CS 320: Principles of Programming Languages

Katie Casamento, Portland State University, Fall 2018
Based on slides by Mark P. Jones, Portland State University, Winter 2017

Winter 2018
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 https://web.cecs.pdx.edu/~cas28/320/

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

Why study programming languages?

Why study programming languages?

- 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

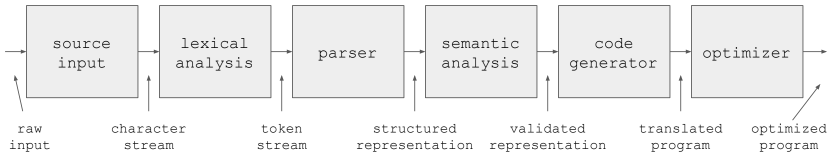
Why study programming languages?

Because programming 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:



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…

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 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

Language = syntax + semantics

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

semantics: what the syntax means or represents

Language = **syntax** + semantics

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: transformation from concrete syntax to abstract syntax

Language = syntax + semantics

static semantics: 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:

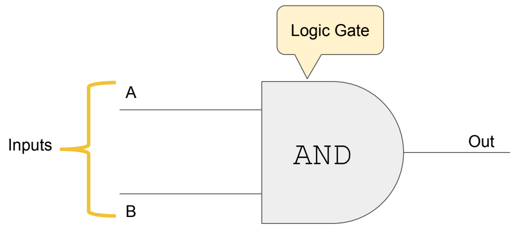
Propositional (or Digital) Logic

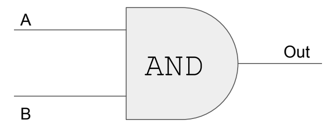
Why study propositional/digital logic?

- It's 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 - we'll be coming back to define these terms more precisely as the class proceeds

Basic Building Blocks

and a taste of Haskell

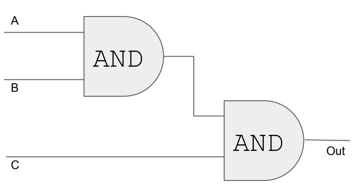




A	B	Out
False	False	False
False	True	False
True	False	False
True	True	True

Symbols/Syntax

Meaning/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

Multiple operators

Compositional semantics

(the semantics of the whole is determined by the semantics of the parts)

Terminology

- Constant: element with no inputs
  - Literal: symbol representing a constant ("TRUE", "FALSE")
  - Value: physical bit flowing through a wire (high signal, low signal)
- Operator: element that transforms input values into output value
  - Unary operator: one-input operator (e.g. NOT)
  - Binary operator: two-input operator (e.g. AND, OR)
- Arity: number of inputs to an operator
  - 0 => constant
  - 1 => unary
  - 2 => binary
  - 3 => ternary
  - …

What is this?

False

- Dark pixels on a light background
- A collection of lines/strokes
- A sequence of characters
- A single word ("token")
- A boolean expression
- A truth value

One thing can be seen in many different ways

Diagrams are concrete syntax

- 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 components
  - the length and shape (and color) of each wire
- Diagrams provide a notion 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 away unimportant details

Every circuit with one output is exactly one of the following (for our purposes):

- An AND or OR 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
- An input to the circuit, identified by a (string) name

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.

Quick aside: Haskell in this course

- Install the Haskell Platform at https://www.haskell.org/downloads#platform (or use the departmental Linux computers, which already have it)
- Read Prof. Jones' Haskell Quick Reference document on D2L or at https://web.cecs.pdx.edu/~cas28/320/HaskellQuickReference.html
- Ask for help!

A possible abstract syntax

keyword
type
constructor

Diving into a bit of Haskell:

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

- AND, OR, etc. are *constructors*, but that word's not used in quite the same sense as in OOP - just in the sense that they construct larger values out of smaller values
- Constructors are *applied* to arguments
- Prop is an *algebraic datatype* (we'll come back to these in a few weeks)
- "|" is pronounced "or"
- "a Prop is 'TRUE' or 'FALSE' or 'IN' applied to a String or 'AND' applied to two Props or 'OR' applied to two Props or 'NOT' applied to a Prop"

Example

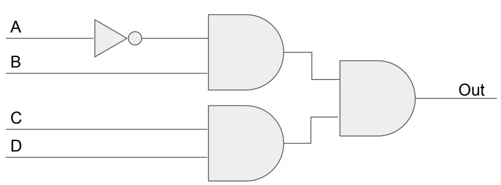
```
AND (AND (NOT (IN "A")) (IN "B")) (AND (IN "C") (IN "D"))
```

These Prop expressions are written in *prefix* notation, meaning the operator symbol is in front of the operands, in contrast to *infix* notation where the operator is between the operands (e.g. "A && B").

Example

What does this circuit look like?

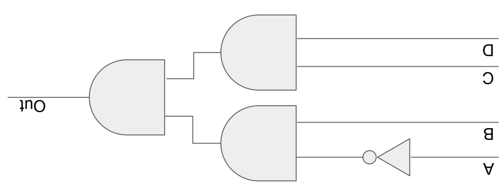
```
AND (AND (NOT (IN "A")) (IN "B")) (AND (IN "C") (IN "D"))
```



Example

What does this circuit look like?

