Language Specification

First Example: Propositional Logic

Professor William L. Harrison

September 5, 2018

An example

Q: Is the following a legal C program?

```
$ cat helloworld.c
#include <stdio.h>
int main() {
  printf("hello world\n")
}
```

An example

Q: Is the following a legal C program?

```
$ cat helloworld.c
#include <stdio.h>
int main() {
  printf("hello world\n")
}
```

Nope.

```
$ gcc helloworld.c
helloworld.c: In function ‘main’:
helloworld.c:3: error: parse error
                           before ’}’ token
```

Review

- C has some means of expressing and checking structure of an input file that a programmer claims is a program.

Review

- C has some means of expressing and checking structure of an input file that a programmer claims is a program.
- "Context-free Grammar" (CFG): structural rules that determine whether a sequence of symbols is, in fact, a sentence (program) in a language.

Review

- C has some means of expressing and checking structure of an input file that a programmer claims is a program.
- "Context-free Grammar" (CFG): structural rules that determine whether a sequence of symbols is, in fact, a sentence (program) in a language.
- CFGs are expressive enough to describe PL syntax and can be readily adapted to programming (parsing).

Review

- C has some means of expressing and checking structure of an input file that a programmer claims is a program.
- "Context-free Grammar" (CFG): structural rules that determine whether a sequence of symbols is, in fact, a sentence (program) in a language.
- CFGs are expressive enough to describe PL syntax and can be readily adapted to programming (parsing).
- Kernighan & Ritchie (2nd edition, App. 9.2, page 222):

$$\textit{expression-statement} : \ \textit{expression}_{opt} \ ;$$

Review

- C has some means of expressing and checking structure of an input file that a programmer claims is a program.
- "Context-free Grammar" (CFG): structural rules that determine whether a sequence of symbols is, in fact, a sentence (program) in a language.
- CFGs are expressive enough to describe PL syntax and can be readily adapted to programming (parsing).
- Kernighan & Ritchie (2nd edition, App. 9.2, page 222):

$$\textit{expression-statement} : \ \textit{expression}_{opt} \ ;$$

Says "an expression-statement is an expression (in this case the call to printf) followed by a semicolon."

Separating Syntax and Semantics

The ideas and issues which we will consider are:

- What is a language?
- Syntax: How do we define *precisely* what are the well-formed sentences of a language?
- Semantics: Given a well-formed sentence, what does it mean?
- The separation between syntax and semantics.

Solving linear equation: $5x + 7 = 9$

Assume

$$5x + 4 \quad = \quad 9 \qquad \text{(i)}$$

Solving linear equation: $5x + 7 = 9$

Assume

$$5x + 4 = 9 \qquad \text{(i)}$$

Subtract 4 from each side of Equation (i):

$$5x = 5 \qquad \text{(ii)}$$

Solving linear equation: $5x + 7 = 9$

Assume

$$5x + 4 = 9 \qquad \text{(i)}$$

Subtract 4 from each side of Equation (i):

$$5x = 5 \qquad \text{(ii)}$$

Divide both sides of Equation (ii) by 5:

$$x = 1 \qquad \text{(iii)}$$

Two rules

$$
\begin{array}{lcll}
kx + l = m & \Rightarrow & kx = m - l & \text{“subtract from both sides”} \\
kx = l & \Rightarrow & x = l/k & (k \neq 0)\,\text{“divide both sides”}
\end{array}
$$

Two rules

$$\begin{array}{lcll} kx + l = m & \Rightarrow & kx = m - l & \text{"subtract from both sides"} \\ kx = l & \Rightarrow & x = l/k & (k \neq 0)\text{"divide both sides"} \end{array}$$

Consider

$$5x + 4 = 9$$

Two rules

$$\begin{array}{llll} kx + l = m & \Rightarrow & kx = m - l & \text{“subtract from both sides”} \\ kx = l & \Rightarrow & x = l/k & (k \neq 0)\text{“divide both sides”} \end{array}$$

Consider

$$5x + 4 = 9 \quad \Rightarrow \quad 5x = 9 - 4$$

Two rules

$$
\begin{array}{llll}
kx + l = m & \Rightarrow & kx = m - l & \text{“subtract from both sides”} \\
kx = l & \Rightarrow & x = l/k & (k \neq 0)\,\text{“divide both sides”}
\end{array}
$$

