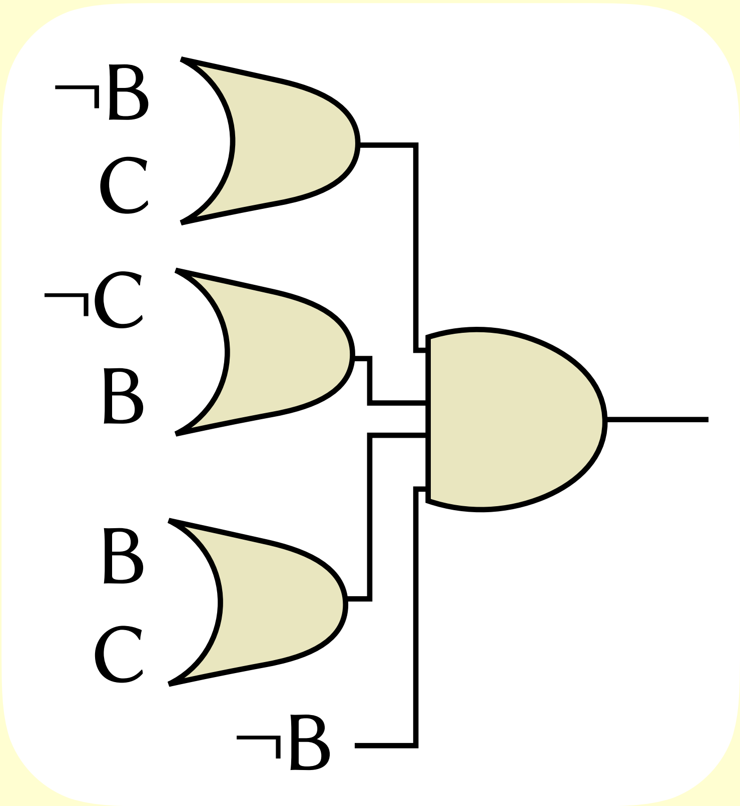
DP example 2



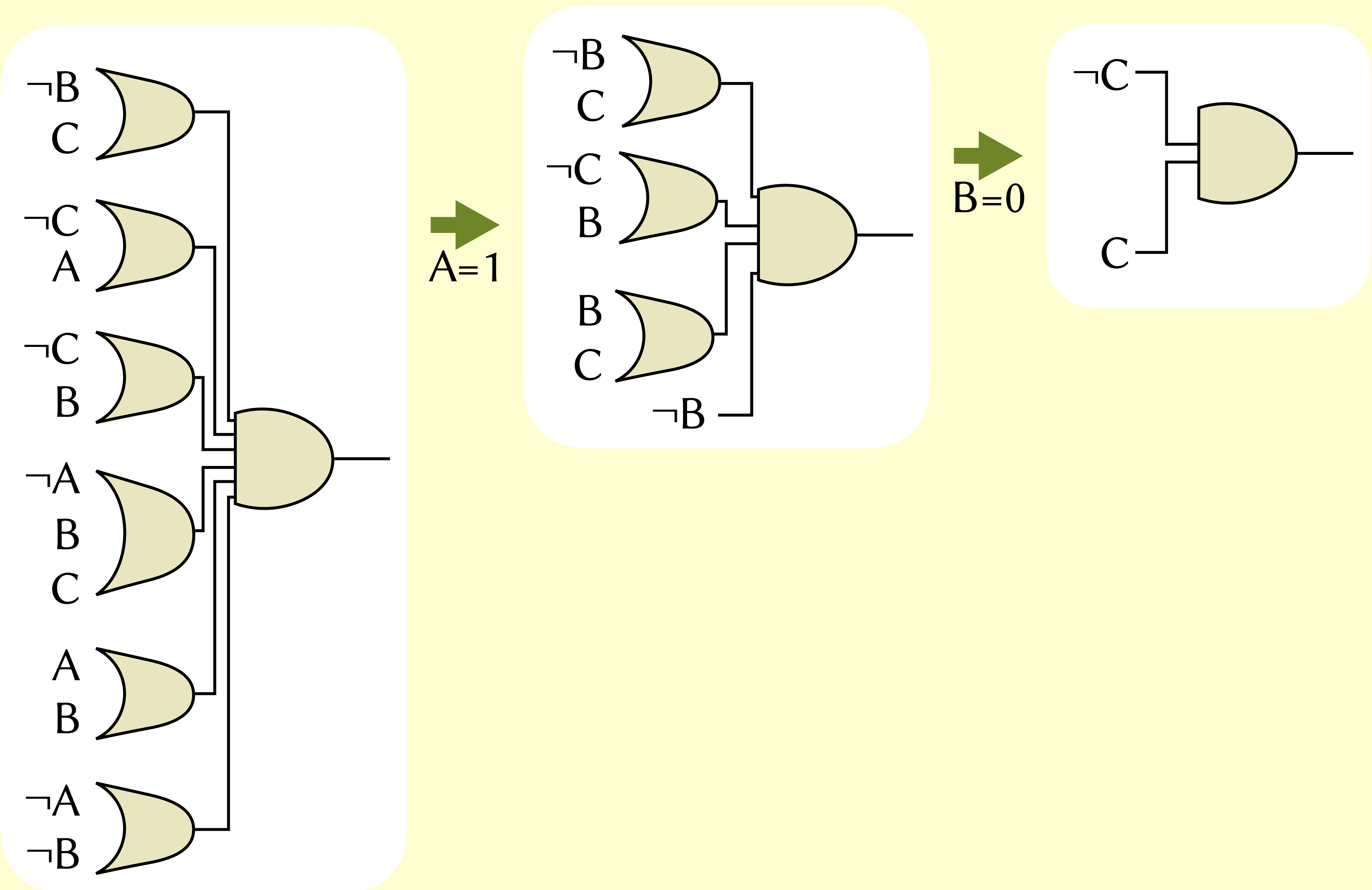
DP example 2



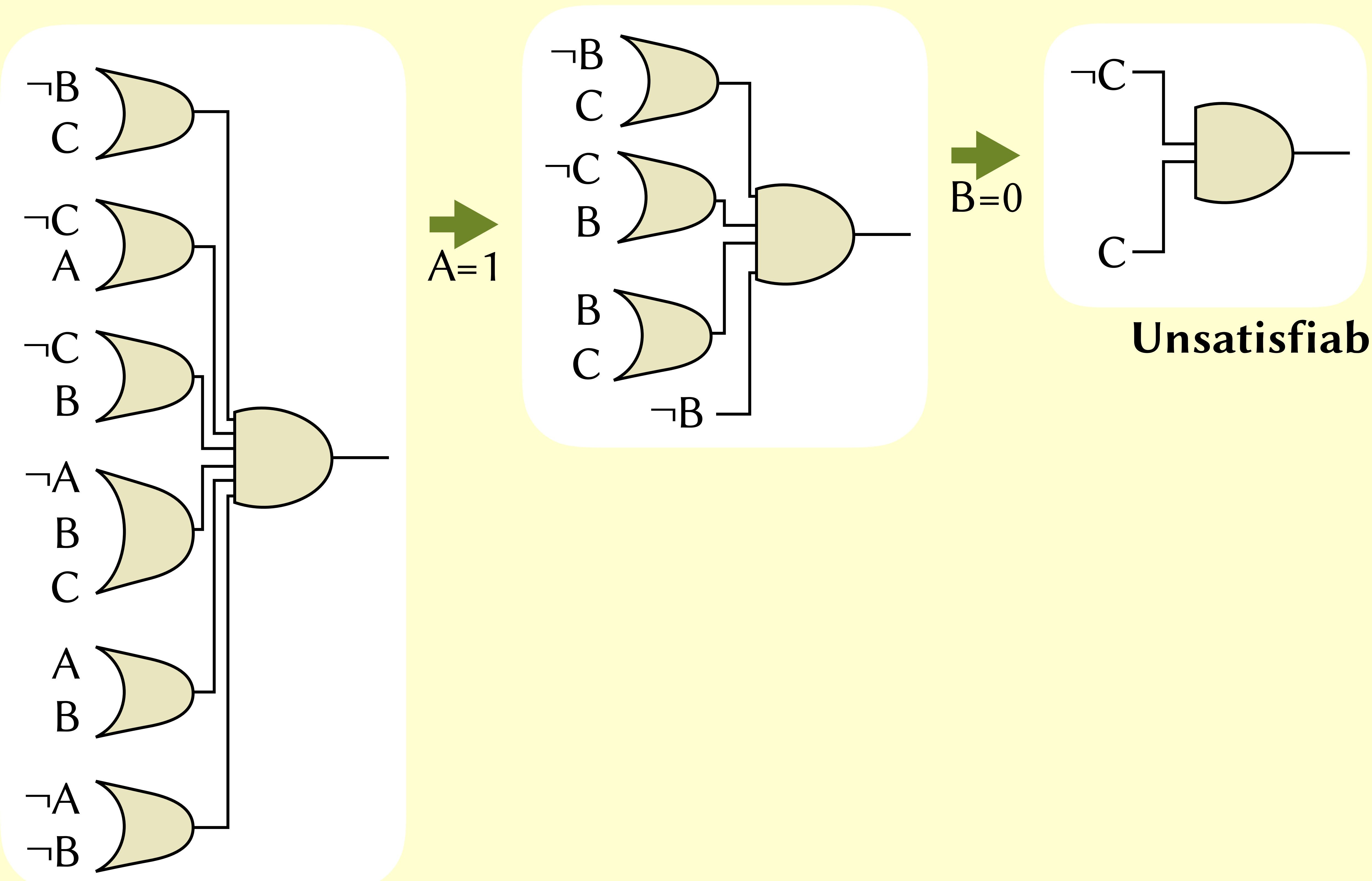
A=1



DP example 2

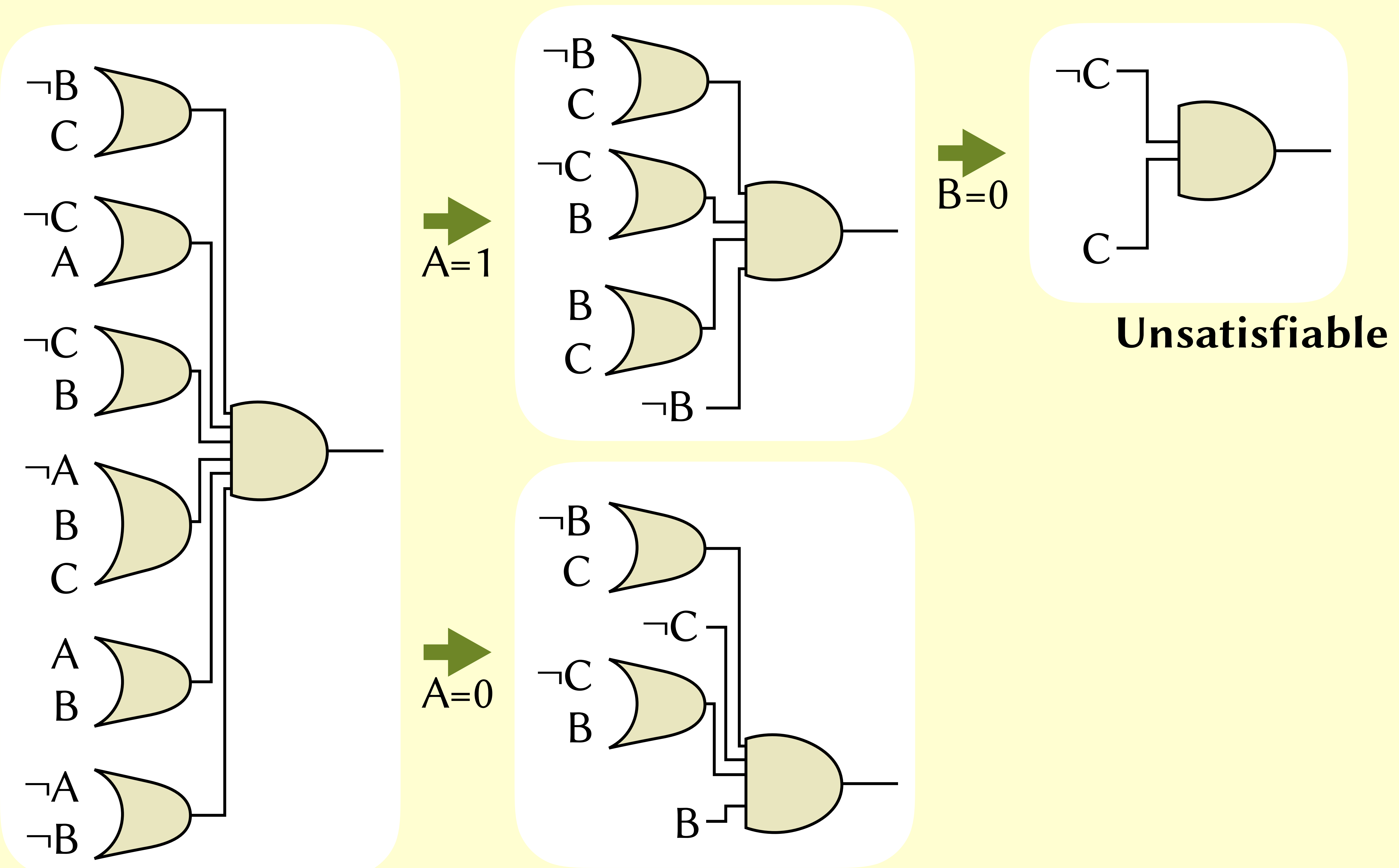


DP example 2

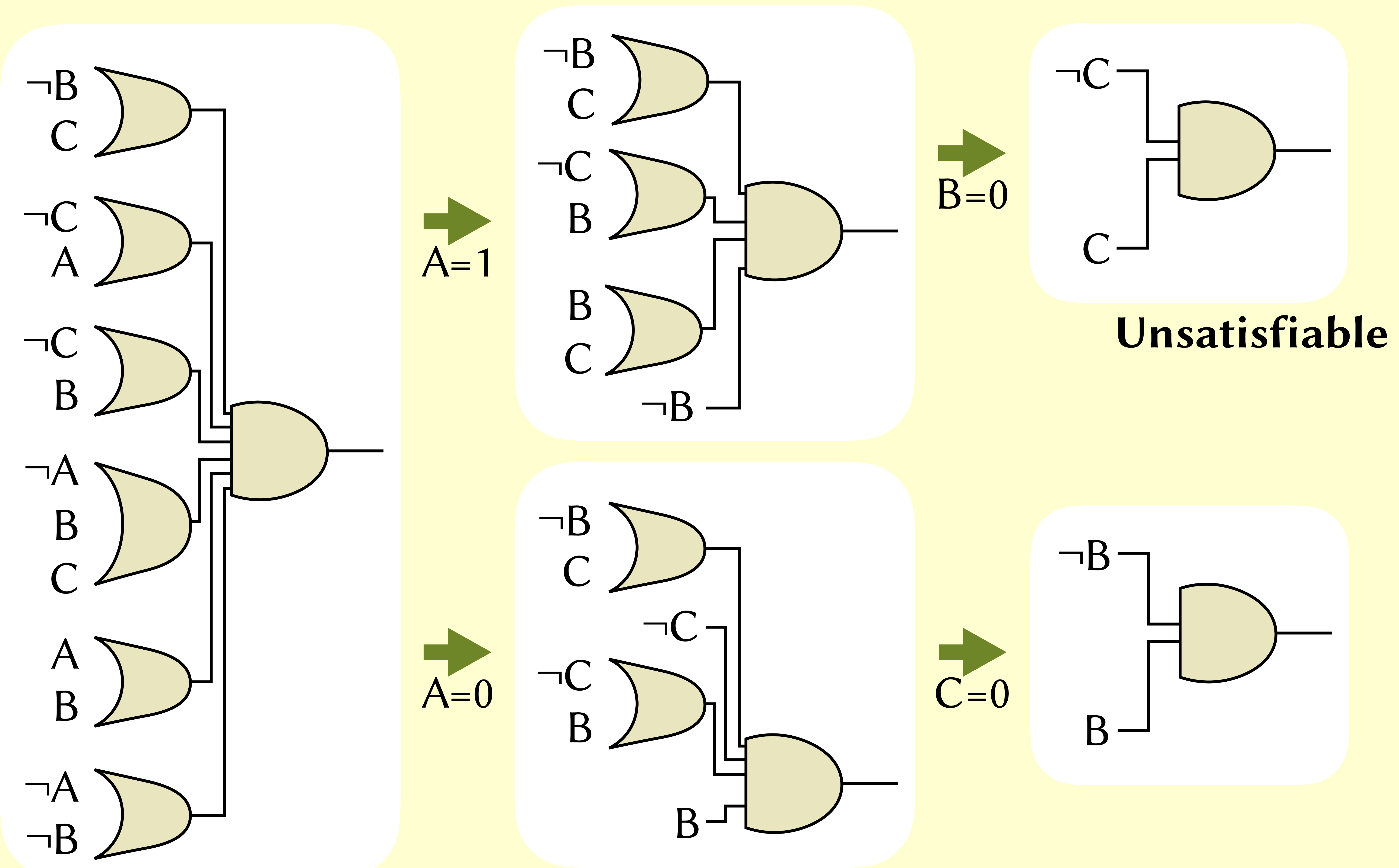


Unsatisfiable

DP example 2

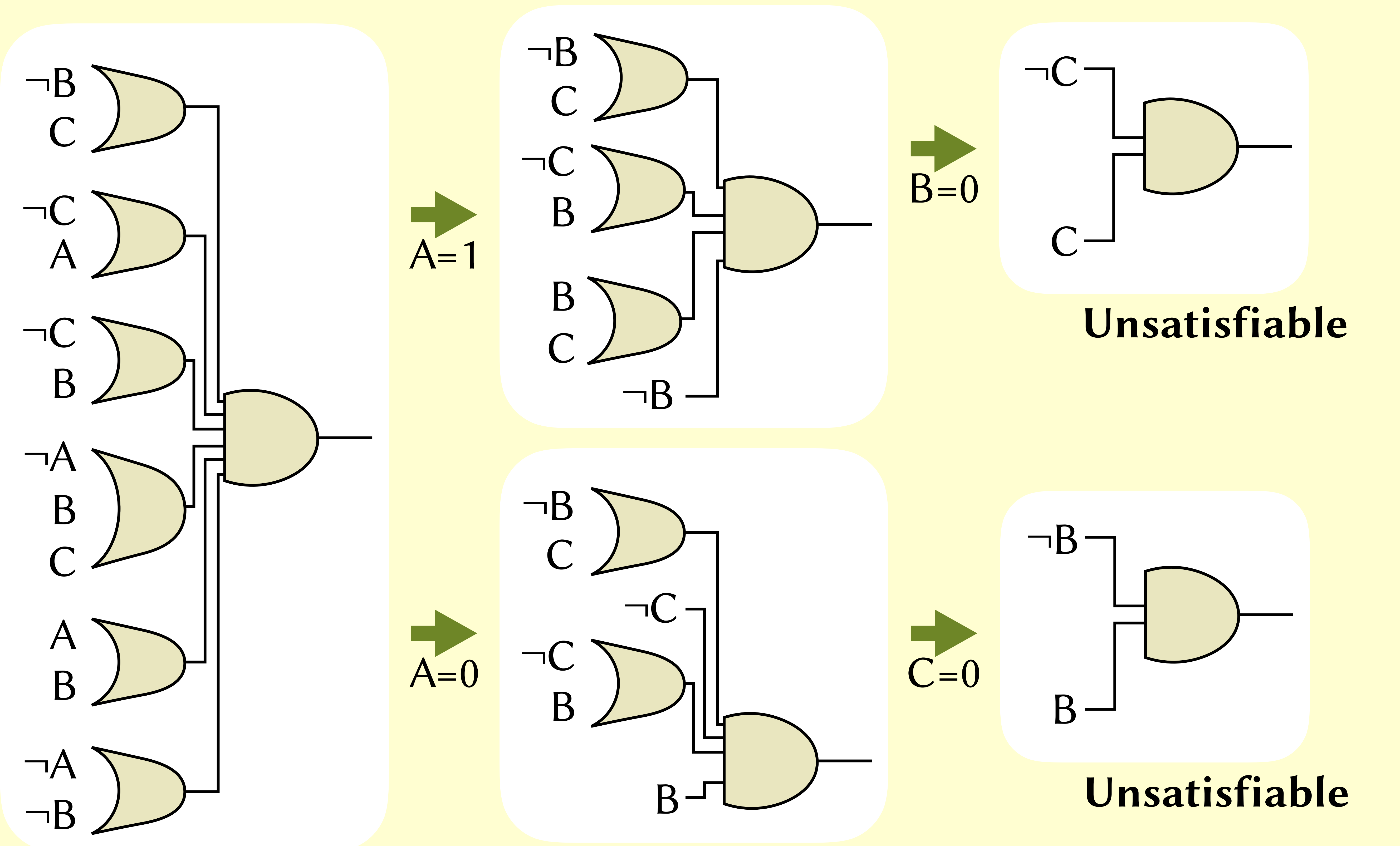


DP example 2



¬B
C
¬C
A
¬C
B
¬A
B
C
A
B
¬A
¬B
A=1
¬B
C
¬C
B
B
C
¬B
B=0
¬C
C
Unsatisfiable
A=0
¬B
C
¬C
¬C
B
B
C=0
¬B
B

DP example 2



¬B
C
¬C
A
¬C
B
¬A
B
C
A
B
¬A
¬B
A=1
A=0
¬B
C
¬C
B
B
C
¬B
¬B
C
¬C
¬C
B
B
B=0
C=0
¬C
C
Unsatisfiable
¬B
B
Unsatisfiable

