CS 320: Principles of Programming Languages

Mark P Jones, Portland State University

Spring 2019

Week 1: Introduction - Syntax and Semantics

Please review the course syllabus!

The course syllabus is available:

- In the "General Information" section of D2L Course Content
- On the web at http://web.cecs.pdx.edu/~mpj/cs320/

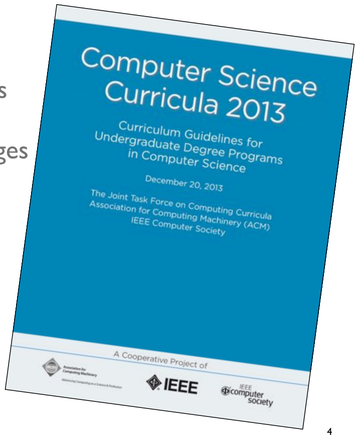
(Two different ways to access the same content!)

Please review the syllabus and be ready to raise any questions that you have about it at the start of the lecture next Tuesday.

Why?

Why Study Programming Languages?

- Because it's a great subject!
- Because it's a required class?
- Because professional societies recommend and expect the study of programming languages as a key component of an undergraduate CS degree



Computer Science Curricula 2013

Curriculum Guidelines for Undergraduate Degree Programs in Computer Science

December 20, 2013

The Joint Task Force on Computing Curricula
Association for Computing Machinery (ACM)
IEEE Computer Society

A Cooperative Project of

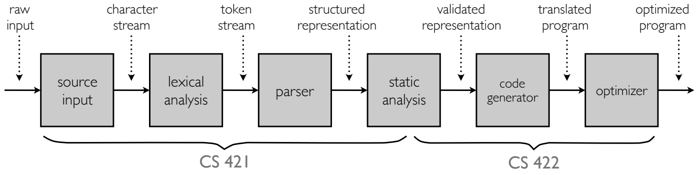
Why Study Programming Languages?

Because languages are everywhere!

- Program development
- Web development
- Gaming
- Configuration files and scripting
- Music
- Art
- …

Why Study Programming Languages?

Because compilers are interesting pieces of software:



- Programming language foundations
- Insights into compiler behavior
- Experience with compiler construction and tools
- Technical depth: LL and LR parsing, assembly, LLVM, etc...
- CS 320 provides a foundation for further study ...

Why Study Programming Languages?

Because a foundation in programming languages is useful in practice:

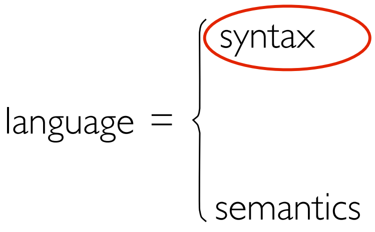
- Learn general concepts, notations, and tools
- Improve your understanding of compiler behavior
- Improve your programming skills
- Improve your ability to learn and work with new languages

Why Study Programming Languages?

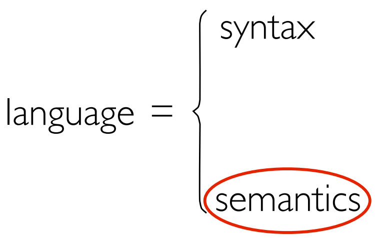
Because language design is still important:

- Languages empower developers to:
  - Express their ideas more directly
  - Execute their designs on a computer
- The long term goal is to develop better languages (and tools) that:
  - Open programming to more people and more applications
  - Increase programmer productivity
  - Enhance software quality (functionality, reliability, security, performance, power, ...)

Syntax and Semantics



syntax: the written/spoken/symbolic/physical form; how things are communicated



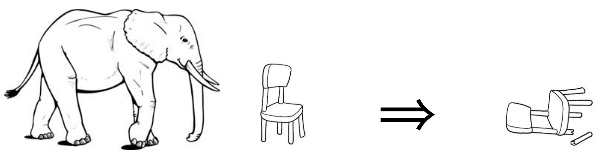
semantics: what the syntax means or represents

Syntax and Semantics

(written form) (meaning)

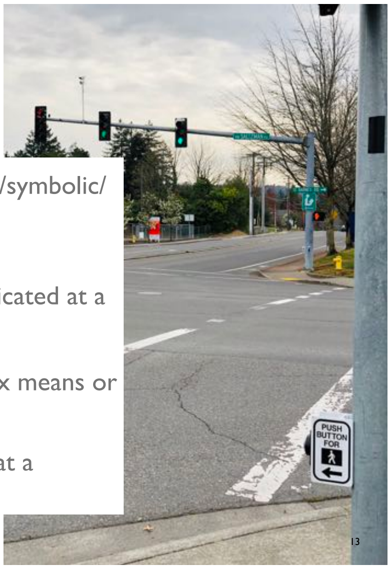
“The elephant
sat on a chair”

⇓



Quick quiz

What is the role of *syntax* and *semantics* at a crosswalk?