Consider

$$5x + 4 = 9 \quad \Rightarrow \quad 5x = 9 - 4 \quad \Rightarrow \quad x = (9 - 4)/5$$

Two rules

$$
\begin{array}{llll}
kx + l = m & \Rightarrow & kx = m - l & \text{“subtract from both sides”} \\
kx = l & \Rightarrow & x = l/k & (k \neq 0)\,\text{“divide both sides”}
\end{array}
$$

Consider

$$5x + 4 = 9 \quad \Rightarrow \quad 5x = 9 - 4 \quad \Rightarrow \quad x = (9 - 4)/5$$

Question: Can we solve $3x + 5 + 6x = 0$ with these rules?

Two rules

$$
\begin{array}{lcll}
kx + l = m & \Rightarrow & kx = m - l & \text{"subtract from both sides"} \\
kx = l & \Rightarrow & x = l/k & (k \neq 0)\,\text{"divide both sides"}
\end{array}
$$

Consider

$$5x + 4 = 9 \quad \Rightarrow \quad 5x = 9 - 4 \quad \Rightarrow \quad x = (9 - 4)/5$$

Question: Can we solve $3x + 5 + 6x = 0$ with these rules?
Answer: No.

Derivation System (High Level)

- Has some notion of a "sentence" or "formula"

$$-(E + -E) \qquad\qquad 2B \vee \neg 2B$$

Derivation System (High Level)

- Has some notion of a "sentence" or "formula"

$$-(E + -E) \qquad\qquad 2B \vee \neg 2B$$

- Rules for producing new formulae from existing ones

$$E \Rightarrow -E \qquad\qquad \frac{\varphi \quad \varphi \supset \gamma}{\gamma}$$

Derivation System (High Level)

- Has some notion of a "sentence" or "formula"

$$-(E + -E) \qquad\qquad 2B \vee \neg 2B$$

- Rules for producing new formulae from existing ones

$$E \Rightarrow -E \qquad\qquad \frac{\varphi \quad \varphi \supset \gamma}{\gamma}$$

- Notion of "proof" or "derivation". Sequence of sentences:

$$S_1, \ldots, S_n$$

where S_i result of applying a rule to (members of) $\{S_1, \ldots, S_{i-1}\}$

Derivation Systems Everywhere

- Defining syntax

$$
\begin{aligned}
&E \Rightarrow i && (\textit{where } i \in \{\ldots, -1, 0, 1, \ldots\}) \\
&E \Rightarrow -\ E
\end{aligned}
$$

Derivation Systems Everywhere

- Defining syntax

$$E \Rightarrow i \qquad (\textit{where } i \in \{\ldots, -1, 0, 1, \ldots\})$$
$$E \Rightarrow - \; E$$

Derivation: E

Derivation Systems Everywhere

- Defining syntax

$$E \Rightarrow i \qquad (\textit{where } i \in \{\ldots, -1, 0, 1, \ldots\})$$
$$E \Rightarrow - \ E$$

Derivation: $E \Rightarrow -E$

Derivation Systems Everywhere

- Defining syntax

$$E \Rightarrow i \qquad (\textit{where } i \in \{\ldots, -1, 0, 1, \ldots\})$$
$$E \Rightarrow - \; E$$

Derivation: $E \Rightarrow -E \Rightarrow - - E$

Derivation Systems Everywhere

- Defining syntax

$$E \Rightarrow i \qquad (\textit{where } i \in \{\ldots, -1, 0, 1, \ldots\})$$
$$E \Rightarrow -\ E$$

Derivation: $E \Rightarrow -E \Rightarrow - - E \Rightarrow - - 9$

Derivation Systems Everywhere

- Defining syntax

$E \Rightarrow i \qquad (\textit{where } i \in \{\ldots, -1, 0, 1, \ldots\})$
$E \Rightarrow -\ E$

Derivation: $E \Rightarrow -E \Rightarrow - - E \Rightarrow - - 9$

$\therefore - - 9$ is an E

Derivation Systems Everywhere

- Defining types

$$\frac{i \in \{\ldots, -1, 0, 1, \ldots\}}{i :: \mathit{Int}} \qquad \frac{e :: \mathit{Int}}{-e :: \mathit{Int}}$$