Towards SMT solving

Towards SMT solving

- We can now prove basic Boolean formulas. But what about proving something like $A \times (B + C) = A \times B + A \times C$?

Towards SMT solving

- We can now prove basic Boolean formulas. But what about proving something like $A \times (B + C) = A \times B + A \times C$?

- If these are 32-bit integers, we could make this a SAT problem by treating each variable as 32 Boolean variables and encoding the rules of Boolean arithmetic.

Towards SMT solving

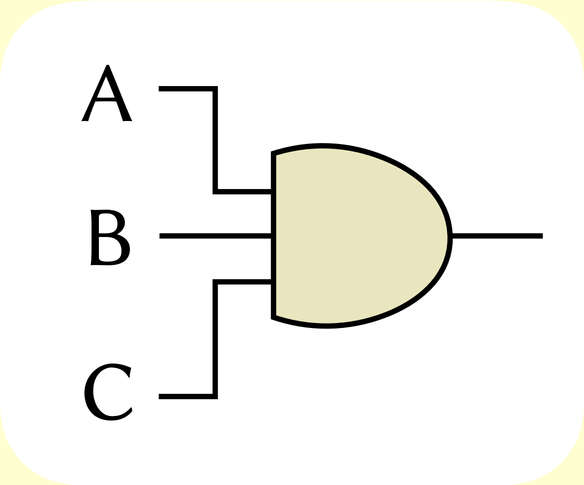
