The Tree Method

1. Write down each premise
2. Negate the conclusion
3. Look for a counterexample
   a) If all leaves are blocked off: no counterexample exists
   b) Else, if some leaf is not blocked off but all compound propositions are checked off: counterexamples exist
   c) If there an unchecked compound proposition, expand it **at every leaf below** and check off the proposition
   d) Close off every leaf whose path to the top contains a contradiction

Why negate the conclusion?

Suppose we don't negate the conclusion ….

If any leaf reveals a contradiction, then the argument is invalid.

If every leaf remains alive, then the argument is valid.

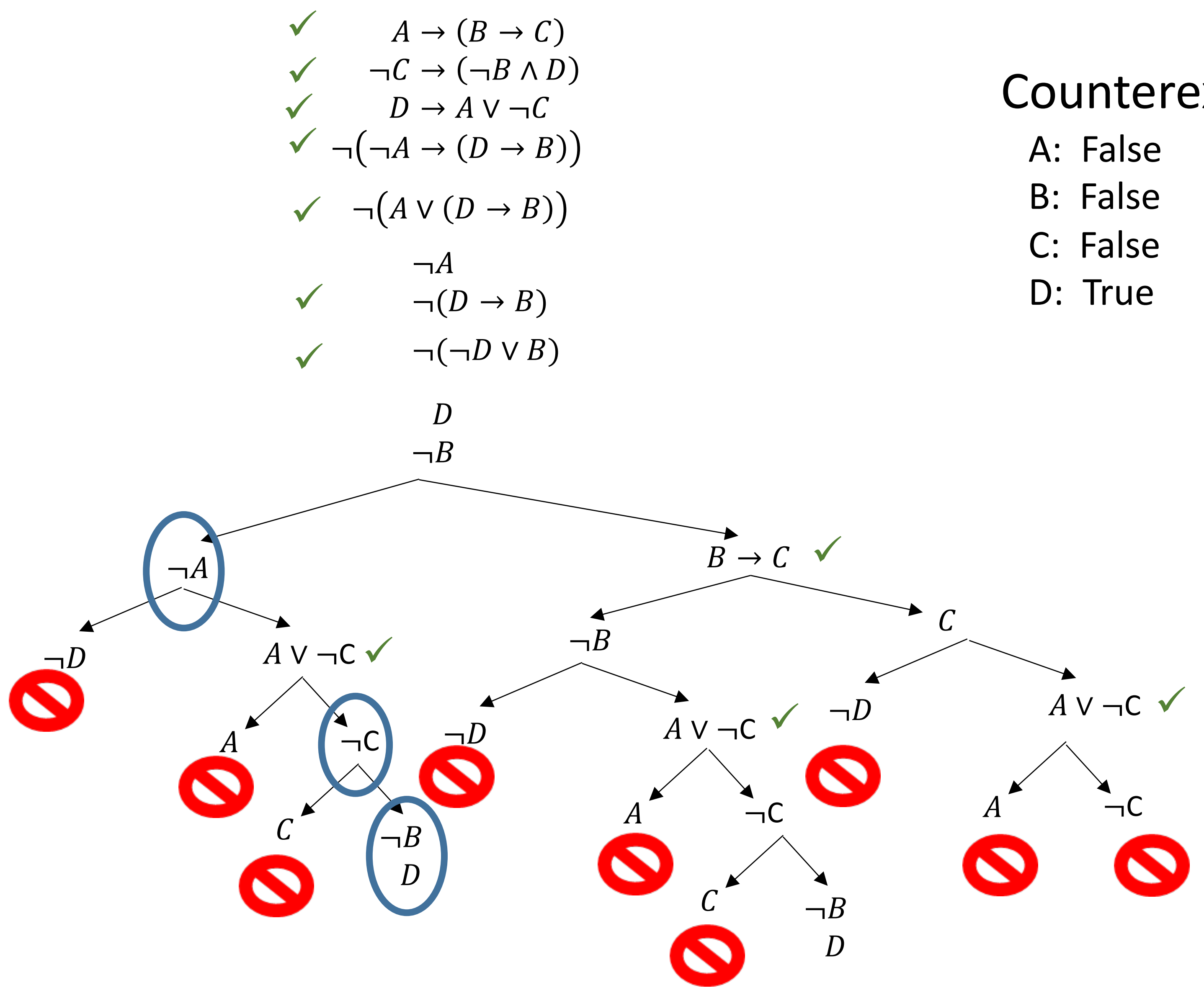
Would this be wrong?

$$\begin{array}{l} \quad H \\ \underline{\neg H \qquad} \\ \therefore\ C \end{array}$$

This argument IS valid, but the root doesn't remain alive!

An Example

$$A \to (B \to C)$$
$$\neg C \to (\neg B \land D)$$
$$\underline{D \to A \lor \neg C}$$
$$\therefore \neg A \to (D \to B)$$



Counterexample!