Derivation Systems Everywhere

- Defining types

$$\frac{i \in \{\ldots, -1, 0, 1, \ldots\}}{i :: Int} \qquad \frac{e :: Int}{-e :: Int}$$

Derivation

$$\cfrac{\cfrac{\cfrac{9 \in \{\ldots, -1, 0, 1, \ldots\}}{9 :: Int}}{-9 :: Int}}{--9 :: Int}$$

Derivation Systems Everywhere

- Defining types

$$\frac{i \in \{\ldots, -1, 0, 1, \ldots\}}{i :: \mathit{Int}} \qquad \frac{e :: \mathit{Int}}{-e :: \mathit{Int}}$$

Derivation

$$\cfrac{\cfrac{\cfrac{9 \in \{\ldots, -1, 0, 1, \ldots\}}{9 :: \mathit{Int}}}{-9 :: \mathit{Int}}}{- - 9 :: \mathit{Int}}$$

$$\therefore - - 9 :: \mathit{Int}$$

Derivation Systems Everywhere

- Defining meaning

  $length[x, y, z]$

Derivation Systems Everywhere

- Defining meaning

  $$\begin{aligned}&\mathit{length}[x, y, z]\\ &\qquad \Rightarrow 1 + \mathit{length}\,[y, z]\end{aligned}$$

Derivation Systems Everywhere

- Defining meaning

$$
\begin{aligned}
&\mathit{length}[x, y, z] \\
&\qquad \Rightarrow 1 + \mathit{length}\,[y, z] \\
&\qquad \Rightarrow 1 + 1 + \mathit{length}\,[z]
\end{aligned}
$$

Derivation Systems Everywhere

- Defining meaning

$$
\begin{aligned}
&\mathit{length}[x, y, z] \\
&\qquad \Rightarrow 1 + \mathit{length}\,[y, z] \\
&\qquad \Rightarrow 1 + 1 + \mathit{length}\,[z] \\
&\qquad \Rightarrow 1 + 1 + 1 + \mathit{length}\,[]
\end{aligned}
$$

Derivation Systems Everywhere

- Defining meaning

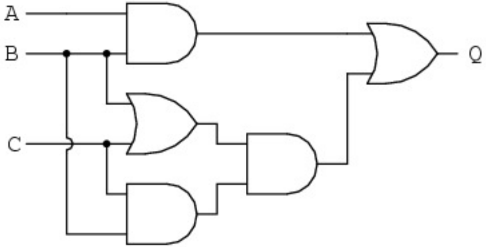
$$
\begin{aligned}
&\mathit{length}[x, y, z] \\
&\qquad \Rightarrow 1 \,+\, \mathit{length}\,[y, z] \\
&\qquad \Rightarrow 1 \,+\, 1 \,+\, \mathit{length}\,[z] \\
&\qquad \Rightarrow 1 \,+\, 1 \,+\, 1 \,+\, \mathit{length}\,[] \\
&\qquad \Rightarrow 1 \,+\, 1 \,+\, 1 \,+\, 0
\end{aligned}
$$

Derivation Systems Everywhere

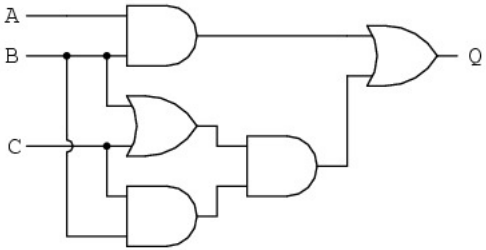
- Defining meaning

$$
\begin{aligned}
&\mathit{length}[x, y, z] \\
&\quad \Rightarrow 1 + \mathit{length}\,[y, z] \\
&\quad \Rightarrow 1 + 1 + \mathit{length}\,[z] \\
&\quad \Rightarrow 1 + 1 + 1 + \mathit{length}\,[] \\
&\quad \Rightarrow 1 + 1 + 1 + 0 \\
&\quad \Rightarrow 3
\end{aligned}
$$

Digital Logic



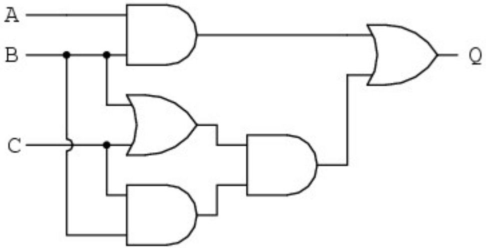
Digital Logic



An Equivalent Boolean Expression

```
(A and B) or ((B or C) and (C and B))
```

Truth Table



A	B	C	(A and B) or ((B or C) and (C and B))
T	T	T	?
T	T	F	?