- We can now prove basic Boolean formulas. But what about proving something like $A \times (B + C) = A \times B + A \times C$?
- If these are 32-bit integers, we could make this a SAT problem by treating each variable as 32 Boolean variables and encoding the rules of Boolean arithmetic.
- Or we can move up to SMT: *satisfiability modulo theories.*

Towards SMT solving

- We can now prove basic Boolean formulas. But what about proving something like $A \times (B + C) = A \times B + A \times C$?

- If these are 32-bit integers, we could make this a SAT problem by treating each variable as 32 Boolean variables and encoding the rules of Boolean arithmetic.

- Or we can move up to SMT: *satisfiability modulo theories.*

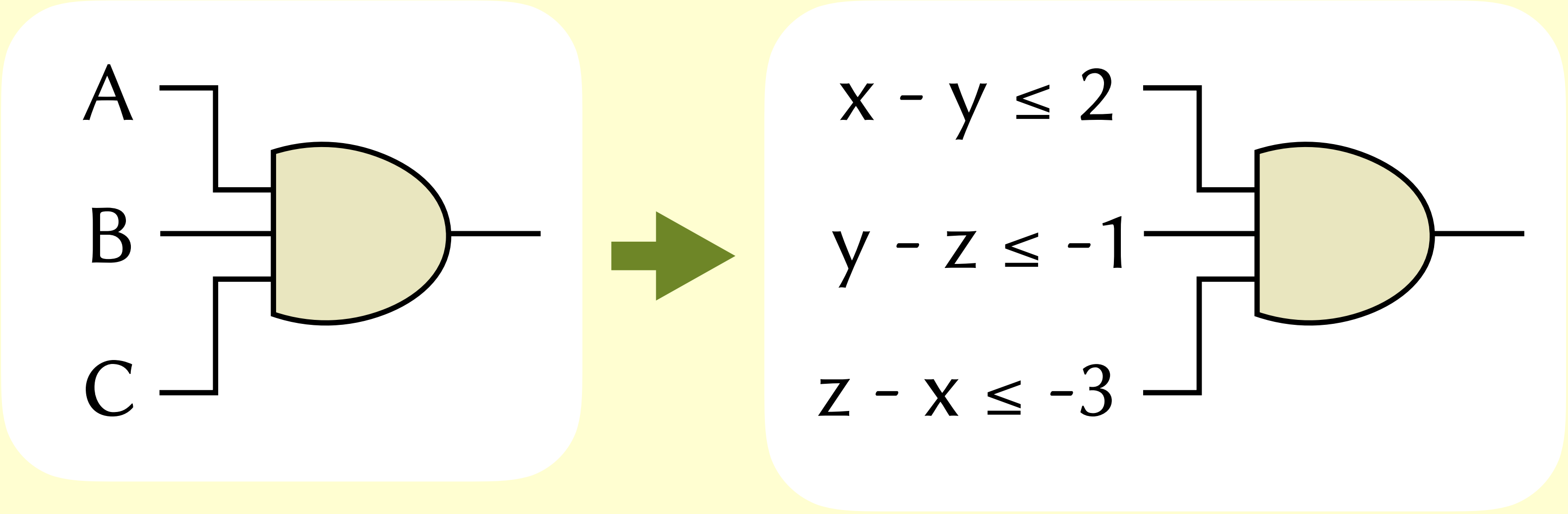


Towards SMT solving

- We can now prove basic Boolean formulas. But what about proving something like A × (B + C) = A × B + A × C?

- If these are 32-bit integers, we could make this a SAT problem by treating each variable as 32 Boolean variables and encoding the rules of Boolean arithmetic.

- Or we can move up to SMT: *satisfiability modulo theories.*



Some theories

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).
- **Difference logic**, where statements take the form $x - y \leq c$.