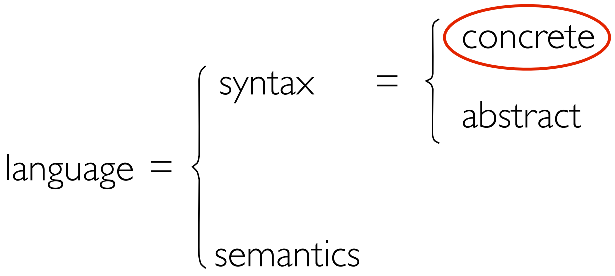


syntax: "the written/spoken/symbolic/ physical form; how things are communicated"

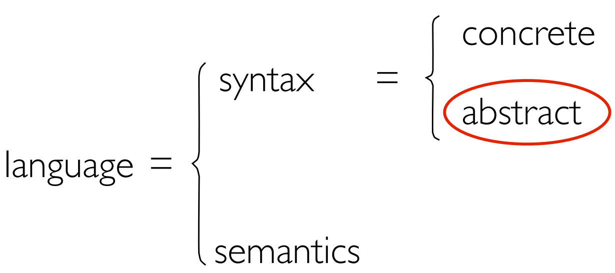
- How are things communicated at a crosswalk?

semantics: "what the syntax means or represents"

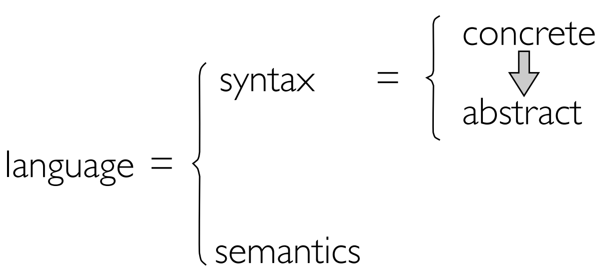
- What does the "syntax" at a crosswalk mean?



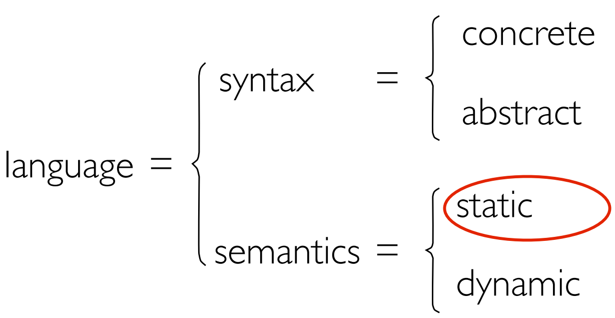
concrete syntax: the representation of a program text in its source form as a sequence of bits/bytes/characters/lines



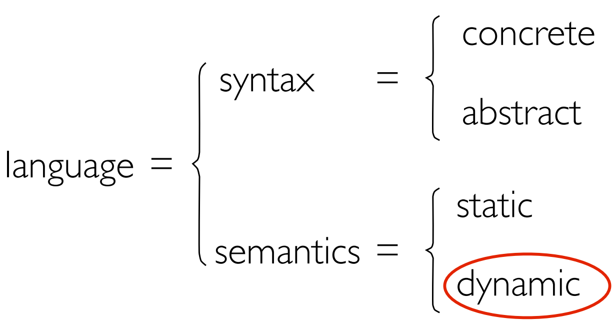
abstract syntax: the representation of a program structure, independent of written form



syntax analysis: This is one of the areas where theoretical computer science has had major impact on the practice of software development



static semantics: those aspects of a program's behavior/meaning that can/must be checked at compile time



dynamic semantics: the behavior of a program at runtime

Example: "Propositional Logic" (also known as "Digital Logic")

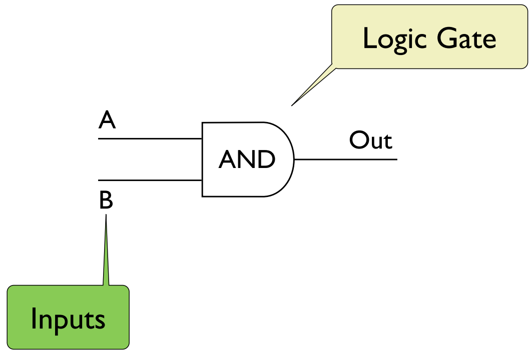
Why study propositional/digital logic?