Last Time/Today

- Last time:
  - Derivation Systems for Programming Languages: context-free grammars, type systems, small-step semantics,...
  - Introduced "pencil and paper" language design: Propositional Logic
  - Why? To illustrate the distinction between syntax and semantics.
- Today: representing the design in Haskell
- HW1 out Tuesday, 9/4

Propositional Logic

- **Proposition**: a statement that is either true or false
  E.g., "*It is raining*", "*Socrates was Greek*", etc.

Propositional Logic

- **Proposition**: a statement that is either true or false
  E.g., "*It is raining*", "*Socrates was Greek*", etc.
- **Propositional Sentences**
  E.g., Let p and q stand for "it is raining" and "the street is wet", respectively, then $\mathrm{p} \supset \mathrm{q}$ is a propositional sentence. Connective $\supset$ stands for "implies".

Propositional Logic

- **Proposition**: a statement that is either true or false
  E.g., "*It is raining*", "*Socrates was Greek*", etc.
- **Propositional Sentences**
  E.g., Let p and q stand for "it is raining" and "the street is wet", respectively, then $\mathrm{p} \supset \mathrm{q}$ is a propositional sentence.
  Connective $\supset$ stands for "implies".
- **Propositional Logic**:
  A derivation system for logical consequence in Prop. Logic
  I.e., assuming $P_1, \ldots, P_n$, must Q hold?

Propositional Logic

- **Proposition**: a statement that is either true or false
  E.g., "*It is raining*", "*Socrates was Greek*", etc.
- **Propositional Sentences**
  E.g., Let p and q stand for "it is raining" and "the street is wet", respectively, then $\mathrm{p} \supset \mathrm{q}$ is a propositional sentence.
  Connective $\supset$ stands for "implies".
- **Propositional Logic**:
  A derivation system for logical consequence in Prop. Logic
  I.e., assuming $P_1, \ldots, P_n$, must Q hold?
- **Propositional Logic Semantics**: namely, truth tables.

Propositional Logic

- **Proposition**: a statement that is either true or false
  E.g., "*It is raining*", "*Socrates was Greek*", etc.
- **Propositional Sentences**
  E.g., Let p and q stand for "it is raining" and "the street is wet", respectively, then $\mathrm{p} \supset \mathrm{q}$ is a propositional sentence.
  Connective $\supset$ stands for "implies".
- **Propositional Logic**:
  A derivation system for logical consequence in Prop. Logic
  I.e., assuming $P_1, \ldots, P_n$, must Q hold?
- **Propositional Logic Semantics**: namely, truth tables.

Derivation systems will play a role in all of these.

The Language Syntax

The propositional calculus is the simplest form of mathematical logic.

Definition (Propositional Calculus)

A *propositional formula* has one of the following forms:

- a *propositional variable*; usually denoted by a roman letter, p, q, r, etc.
- a *negation*; e.g., $\neg\varphi$ where φ is a propositional formula.
- an *implication*; e.g., $(\varphi \supset \gamma)$ where φ and γ are propositional formulae.

The Language Syntax as Context Free Grammar

Before giving a precise definition, let's consider an example. Let *Var* be an infinite set of symbols. We will refer to typical elements of *Var* with lower case roman letters (e.g., *p*, *q*, *r*, etc.). Assume $\{(,),\neg,\wedge\} \cap \mathit{Var} = \emptyset$, then let alphabet *A* be the set $\{(,),\neg,\wedge\} \cup \mathit{Var}$. Here is a CFG:

$$
\begin{aligned}
\mathit{Prop} &\rightarrow p \qquad \text{for any } p \in \mathit{Var} && (1)\\
\mathit{Prop} &\rightarrow (\neg \mathit{Prop}) && (2)\\
\mathit{Prop} &\rightarrow (\mathit{Prop} \supset \mathit{Prop}) && (3)
\end{aligned}
$$

This CFG defines a language, denoted $\mathcal{L}(\mathit{Prop})$.

Deriving members of $\mathcal{L}(Prop)$

- How do we determine if a particular sequence[1] of symbols from A is in $\mathcal{L}(Prop)$?