A: False

B: False

C: False

D: True

One Philosopher's Argument

1. If God exists, then God is omnipotent. $E \Rightarrow OP$
2. If God exists, then God is omniscient. $E \Rightarrow ON$
3. If God exists, then God is benevolent. $E \Rightarrow B$
4. If God can prevent evil, then if God knows that evil exists then God is not benevolent if God does not prevent evil. $P \Rightarrow (K \Rightarrow (\neg Z \Rightarrow \neg B))$
5. If God cannot prevent evil, then God is not omnipotent. $\neg P \Rightarrow \neg OP$
6. If God is omniscient, then God knows that evil exists if it does indeed exist. $ON \Rightarrow (X \Rightarrow K)$
7. Evil does not exist if God prevents it. $Z \Rightarrow \neg X$
8. Evil exists. X

∴ God does not exist $\therefore \neg E$

- E = God exists
- OP = God is omnipotent
- ON = God is omniscient
- B = God is benevolent
- P = God can prevent evil
- K = God knows evil exists
- X = Evil exists
- Z = God prevents evil

Is this argument valid?

$E \Rightarrow OP$ ✓

$E \Rightarrow ON$ ✓

$E \Rightarrow B$ ✓

$P \Rightarrow (K \Rightarrow (\neg Z \Rightarrow \neg B))$ ✓

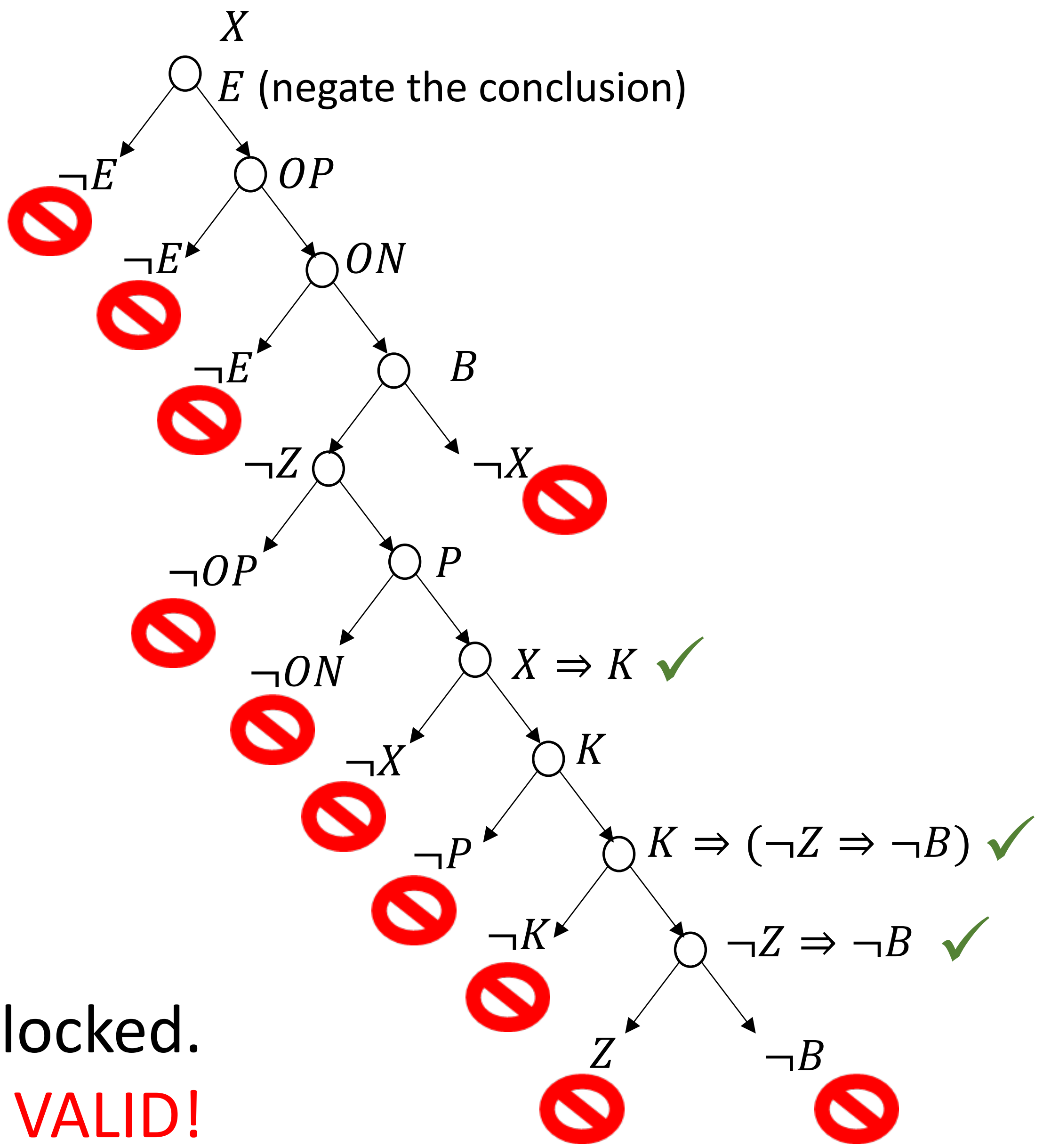
$\neg P \Rightarrow \neg OP$ ✓

$ON \Rightarrow (X \Rightarrow K)$ ✓

$Z \Rightarrow \neg X$ ✓

X

$\therefore \neg E$



All paths blocked.
Therefore, VALID!

Is an inference valid?

1. Use the rules of inference, starting with the premise and try to derive the conclusion.

2. Build a truth table.

3. Apply the tree method to search for counterexamples.

Scheme

Scheme is a child of the first functional language LISP

LISP = **LIS**t **P**rocessing

Our goal is not to learn Scheme, but rather to use its simplest, most basic features to construct recursive programs.

Scheme Identifiers

Numbers: Integers, rationals, …

Boolean: `#t`, `#f`

Identifiers: any sequence of letters, digits, characters (+, %, $, …)

but not including whitespace or `( ) [ ] { } " , ' ` ` ` ; # | \`

and are not numbers

Examples of Identifiers:

```
Isthis1identifier?
pass/fail
a-b+c*2-3-4
```

We'll use identifiers to name variables and functions.

There are some special identifiers which we should not use as variable names.

Scheme Expressions

Two kinds of expressions:

ATOMS: constant (number, boolean value), variable

e.g. `25`, `x`, `whome?`, …

LISTS: a structure made up of atoms and lists

e.g. `(1 2 3)`, `(1 a b me)`, `(+ 3 5)`, …

also `(0 (1 2) (a (b) c) (d (e (f))))`