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).

- **Difference logic**, where statements take the form $x - y \leq c$.

- **Presburger arithmetic**, which allows statements about integers containing +, -, 0, 1, and =.

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).
- **Difference logic**, where statements take the form $x - y \leq c$.
- **Presburger arithmetic**, which allows statements about integers containing +, -, 0, 1, and =.

Mojżesz Presburger
1904–c.1943

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).
- **Difference logic**, where statements take the form $x - y \leq c$.
- **Presburger arithmetic**, which allows statements about integers containing +, -, 0, 1, and =.

1. $\neg (x + 1 = 0)$
2. $x + 1 = y + 1 \Rightarrow x = y$
3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. $(P(0) \land (\forall x.\ P(x) \Rightarrow P(x+1))) \Rightarrow \forall y.\ P(y)$ (for any P)

Mojżesz Presburger
1904–c.1943

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).

- **Difference logic**, where statements take the form x - y ≤ c.

- **Presburger arithmetic**, which allows statements about integers containing +, -, 0, 1, and =.

- **Non-linear arithmetic**, which allows queries like:

$$(\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y) \wedge \left(\neg b \vee y < -34.4 \vee \exp(x) > \frac{y}{x}\right)$$

Some theories

- **Equality and uninterpreted functions**, which knows that you can't have x=y and y=z without x=z, and that you can't have x=y without f(x)=f(y).

- **Difference logic**, where statements take the form x - y ≤ c.

- **Presburger arithmetic**, which allows statements about integers containing +, -, 0, 1, and =.

- **Non-linear arithmetic**, which allows queries like:
$$(\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y) \wedge \left(\neg b \vee y < -34.4 \vee \exp(x) > \frac{y}{x}\right)$$

- **Theory of arrays**, **theory of bit-vectors**, etc.

Decidability of Presburger

Decidability of Presburger

$x + y = z$

Decidability of Presburger

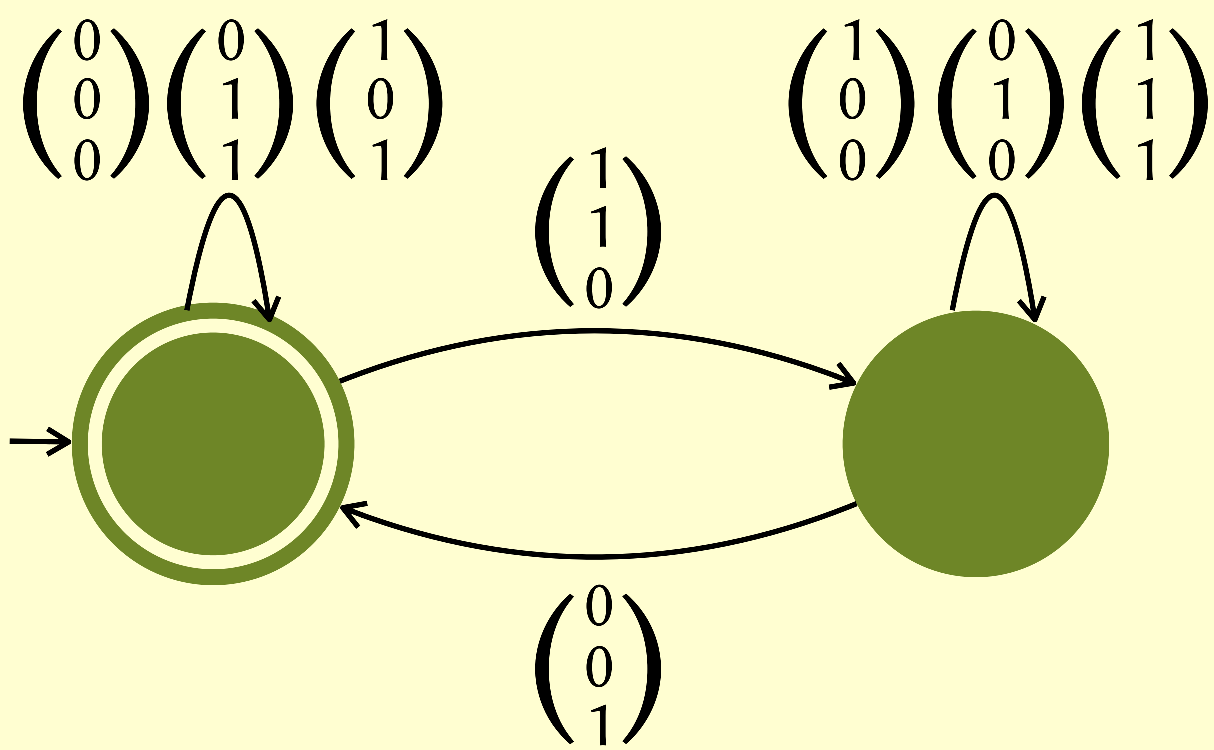
$x + y = z$

	1	2	4	8	16	32	64
x =	0	1	0	0	1	0	0
y =	0	1	0	1	0	1	0
z =	0	0	1	1	1	1	0

Decidability of Presburger

x + y = z

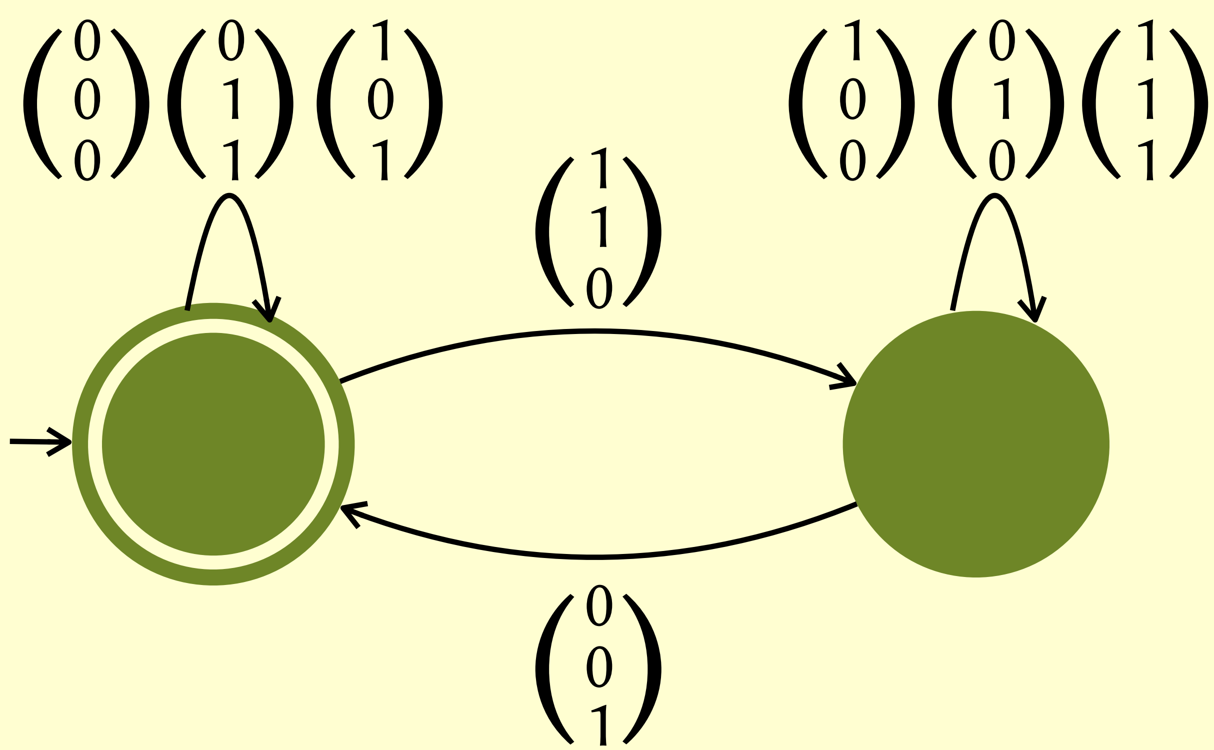
	1	2	4	8	16	32	64
x =	0	1	0	0	1	0	0
y =	0	1	0	1	0	1	0
z =	0	0	1	1	1	1	0