[1] I use "string" and "sequence of symbols" interchangably.

Deriving members of $\mathcal{L}(Prop)$

- How do we determine if a particular sequence[1] of symbols from A is in $\mathcal{L}(Prop)$?
- We perform a *derivation* of the string.

[1] I use "string" and "sequence of symbols" interchangably.

Deriving members of $\mathcal{L}(Prop)$

- How do we determine if a particular sequence[1] of symbols from A is in $\mathcal{L}(Prop)$?
- We perform a *derivation* of the string.
- For instance, is the string $(\neg p) \in \mathcal{L}(Prop)$? Yes, and here's the derivation:

$$\begin{aligned} Prop \rightarrow & (\neg Prop) \\ \rightarrow & (\neg p) \end{aligned}$$

[1] I use "string" and "sequence of symbols" interchangably.

"Woofs"

Definition (Well-Formed Formulae of Propositional Logic)

The primitive symbols of L are:

$$\neg \quad \supset \quad (\;)$$

The propositional symbols of L are of the form A_i for any positive integer i. The symbols, $\neg$ and $\supset$, are called *connectives*. Any propositional symbol is a *well-formed formula* (wff) of L. Furthermore, if φ and γ are wffs, the so are:

$$(\neg\varphi)$$

and

$$(\varphi \supset \gamma)$$

Definitional Extensions

Definition (Disjunction, Conjunction and Equivalence)

Familiar connectives are defined by:

$$
\begin{aligned}
(\varphi \vee \gamma) &\text{ is } \neg\varphi \supset \gamma && \text{(disjunction)} \\
(\varphi \wedge \gamma) &\text{ is } \neg(\neg\varphi \vee \neg\gamma) && \text{(conjunction)} \\
(\varphi \leftrightarrow \gamma) &\text{ is } (\varphi \supset \gamma) \wedge (\gamma \supset \varphi) && \text{(equivalence)}
\end{aligned}
$$

Definitional Extensions

Definition (Disjunction, Conjunction and Equivalence)

Familiar connectives are defined by:

$$
\begin{aligned}
(\varphi \vee \gamma) &\text{ is } \neg\varphi \supset \gamma && \text{(disjunction)}\\
(\varphi \wedge \gamma) &\text{ is } \neg(\neg\varphi \vee \neg\gamma) && \text{(conjunction)}\\
(\varphi \leftrightarrow \gamma) &\text{ is } (\varphi \supset \gamma) \wedge (\gamma \supset \varphi) && \text{(equivalence)}
\end{aligned}
$$

I will typically drop the parentheses when possible.

Language Syntax as a Haskell `data` declaration

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop
```

Language Syntax as a Haskell data declaration

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop
```

Compare with CFG version

$$\begin{array}{lcll} \mathit{Prop} & \rightarrow & p & \text{for any } p \in \mathit{Var} \\ \mathit{Prop} & \rightarrow & (\neg \mathit{Prop}) & \\ \mathit{Prop} & \rightarrow & (\mathit{Prop} \supset \mathit{Prop}) & \end{array}$$

Example

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop

negp = Not (Atom "p")
```

Example

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop

negp = Not (Atom "p")
```

Compare with:

$$\begin{aligned} Prop &\rightarrow (\ \neg\ Prop\ ) \\ &\rightarrow (\ \neg\ p\ ) \end{aligned}$$

Testing It Out, Part I

```
*PropLogic> negp
<interactive>:1:1:
    No instance for (Show Prop)
      arising from a use of ‘print’
    Possible fix: add an instance declaration for (Show Prop)
    In a stmt of an interactive GHCi command: print it
*PropLogic>
```

“No instance for (Show Prop)”

- This means that we have to write a function, $\mathit{show} :: \mathit{Prop} \rightarrow \mathit{String}$.

"No instance for (Show Prop)"

- This means that we have to write a function, $show :: Prop \to String$.
- To write a function of a particular type, we **always** start off from its **type template**.
  - It is almost always the right way to go.