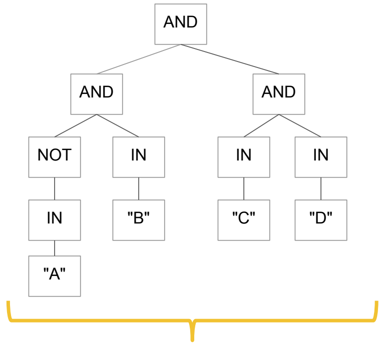
```
AND (AND (NOT (IN "A")) (IN "B")) (AND (IN "C") (IN "D"))
```



Example

What does this circuit look like?

```
AND
  (AND
    (NOT
      (IN "A"))
    (IN "B"))
  (AND
    (IN "C")
    (IN "D"))
```



Abstract syntax tree (AST)

Computing over abstract syntax trees

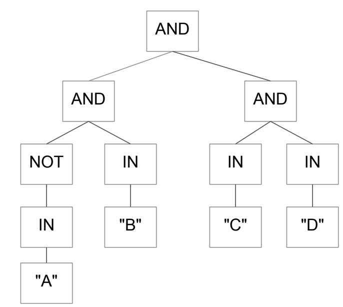
Once we represent circuits as data structures, we can write programs to manipulate them or compute properties of them:

```
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 (IN v)    = [v]
```

Evaluation

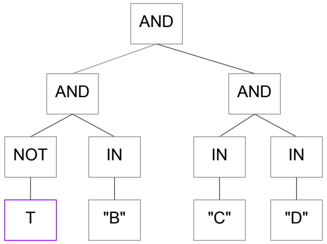
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE, "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



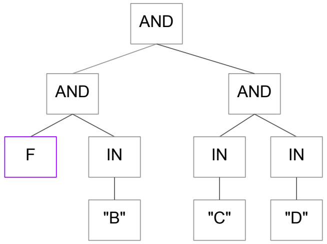
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



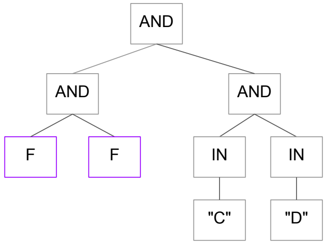
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



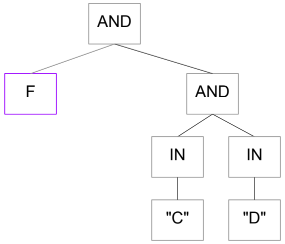
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



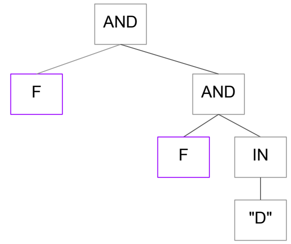
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



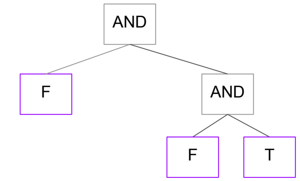
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



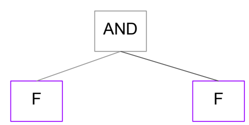
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



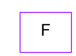
Example

- What is the output of this circuit?
- What value does it produce?
- That depends on the values of the inputs!
- We can capture this in an environment that maps input 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 inputs!
- We can capture this in an environment that maps input names to values:
  `"A" = TRUE,  "B" = FALSE,`
  `"C" = FALSE, "D" = TRUE`
- Now we can *evaluate* this expression!



```
[("A", TRUE), ("B", FALSE),
("C", FALSE), ("D", TRUE)]
```

Reduction and normalization

What we've just seen is a *reduction* of an expression to a *normal form* where no more reductions are possible.

In Haskell notation:

```
AND (AND (NOT (IN "A")) (IN "B")) (AND (IN "C") (IN "D"))
⇒ AND (AND (NOT TRUE) (IN "B")) (AND (IN "C") (IN "D"))
⇒ AND (AND FALSE (IN "B")) (AND (IN "C") (IN "D"))
⇒ AND (AND FALSE FALSE) (AND (IN "C") (IN "D"))
⇒ AND FALSE (AND (IN "C") (IN "D"))
⇒ AND FALSE (AND TRUE (IN "D"))
⇒ AND FALSE (AND TRUE FALSE)
⇒ AND FALSE FALSE
⇒ FALSE
```

Formal notation for reductions

- Reduction of an expression requires an environment to provide values for any identifiers that it contains
- We sometimes write "`env ⊢ E1 ⇒ E2`" to mean that the expression E1 can be reduced in exactly one step to the expression E2 under the environment `env`, and "`env ⊢ E1 ⇒* E2`" to mean that E1 can be reduced in zero or more steps to the expression E2 under `env`
- In each case, reduction steps may use the values for the identifiers that are specified in the environment `env`

Operational semantics

- The process of calculating a normal form for an expression is referred to as *normalization*
- A normalization procedure gives an *operational semantics* for the language
- Other *evaluation strategies* are possible:
  - left-to-right or right-to-left
  - strict or lazy
  - deterministic or nondeterministic
- 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?