which is a list of 4 elements, 3 of them lists

Evaluating Expressions

Every Scheme expression has a value.

When you enter an expression, the interpreter prints out its value, and gives an error if the expression does not have a value.

ATOMS: e.g. The value of a number is the number itself.

The value of `#t` is `#t`.

The value of any variable `x` is whatever it was defined as.

To bind a numeric value to a variable, we enter

```
(define x 5)
(define y 17)
```

NOTE: We could also enter `(define define 2)` but not a good idea!

Evaluating List Expressions

Given a list:

`(f a b c)`

The interpreter will:

1. Check if the first element is the name of a defined function.

   If not, the interpreter gives an error.

   e.g. `(1 2 3)`, `(1 + 2)`

2. Evaluate each of the remaining elements of the list.

   If any of the arguments is undefined, then return an error.

3. Apply the function `f` (as defined) to the values returned in Step 2.

Evaluating List Expressions

Examples:

`(+ 1 3)` returns 4, since `+` is a defined function.

`(* 2 4)` returns 8, since `*` is a defined function.

What is `(+ (* 2 5) (* 3 4))`?

= `(+ 10 12)`

= 22

Defining New Functions

How do we create a function that adds the squares of its two arguments? $f(x, y) = x^2 + y^2$

Define the function using the expression:

$$\texttt{(define }\underbrace{\texttt{(f x y)}}_{\substack{\text{function name and}\\ \text{list of arguments}}}\ \underbrace{\texttt{(+ (* x x) (* y y))}}_{\text{function body}}\texttt{)}$$

This binds the function name `f` to the function we desire.

Creating Lists

What expression returns the list `(a b c)`?

Answer: `'(a b c)` which is shorthand for `(quote (a b c))`

The function `quote` suppresses evaluation of its argument.

Manipulating Lists

How do we extract the first element from the list `(a b c)`?

Answer: `(car '(a b c))` returns `a`

The function `car` returns the first element of a list.

The function `cdr` returns the rest of the list, (i.e., the list minus its first element).

`(cdr '(a (b c) d))` returns `((b c) d)`

Manipulating Lists

How do we add an element x to $(a\ b\ c)$ to create $(x\ a\ b\ c)$?

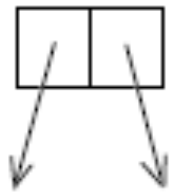
Answer: $\left(\text{cons}\ \ '\text{x}\ \ '(\text{a b c})\right)$ returns (x a b c).

The function $cons$ inserts the first argument into the second argument which is a list.

Why these bizarre function names car, cdr, cons ?