- It is relatively simple and familiar
- It has a strong mathematical foundation (e.g., CS 251)
- It provides a foundation for understanding computer hardware
- It plays an essential role in practical programming languages (boolean-valued expressions, if and while statements, etc…)

- We'll introduce a lot of terminology along the way, but this is just an introduction, and we'll be coming back to define these terms more precisely as the class proceeds …

Basic Building Blocks

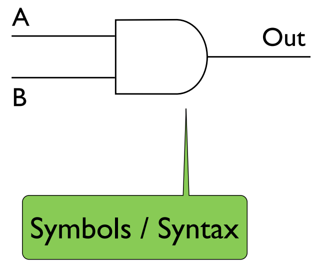
… and a small taste of Haskell



Logic Gate

Inputs

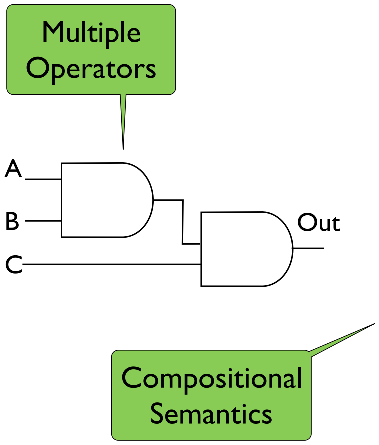
Truth Table



Symbols / Syntax

A	B	Out
False	False	False
False	True	False
True	False	False
True	True	True

Meaning / Semantics

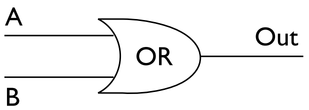


Multiple Operators

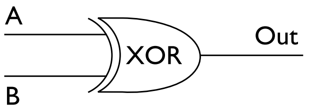
Compositional Semantics

A	B	C	Out
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	False
True	False	False	False
True	False	True	False
True	True	False	False
True	True	True	True

The semantics of the whole is determined by the semantics of the parts

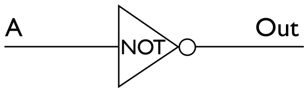


Binary Operators



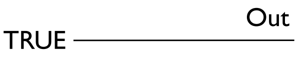
A	B	Out
False	False	False
False	True	True
True	False	True
True	True	True

A	B	Out
False	False	False
False	True	True
True	False	True
True	True	False



A	Out
False	True
True	False

Unary Operator

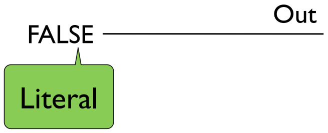


Out
True

Arity

0 ⇒ constant
1 ⇒ unary
2 ⇒ binary
3 ⇒ ternary
...

Constants



Out
False

Value

What is this?

False

Dark pixels on a light background

A collection of lines/strokes

A sequence of characters

A single word (“token”)

An expression

A boolean expression

A truth value

One thing can be seen in many different ways

A change of font/symbol can sometimes help us to distinguish between multiple interpretations …

Diagrams are concrete syntax

- Our diagrams provide an intuitive, graphical description of a logical circuit or formula
- But the diagrams contain many superfluous details:
  - the exact placement of the various components
  - the amount of space between the components
  - the length and shape (and color) of each wire
- Our diagrams are concrete syntax: they provide a notation for writing and communicating
- For abstract syntax, we can ignore a lot of the details and focus on the essential structure of each circuit

Abstracting from unimportant details

Every circuit with one output is exactly one of the following:

- An AND, OR, or XOR gate, whose inputs are the outputs of two (smaller) circuits
- A NOT gate, whose input is the output of a (smaller) circuit
- A TRUE or FALSE literal
- A parameter, identified by a name (i.e., a string of characters)

Every circuit with one output fits exactly one of these descriptions

Every circuit with multiple outputs can be represented as a set of circuits, one per output

A possible abstract syntax

- Using the notation of Haskell

```
data Prop = AND Prop Prop
          | OR  Prop Prop
          | NOT Prop
          | TRUE
          | FALSE
          | VAR String
```

- (Technically speaking, this is now an "Embedded Domain Specific Language" or EDSL)

Example

- What does this circuit look like?

```
OR (AND (NOT (VAR "A")) (VAR
"B")) (AND (VAR "C") (VAR
"D"))
```

Example

- What does this circuit look like?