"No instance for (Show Prop)"

- This means that we have to write a function, $show\ ::\ Prop \rightarrow String$.
- To write a function of a particular type, we **always** start off from its **type template**.
  - It is almost always the right way to go.
- The type template for show is determined by the definition of Prop data type:

```
show (Atom p)              = undefined
show (Not prop)            = undefined
show (Imply prop1 prop2) = undefined
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =  p
show (Not prop)            =
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =  p
show (Not prop)            =  "(-" ++ show prop ++ ")"
show (Imply prop1 prop2) =
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =  p
show (Not prop)            =  "(-" ++ show prop ++ ")"
show (Imply prop1 prop2) =
     "(" ++ show prop1 ++ " => " ++ show prop2 ++ ")"
```

Show Prop

Make this into an instance declaration:

```
instance Show Prop where
  show (Atom p)              = p
  show (Not prop)            = "(-" ++ show prop ++ ")"
  show (Imply prop1 prop2) =
       "(" ++ show prop1 ++ " => " ++ show prop2 ++ ")"
```

Show Prop

Make this into an instance declaration:

```
instance Show Prop where
  show (Atom p)            = p
  show (Not prop)          = "(-" ++ show prop ++ ")"
  show (Imply prop1 prop2) =
       "(" ++ show prop1 ++ " => " ++ show prop2 ++ ")"
```

```
*PropLogic> negp
(-p)
```

Another Instance Example

```
instance Eq Prop where
  (Atom p) == (Atom q)        = p == q
  (Not x) == (Not y)          = x == y
  (Imply x y) == (Imply u v) = (x == u) && (y == v)
  _ == _                      = False
```

Recall: Definitional Extensions

Definition (Disjunction, Conjunction and Equivalence)

Familiar connectives are defined by:

$$(\varphi \vee \gamma) \text{ is } \neg\varphi \supset \gamma \qquad \text{(disjunction)}$$
$$(\varphi \wedge \gamma) \text{ is } \neg(\neg\varphi \vee \neg\gamma) \qquad \text{(conjunction)}$$
$$(\varphi \leftrightarrow \gamma) \text{ is } (\varphi \supset \gamma) \wedge (\gamma \supset \varphi) \qquad \text{(equivalence)}$$

How do we represent these definitional extensions?

Defined Connectives as Functions

```
orPL ::
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL ::
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL ::  Prop -> Prop -> Prop
andPL phi gamma =
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL ::  Prop -> Prop -> Prop
andPL phi gamma =  Not (orPL (Not phi) (Not gamma))

iffPL ::
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL ::  Prop -> Prop -> Prop
andPL phi gamma =  Not (orPL (Not phi) (Not gamma))

iffPL ::  Prop -> Prop -> Prop
iffPL phi gamma = andPL (Imply phi gamma) (Imply gamma phi)
```

Two Notions of Validity for Propositional Logic

- Syntactic:
  - Derivation system for Propositional Logic
  - Axioms, Inference Rules, and Proofs
- Semantic:
  - Truth tables

Semantics: What does it mean?

Semantics (a.k.a., model theory) is another way of establishing the validity of a wff. The semantics of propositional logic consists of the well-known "truth tables".

A	B	$\neg B$	(A $\supset$ B)	(A $\wedge$ B)	$\neg(A \supset \neg B)$
T	T	F	T	T	
T	F	T	F	F	
F	T	F	T	F	
F	F	T	T	F	

Another Truth Table

A	B	C	$(A \wedge B) \vee ((B \vee C) \wedge (C \wedge B))$
T	T	T	
T	T	F	
T	F	T	
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

Axiom System for Propositional Logic

$$\varphi \supset (\gamma \supset \varphi) \tag{Ax.1}$$

$$(\varphi \supset (\gamma \supset \psi)) \supset ((\varphi \supset \gamma) \supset (\varphi \supset \psi)) \tag{Ax.2}$$

$$((\neg\gamma \supset \neg\varphi) \supset ((\neg\gamma \supset \varphi) \supset \gamma)) \tag{Ax.3}$$

There is only one inference rule in propositional logic, namely *Modus Ponens*.

$$\frac{\varphi \quad \varphi \supset \gamma}{\gamma} \text{ (MP)}$$

Instances

An instance of an axiom is a substitution of a wff for φ, γ, ψ
Instances of Axiom 1 $(\varphi \supset (\gamma \supset \varphi))$ include

Instance	Substitution
$A \supset (B \supset A)$	$[\varphi \mapsto A, \gamma \mapsto B]$
$A \supset ((A \supset A) \supset A)$	$[\varphi \mapsto A, \gamma \mapsto (A \supset A)]$
$\vdots$	$\vdots$

Formal Proofs

Definition (Proof)

Let Φ be the sequence $\varphi_1, \ldots, \varphi_n$ of propositional wffs. Then, Φ is a *proof* of φ_n if, and only if, for each φ_i in Φ, φ_i is either:

- an instance of Ax.1, Ax.2, or Ax.3, or
- there are φ_j and φ_k such that $j < i$ and $k < i$ and φ_i follows from φ_j and φ_k by *MP*.