Decidability of Presburger

$x + y = z$

	1	2	4	8	16	32	64
x =	0	1	0	0	1	0	0
y =	0	1	0	1	0	1	0
z =	0	0	1	1	1	1	0



Julius Richard Büchi
1924–1984

Adding multiplication

1. $\neg (x + 1 = 0)$
2. $x + 1 = y + 1 \Longrightarrow x = y$
3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. $x \times 0 = 0$
6. $x \times (y + 1) = x \times y + x$
7. $(P(0) \wedge (\forall x.\ P(x) \Longrightarrow P(x+1))) \Longrightarrow \forall y.\ P(y)$ (for any P)

Adding multiplication

1. $\neg (x + 1 = 0)$
2. $x + 1 = y + 1 \Rightarrow x = y$
3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. $x \times 0 = 0$
6. $x \times (y + 1) = x \times y + x$
7. $(P(0) \land (\forall x.\ P(x) \Rightarrow P(x+1))) \Rightarrow \forall y.\ P(y)$ (for any P)

Giuseppe Peano
1858–1932

Undecidability of Peano

Undecidability of Peano

- Now we have multiplication, we can write a statement representing the Collatz conjecture: does there exist an infinite sequence of positive integers x_0, x_1, x_2, ... such that

$$
\begin{array}{ll}
2 \times x_{i+1} = x_i & \text{if } x_i \text{ is even} \\
x_{i+1} = 3 \times x_i + 1 & \text{if } x_i \text{ is odd}
\end{array}
$$

Undecidability of Peano

- Now we have multiplication, we can write a statement representing the Collatz conjecture: does there exist an infinite sequence of positive integers x_0, x_1, x_2, ... such that

$$
\begin{array}{ll}
2 \times x_{i+1} = x_i & \text{if } x_i \text{ is even} \\
x_{i+1} = 3 \times x_i + 1 & \text{if } x_i \text{ is odd}
\end{array}
$$

- So **if** arithmetic with multiplication were decidable, we could solve the Collatz conjecture automatically!

Undecidability of Peano

Undecidability of Peano

- Suppose I have an algorithm that can take an arithmetic statement and tell me whether it is true or not.

Undecidability of Peano

- Suppose I have an algorithm that can take an arithmetic statement and tell me whether it is true or not.

- The Halting Problem can be encoded as a statement about arithmetic.

Undecidability of Peano

- Suppose I have an algorithm that can take an arithmetic statement and tell me whether it is true or not.

- The Halting Problem can be encoded as a statement about arithmetic.

- So I can use my algorithm to solve the Halting Problem.

Undecidability of Peano

- Suppose I have an algorithm that can take an arithmetic statement and tell me whether it is true or not.
- The Halting Problem can be encoded as a statement about arithmetic.
- So I can use my algorithm to solve the Halting Problem.
- But it is impossible to write an algorithm to solve the Halting Problem!

Undecidability of Peano

- Suppose I have an algorithm that can take an arithmetic statement and tell me whether it is true or not.
- The Halting Problem can be encoded as a statement about arithmetic.
- So I can use my algorithm to solve the Halting Problem.
- But it is impossible to write an algorithm to solve the Halting Problem!
- So it must also be impossible to write an algorithm to decide whether arithmetic statements are true or not.

Aside: Halting Problem

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

- **Examples:**

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

- **Examples:**

```
S1 = "int s1(char *D) {
        while(1);
      }"
```

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

- **Examples:**

```
S1 = "int s1(char *D) {
        while(1);
      }"
```

`halts(S1, _)` = 0

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

- **Examples:**

```
S1 = "int s1(char *D) {
        while(1);
     }"
```

```
S6 = "int s6(char *D) {
        return 42;
     }"
```