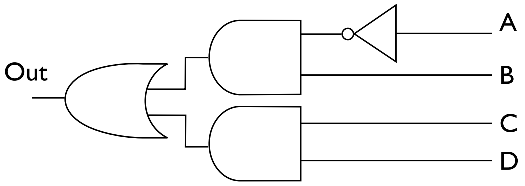
```
OR (AND (NOT (VAR "A"))
        (VAR "B"))
   (AND (VAR "C")
        (VAR "D"))
```

- This is an example of "prefix" notation, where the operator symbol is written in front of the operands
- (compare with "infix" notation, where operators are written *between* operands)

Example

- What does this circuit look like?

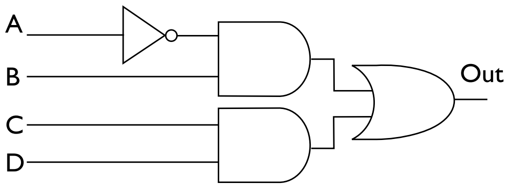
```
OR (AND (NOT (VAR "A"))
        (VAR "B"))
   (AND (VAR "C")
        (VAR "D"))
```



Example

- What does this circuit look like?

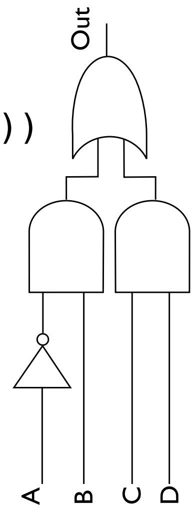
```
OR (AND (NOT (VAR "A"))
        (VAR "B"))
   (AND (VAR "C")
        (VAR "D"))
```



Example

- What does this circuit look like?

```
OR (AND (NOT (VAR "A"))
        (VAR "B"))
   (AND (VAR "C")
        (VAR "D"))
```

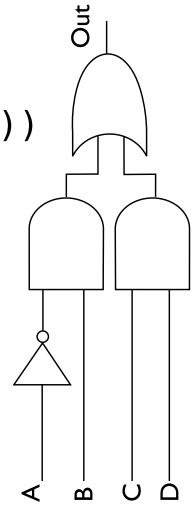


Example

- What does this circuit look like?

```
OR (AND (NOT (VAR "A"))
        (VAR "B"))
   (AND (VAR "C")
        (VAR "D"))
```

- This is an **abstract syntax tree (AST)**

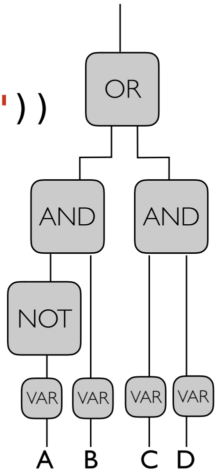


Example

- What does this circuit look like?

```
OR (AND (NOT (VAR "A"))
        (VAR "B"))
   (AND (VAR "C")
        (VAR "D"))
```

- This is an **abstract syntax tree (AST)**



Computing over abstract syntax trees

- Once we represent circuits as data structures, we can write programs to manipulate them or compute properties of them:

```
vars           :: Prop -> [String]
vars (AND p q)  = vars p ++ vars q
vars (OR p q)   = vars p ++ vars q
vars (NOT p)    = vars p
vars TRUE       = []
vars FALSE      = []
vars (VAR v)    = [v]
```

- How do compilers and interpreters work? By performing computations on ASTs!