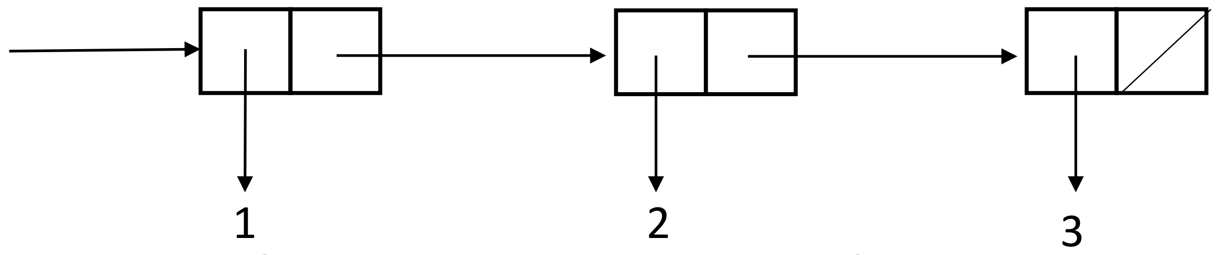
CAR, CDR, CONS

A list is stored in memory as a sequence of constructor boxes.



The left arrow points to the car.
The right arrow points to the cdr.
The arrows represent pointers.

The list (1 2 3) is stored as (cons 1 (cons 2 (cons 3 ‘())))

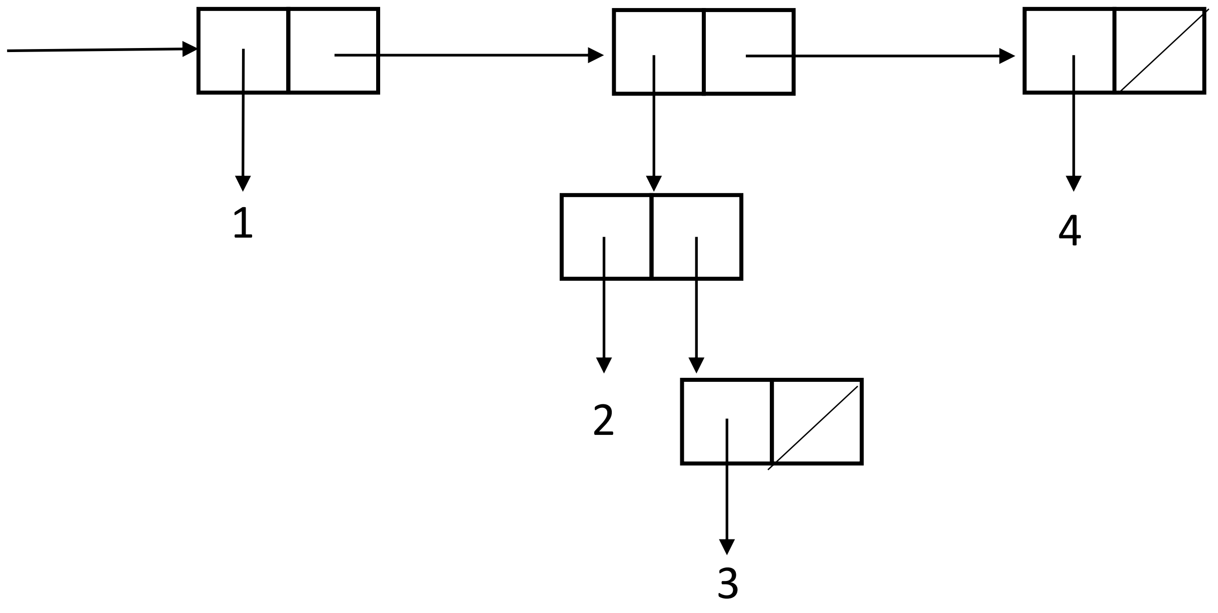


car: contents of the address part of register

cdr: contents of the decrement part of register

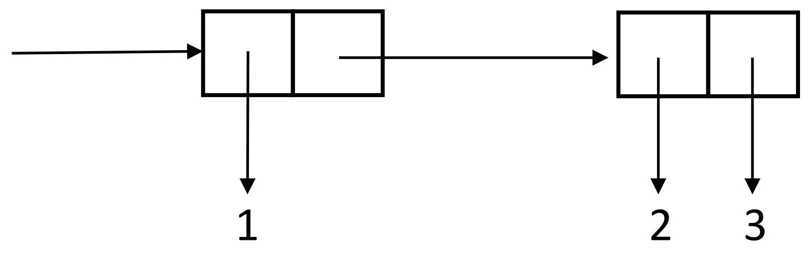
CAR, CDR, CONS

(1 (2 3) 4) is stored as (cons 1 (cons   (cons 2 (cons 3 ‘())) (cons 4 ‘())))



Box-Pointer Diagrams

Box-Pointer Diagrams



This represents the expression (1 2 . 3)

If you see an unexpected “ .” in your output list, check carefully and you will find that the second argument to a cons expression was an atom and not a list!