`halts(S1, _)` = 0

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

- **Examples:**

```
S1 = "int s1(char *D) {
        while(1);
      }"
```

`halts(S1, _)` = 0

```
S6 = "int s6(char *D) {
        return 42;
      }"
```

`halts(S6, _)` = 1

Aside: Halting Problem

Aside: Halting Problem

Aside: Halting Problem

```
S1

S2

S3

S4

S5

S6
```

Aside: Halting Problem

	S1	S2	S3	S4	S5	S6
S1						
S2						
S3						
S4						
S5						
S6						

Aside: Halting Problem

	S1	S2	S3	S4	S5	S6
S1						
S2						
S3						
S4						
S5						
S6						

strings interpreted as source code

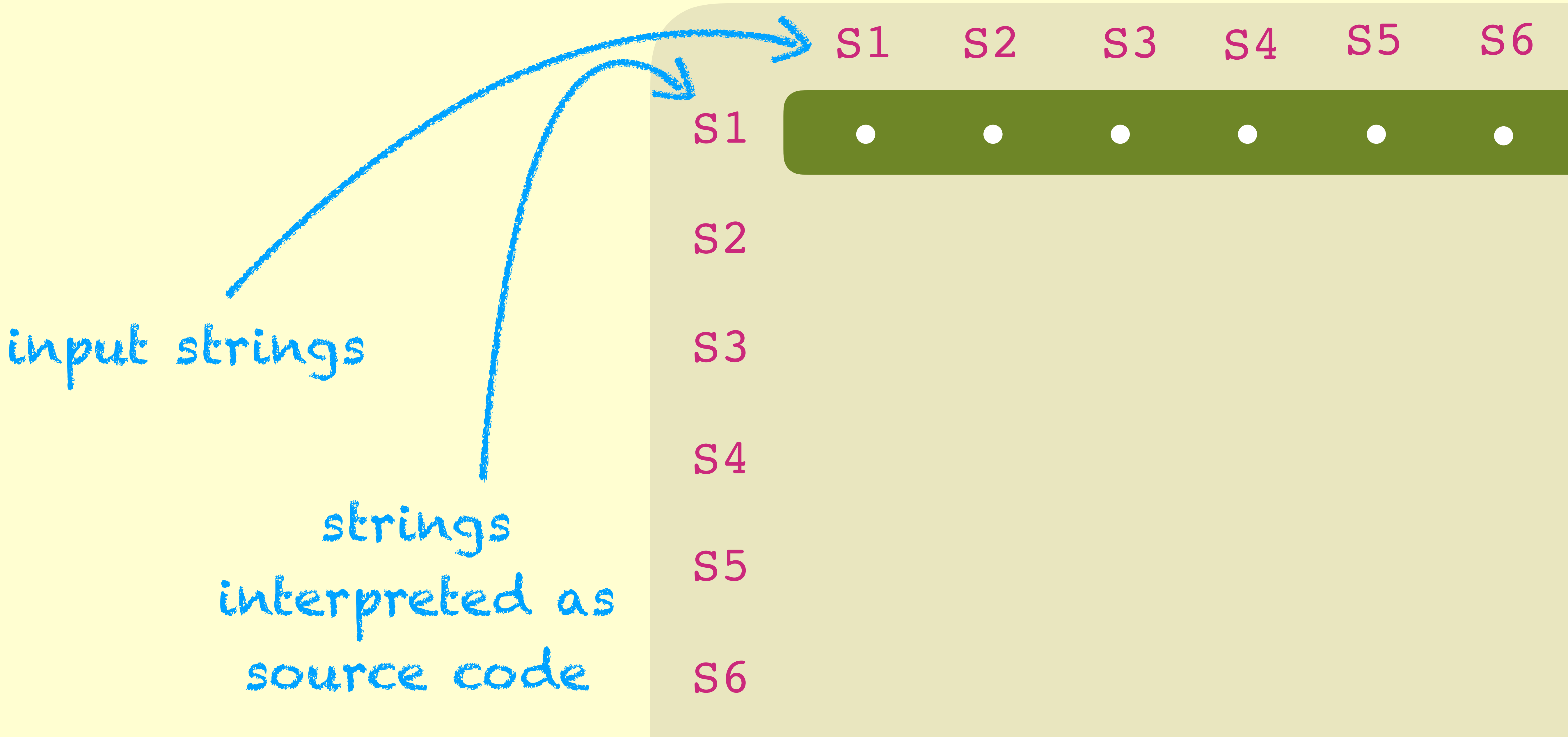
Aside: Halting Problem

input strings

strings interpreted as source code

	S1	S2	S3	S4	S5	S6
S1						
S2						
S3						
S4						
S5						
S6						

Aside: Halting Problem



Aside: Halting Problem

input strings

strings interpreted as source code

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2						
S3						
S4						
S5						
S6	✔	✔	✔	✔	✔	✔

Aside: Halting Problem

input strings

strings interpreted as source code

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2	✔	✔	✔	•	•	•
S3	•	✔	•	✔	•	•
S4	•	•	•	•	•	✔
S5	✔	•	✔	•	✔	•
S6	✔	✔	✔	✔	✔	✔

Aside: Halting Problem

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2	✔	✔	✔	•	•	•
S3	•	✔	•	✔	•	•
S4	•	•	•	•	•	✔
S5	✔	•	✔	•	✔	•
S6	✔	✔	✔	✔	✔	✔

input strings

strings interpreted as source code

Aside: Halting Problem

•	✔	•	•	✔	✔

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2	✔	✔	✔	•	•	•
S3	•	✔	•	✔	•	•
S4	•	•	•	•	•	✔
S5	✔	•	✔	•	✔	•
S6	✔	✔	✔	✔	✔	✔

input strings

strings
interpreted as
source code

Aside: Halting Problem

✔	•	✔	✔	•	•

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2	✔	✔	✔	•	•	•
S3	•	✔	•	✔	•	•
S4	•	•	•	•	•	✔
S5	✔	•	✔	•	✔	•
S6	✔	✔	✔	✔	✔	✔

input strings

strings interpreted as source code

Aside: Halting Problem

✔	•	✔	✔	•	•

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2	✔	✔	✔	•	•	•
S3	•	✔	•	✔	•	•
S4	•	•	•	•	•	✔
S5	✔	•	✔	•	✔	•
S6	✔	✔	✔	✔	✔	✔

input strings

strings interpreted as source code

Aside: Halting Problem

S	✔	•	✔	✔	•	•

	s1	s2	s3	s4	s5	s6
s1	•	•	•	•	•	•
s2	✔	✔	✔	•	•	•
s3	•	✔	•	✔	•	•
s4	•	•	•	•	•	✔
s5	✔	•	✔	•	✔	•
s6	✔	✔	✔	✔	✔	✔

input strings

strings interpreted as source code

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

input strings

strings interpreted as source code

S	✔	•	✔	✔	•	•

	S1	S2	S3	S4	S5	S6
S1	•	•	•	•	•	•
S2	✔	✔	✔	•	•	•
S3	•	✔	•	✔	•	•
S4	•	•	•	•	•	✔
S5	✔	•	✔	•	✔	•
S6	✔	✔	✔	✔	✔	✔

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

`s(S)` halts

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

$$\texttt{s(S)}\text{ halts} \xrightarrow{\text{defn of }\texttt{s}}$$

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

`s(S)` halts $\xrightarrow{\text{defn of } \texttt{s}}$ `halts(S,S)` = 0

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

$$\texttt{s(S)}\text{ halts} \xrightarrow{\text{defn of }\texttt{s}} \texttt{halts(S,S)} = 0$$

$\Big\downarrow$ defn of `halts`

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

$$\texttt{s(S)}\text{ halts} \xrightarrow{\text{defn of }\texttt{s}} \texttt{halts(S,S)} = 0 \xrightarrow{\text{defn of }\texttt{halts}} \texttt{s(S)}\text{ doesn't halt}$$

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

$$\begin{tikzcd}
\texttt{s(S)}\text{ halts} \arrow[r, "\text{defn of }\texttt{s}"] & \texttt{halts(S,S)} = 0 \arrow[d, "\text{defn of }\texttt{halts}"] \\
{} & \texttt{s(S)}\text{ doesn't halt} \arrow[l, "\text{defn of }\texttt{s}"']
\end{tikzcd}$$