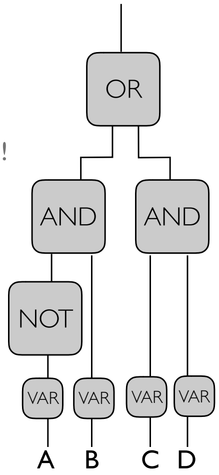
Evaluation

Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

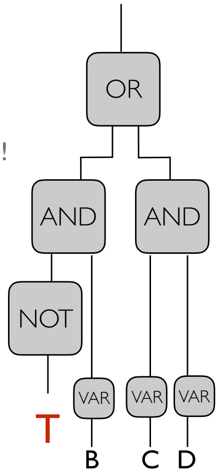


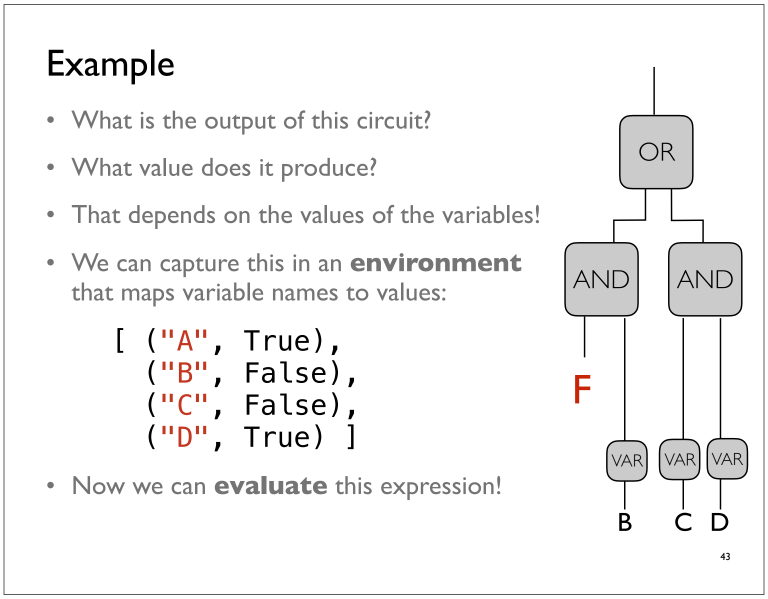
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!