Denotational semantics

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

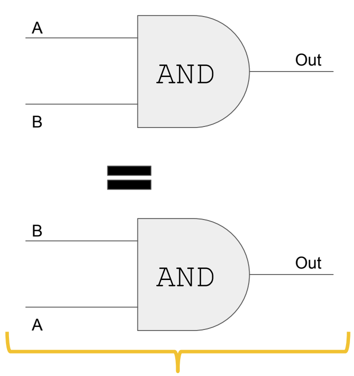
```
type Env = String -> Bool

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

Equivalences

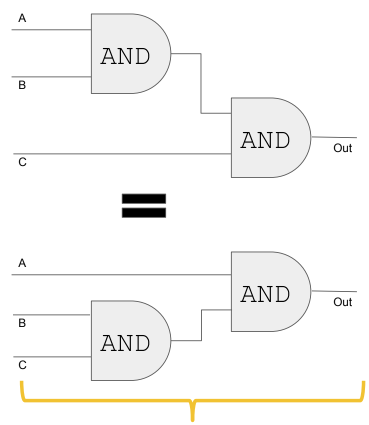
Axiomatic semantics

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



Commutativity

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



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

In terms of abstract syntax

- Two expressions are equivalent if they have the same value
- With an operational semantics:
  - If `env ⊢ E1 ⇒* E` and `env ⊢ E2 ⇒* E` for some `E` and any arbitrary environment `env`, then the two expressions `E1` and `E2` are equivalent
- With a denotational semantics:
  - If `eval E1 env = eval E2 env` for any arbitrary environment `env`, then the two expressions `E1` and `E2` are equivalent

What's missing?

What's missing?

- We've 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 give names to patterns and structures, to promote code reuse and manage complexity
  - control structures: loops, conditionals, recursion, etc. for programmatic construction of circuits
  - types: to classify values and protect against misuse

Variables

Sharing

- Consider the circuit

```
AND
  (OR
    (IN "A")
    (AND (IN "B") (IN "C")))
  (OR
    (AND (IN "B") (IN "C"))
    (IN "D"))
```

- The calculation of `AND (IN "B") (IN "C")` is used as an input to both `OR` gates
- Can we make the expression less redundant?

Sharing

- Consider the circuit

```
AND
  (OR
    (IN "A")
    (AND (IN "B") (IN "C")))
  (OR
    (AND (IN "B") (IN "C"))
    (IN "D"))
```

- Introduce a local variable:

```
let x = AND (IN "B") (IN "C")
in AND (OR (IN "A") x) (OR x (IN "D"))
```

- But this `let` construct is part of Haskell, not part of our Prop language

Abstraction

Abstraction

- The logic gates we've been using are already abstractions of smaller circuits made out of transistors or other logic gates
- We can build an XOR gate out of AND, OR, and NOT gates:
  - `xor p q = OR (AND p (NOT q)) (AND (NOT p) q)`
- Abstraction: giving a name to a (typically recurring) structure or pattern so that it can be reused without copy+pasting the whole thing
- Expands the "vocabulary" of the language
- Essential for managing complexity in large systems

Control Structures

Control structures

- How can we build a generalized AND gate for any arbitrary number of inputs?
- We can construct an N-input AND gate by wiring up multiple standard two-input AND gates
- But we can't express this in our Prop language because there are no constructs for looping or testing conditions

Control structures

keyword
type
constructor
comment

- How can we build a generalized AND gate for any arbitrary number of inputs?
- We can express this in Haskell using a recursive function:

```
andN :: [Prop] -> Prop
andN []     = TRUE             -- because AND p TRUE == p, for any p
andN (p:ps) = AND p (andN ps)  -- p is the head, ps is the tail
```

- For example:

```
andN [IN "A", IN "B", IN "C"]
⇒ AND (IN "A") (andN [IN "B", IN "C"])
⇒ AND (IN "A") (AND (IN "B") (andN [IN "C"]))
⇒ AND (IN "A") (AND (IN "B") (AND (IN "C") (andN [])))
⇒ AND (IN "A") (AND (IN "B") (AND (IN "C") TRUE))
```

Types

Classifying values

- We've already seen several different types of circuits:
  - AND/OR gates have two inputs and one output
  - NOT gates have one input and one output
  - the constants TRUE and FALSE have no inputs and one output
- "`::`" is pronounced "has type"
  - `TRUE               :: Prop`
  - `FALSE              :: Prop`
  - `IN "A"             :: Prop`
  - `IN                 :: String -> Prop`
  - `AND TRUE (IN "A")  :: Prop`
  - `AND                :: Prop -> Prop -> Prop`
- Types are useful for classifying values
- Types are useful for ensuring correct usage

Summary

Summary

- Syntax: from written form (concrete) to structure (abstract)
- Semantics: provides a meaning for the syntax
  - Normalization (operational semantics)
  - Evaluation (denotational semantics)
  - Equivalences (axiomatic semantics)
- When does a language become a programming language?
  - The power of abstraction