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

$$\begin{tikzcd}
 & \texttt{s(S)}\text{ halts} \arrow[r, "\text{defn of } \texttt{s}"] & \texttt{halts(S,S)} = 0 \arrow[d, "\text{defn of } \texttt{halts}"] \\
\texttt{halts(S,S)} = 1 & & \texttt{s(S)}\text{ doesn't halt} \arrow[ll, "\text{defn of } \texttt{s}"']
\end{tikzcd}$$

Aside: Halting Problem

```
S =
"int s(char *D) {
   if (halts(D, D))
     while(1);
   else
     return 42;
 }"
```

Key question:
What happens if we run `s(S)`?

$$
\begin{tikzcd}
\texttt{s(S)}\text{ halts} \arrow[r, "\text{defn of }\texttt{s}"] & \texttt{halts(S,S)} = 0 \arrow[d, "\text{defn of }\texttt{halts}"] \\
\texttt{halts(S,S)} = 1 \arrow[u, "\text{defn of }\texttt{halts}"] & \texttt{s(S)}\text{ doesn't halt} \arrow[l, "\text{defn of }\texttt{s}"']
\end{tikzcd}
$$

Aside: Halting Problem

- **Task.** Write a program `halts` with the following declaration:

  ```
  int halts(char *P, char *D);
  ```

  If the program represented by the string `P` always terminates when run on the input string `D`, then `halts` should return 1. Otherwise it should return 0.

- **Examples:**

```
S1 = "int s1(char *D) {
        while(1);
     }"
```

`halts(S1, _)` = 0

```
S6 = "int s6(char *D) {
        return 42;
     }"
```

`halts(S6, _)` = 1

Aside: Halting Problem

- **Task.** Write a pro[illegible]alts w[illegible]ollowing declaration:

  ```
  int halts(char [illegible]D);
  ```

  If the program repres[illegible]ing `P` always terminates when run on the in[illegible]ng `D`, [illegible]lts should return 1. Otherwise it sh[illegible]rn 0.

- **Examples:**

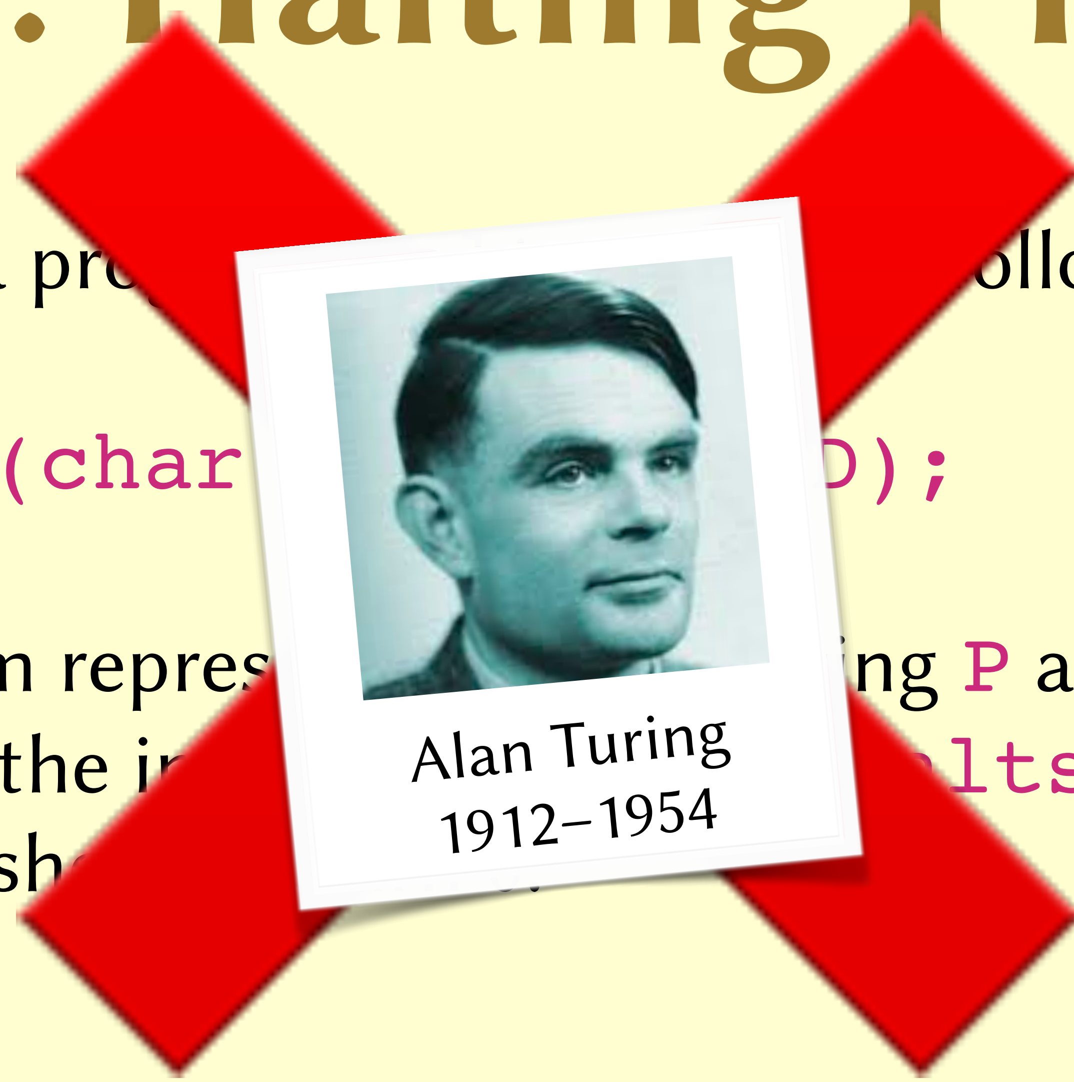
```
S1 = "int s1(char *D) {
        while(1);
     }"
```

`halts(S1, _)` = 0

```
S6 = "int s6(char *D) {
        return 42;
     }"
```

`halts(S6, _)` = 1

Aside: Halting Problem



- **Task.** Write a pr … ollowing declaration:

  `int halts(char` … `D);`

  If the program repres … ng `P` always terminates when run on the i … `lts` should return 1. Otherwise it sh …

- **Examples:**

```
S1 = "int s1(char *D) {
        while(1);
      }"
```

`halts(S1, _)` = 0

```
S6 = "int s6(char *D) {
        return 42;
      }"
```

`halts(S6, _)` = 1

Automatic proof

Automatic proof

- We often rely on automatic provers:

Automatic proof

- We often rely on automatic provers:
    - e.g. in Dafny, to show that `invariant P` is preserved,

Automatic proof

- We often rely on automatic provers:
  - e.g. in Dafny, to show that `invariant P` is preserved,
  - e.g. in Isabelle methods like `by auto`.

Automatic proof

- We often rely on automatic provers:
  - e.g. in Dafny, to show that **`invariant`** `P` is preserved,
  - e.g. in Isabelle methods like **`by`** `auto`.
- How do these automatic provers work?