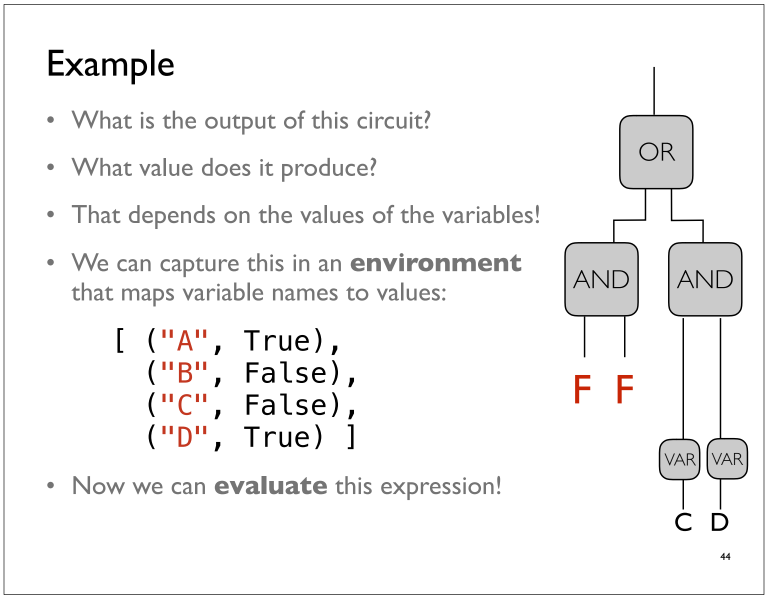


Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

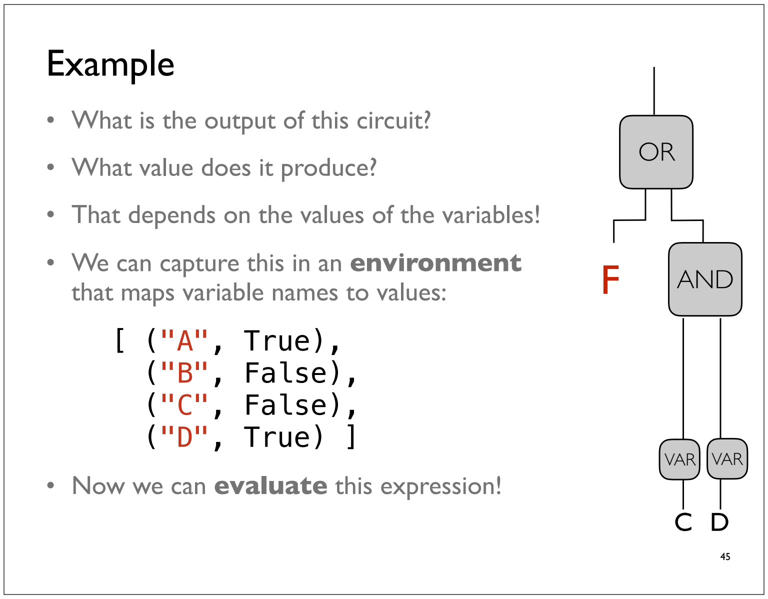


Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

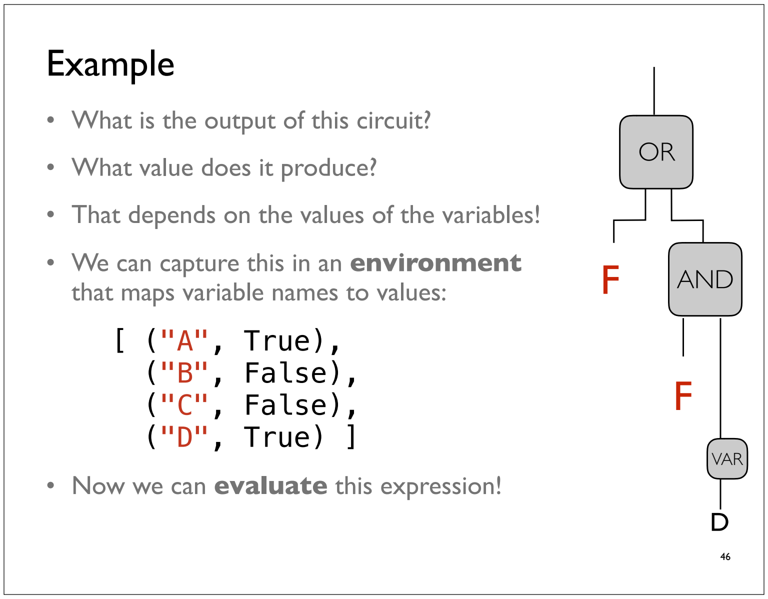


Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

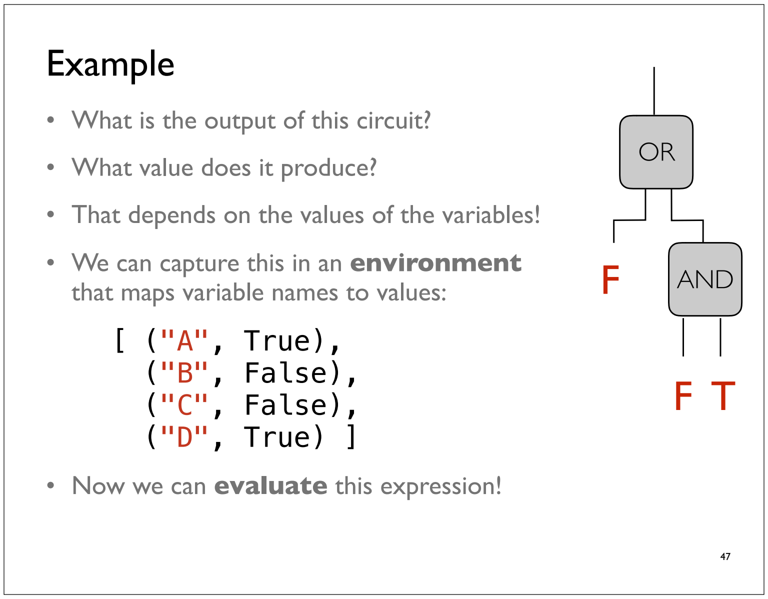


Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

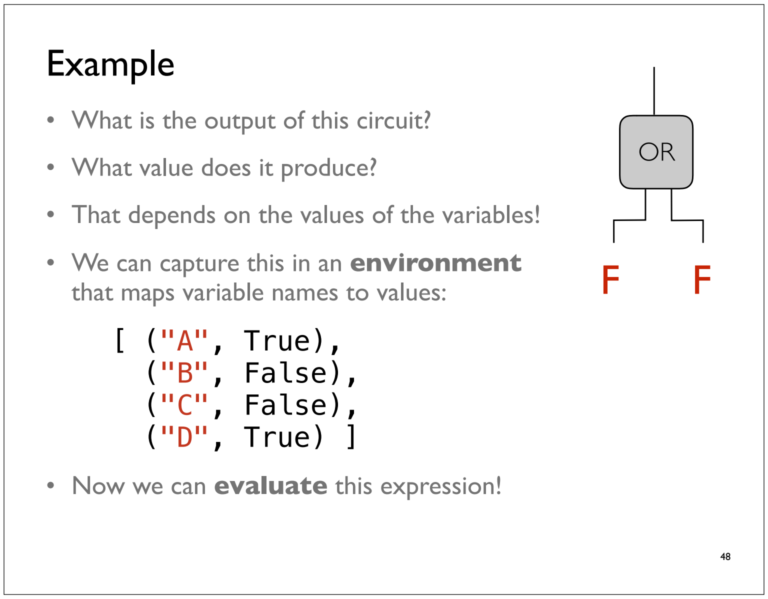


Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!



Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the variables!
- We can capture this in an **environment** that maps variable names to values:

F

```
[ ("A", True),
  ("B", False),
  ("C", False),
  ("D", True) ]
```

- Now we can **evaluate** this expression!

Reduction and normalization

- What we have just seen is a **reduction** of an expression to a **normal form** where no more reductions are possible
- In the notation of abstract syntax:

```
OR (AND (NOT a) b) (AND c d)
⟹ OR (AND (NOT TRUE) b) (AND c d)
⟹ OR (AND FALSE b) (AND c d)
⟹ OR (AND FALSE FALSE) (AND c d)
⟹ OR FALSE (AND c d)
⟹ OR FALSE (AND FALSE d)
⟹ OR FALSE (AND FALSE TRUE)
⟹ OR FALSE FALSE
⟹ FALSE
```

Formal notation for reductions

- Reduction of an expression requires an environment to provide values for any variables that it contains
- We sometimes write
  - **env** ⊢ $\mathbf{E_1} \Longrightarrow \mathbf{E_2}$

    to mean that the expression E_1 can be reduced in **exactly one step** to the expression E_2
  - **env** ⊢ $\mathbf{E_1} \Longrightarrow^* \mathbf{E_2}$

    to mean that the expression E_1 can be reduced in **zero or more steps** to the expression E_2
- In each case, reduction steps may use the values for variables that are specified in the environment env

Evaluation strategies

- The process of calculating a normal form for an expression is referred to as **normalization**
- Other **evaluation strategies** are possible:
  - left-to-right or right-to-left or …
  - strict or lazy
  - deterministic or non-deterministic
- Interesting questions to ask:
  - does the normalization process always terminate?
  - do different reduction orders produce the same result?
  - how many steps does it take to normalize a given expression?
- Normalization gives an **operational semantics** for the language

An alternative to operational semantics

- A **denotational semantics** is a function that maps abstract syntax trees to meanings:

```
eval                :: Env -> Prop-> Bool
eval env (AND p q) = eval env p && eval env q
```

An alternative to operational semantics

- A **denotational semantics** is a function that maps abstract syntax trees to meanings:

```
eval                :: Env -> Prop-> Bool
eval env (AND p q)  = eval env p && eval env q
eval env (OR p q)   = eval env p || eval env q
```

An alternative to operational semantics

- A **denotational semantics** is a function that maps abstract syntax trees to meanings:

```
eval                :: Env -> Prop-> Bool
eval env (AND p q) = eval env p && eval env q
eval env (OR p q)  = eval env p || eval env q
eval env (NOT p)   = not (eval env p)
```

An alternative to operational semantics

- A **denotational semantics** is a function that maps abstract syntax trees to meanings:

```
eval                :: Env -> Prop-> Bool
eval env (AND p q) = eval env p && eval env q
eval env (OR p q)  = eval env p || eval env q
eval env (NOT p)   = not (eval env p)
eval env TRUE      = True
```

An alternative to operational semantics

- A **denotational semantics** is a function that maps abstract syntax trees to meanings:

```
eval                :: Env -> Prop-> Bool
eval env (AND p q) = eval env p && eval env q
eval env (OR p q)  = eval env p || eval env q
eval env (NOT p)   = not (eval env p)
eval env TRUE      = True
eval env FALSE     = False
```

An alternative to operational semantics

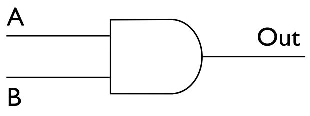
- A **denotational semantics** is a function that maps abstract syntax trees to meanings:

```
eval                :: Env -> Prop-> Bool
eval env (AND p q) = eval env p && eval env q
eval env (OR p q)  = eval env p || eval env q
eval env (NOT p)   = not (eval env p)
eval env TRUE      = True
eval env FALSE     = False
eval env (VAR v)
   = case lookup v env of
          Just b   -> b
          Nothing -> error ("No defn for " ++ v)
```

Equivalences

Axiomatic semantics

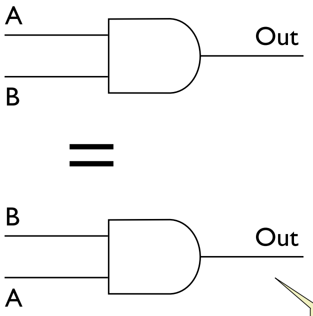
- Can we say anything about the meaning of a given circuit or expression:
  - without knowing the values of all the variables that it contains?
  - without fully evaluating it?
- In certain circumstances, yes!
- These techniques can be useful in practice for:
  - **optimization** to produce more efficient circuits (faster, smaller, lower power, cheaper, etc.)
  - constructing **proofs** of program correctness properties



A	B	Out
False	False	False
False	True	False
True	False	False
True	True	True

Distinct abstract syntax trees, but the same semantics

Equivalences

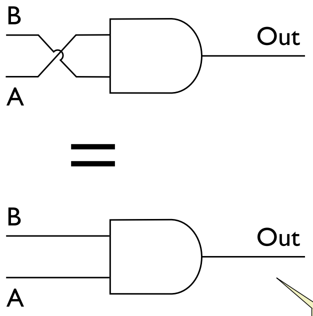


Commutativity

A	B	Out
False	False	False
False	True	False
True	False	False
True	True	True

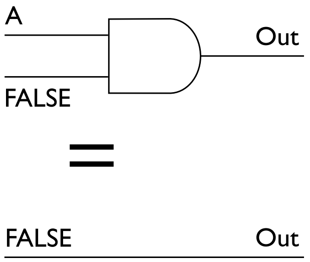
Could eliminate the need for crossed wires

Equivalences



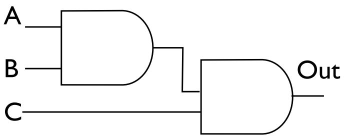
Commutativity

A	B	Out
False	False	False
False	True	False
True	False	False
True	True	True



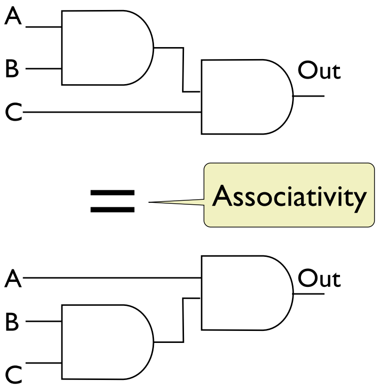
Could eliminate redundant circuit components

A	Out
False	False
True	False



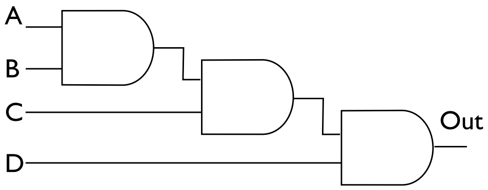
A	B	C	Out
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	False
True	False	False	False
True	False	True	False
True	True	False	False
True	True	True	True

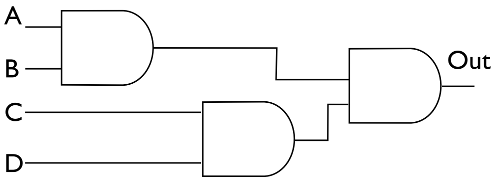
How might this be useful?

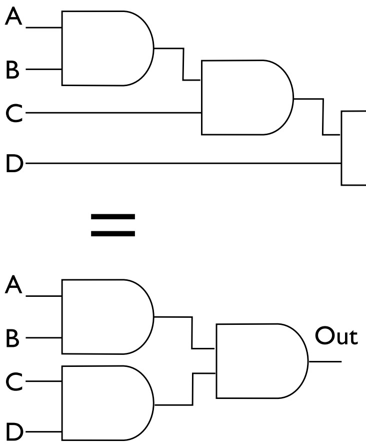


Associativity

A	B	C	Out
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	False
True	False	False	False
True	False	True	False
True	True	False	False
True	True	True	True







Could reduce "gate delay"

In terms of abstract syntax

- Two expressions are equivalent if they have the same value
- With an operational semantics:
    - If env $\vdash E_1 \Longrightarrow^* E$ and env $\vdash E_2 \Longrightarrow^* E$ for some E and with any environment env, then the two expressions E_1 and E_2 are equivalent
- With a denotational semantics:
    - If eval E_1 env = eval E_2 env for any environment env, then the two expressions E_1 and E_2 are equivalent

What is missing?

- We have described a complete **language**, including both its syntax and its semantics
- What features of a practical **programming language** are we missing?
    - variables: for holding intermediate and shared results
    - abstraction: the ability to name patterns and structures to promote code reuse and to manage complexity
    - control structures: loops, conditionals, recursion, for programmatic construction of circuits
    - types: to classify values and protect against misuse

Summary

- Syntax: From written form (concrete) to structure (abstract)
- Semantics: provides a meaning for the syntax
    - Reduction/normalization ("operational semantics")
    - Mapping to values ("denotational semantics")
    - Equivalences ("axiomatic semantics")
- But ...

  When does a language become a **programming** language?