Example Proof

Say I want to prove that $A \supset A$.

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2
2. $A \supset ((A \supset A) \supset A)$ Ax.1

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2
2. $A \supset ((A \supset A) \supset A)$ Ax.1
3. $(A \supset (A \supset A)) \supset (A \supset A)$ MP2,1

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2
2. $A \supset ((A \supset A) \supset A)$ Ax.1
3. $(A \supset (A \supset A)) \supset (A \supset A)$ MP2,1
4. $A \supset (A \supset A)$ Ax.1

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2
2. $A \supset ((A \supset A) \supset A)$ Ax.1
3. $(A \supset (A \supset A)) \supset (A \supset A)$ MP2,1
4. $A \supset (A \supset A)$ Ax.1
5. $A \supset A$ MP3,4

Proof as Tree: $A \supset A$

$$\dfrac{\dfrac{}{A \supset (A \supset A)}\;(\text{Ax.1}) \qquad \dfrac{\dfrac{}{A \supset ((A \supset A) \supset A)}\;(\text{Ax.1}) \qquad \dfrac{}{\begin{array}{c}(A \supset ((A \supset A) \supset A)) \supset \\ ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\;(\text{Ax.2})}{(A \supset (A \supset A)) \supset (A \supset A)}\;(\text{MP})}{A \supset A}\;(\text{MP})$$

Review: Formal Proofs

Definition (Proof)

Let Φ be the sequence $\varphi_1, \ldots, \varphi_n$ of propositional wffs. Then, Φ is a *proof* of φ_n if, and only if, for each φ_i in Φ, φ_i is either:

- an instance of Ax.1, Ax.2, or Ax.3, or
- there are φ_j and φ_k such that $j < i$ and $k < i$ and φ_i follows from φ_j and φ_k by *MP*.

φ_n is said to be a **theorem.** Note that the definition of "theorem" is recursive.

Recall: Axiom System for Propositional Logic

$$\varphi \supset (\gamma \supset \varphi) \tag{Ax.1}$$

$$(\varphi \supset (\gamma \supset \psi)) \supset ((\varphi \supset \gamma) \supset (\varphi \supset \psi)) \tag{Ax.2}$$

$$((\neg\gamma \supset \neg\varphi) \supset ((\neg\gamma \supset \varphi) \supset \gamma)) \tag{Ax.3}$$

There is only one inference rule in propositional logic, namely *Modus Ponens*.

$$\frac{\varphi \quad \varphi \supset \gamma}{\gamma} \text{ (MP)}$$

We want to represent axioms somehow.

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::  Prop -> Prop -> Prop -> Prop
axiom2 phi gamma psi =
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::  Prop -> Prop -> Prop -> Prop
axiom2 phi gamma psi =  Imply pre post
   where pre  = Imply phi (Imply gamma psi)
         post = Imply
                    (Imply phi gamma)
                    (Imply phi psi)

axiom3 phi gamma =
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::  Prop -> Prop -> Prop -> Prop
axiom2 phi gamma psi =  Imply pre post
   where pre  = Imply phi (Imply gamma psi)
         post = Imply
                  (Imply phi gamma)
                  (Imply phi psi)

axiom3 phi gamma =  Imply pre post
   where pre  = Imply (Not gamma) (Not phi)
         post = Imply hyp gamma
            where hyp  = Imply (Not gamma) phi
```

Review: Axiom Instances

An instance of an axiom is a substitution of a wff for φ, γ, ψ
Instances of Axiom 1 $(\varphi \supset (\gamma \supset \varphi))$ include

Instance	Substitution
$A \supset (B \supset A)$	$[\varphi \mapsto A, \gamma \mapsto B]$
$A \supset ((A \supset A) \supset A)$	$[\varphi \mapsto A, \gamma \mapsto (A \supset A)]$
$\vdots$	$\vdots$

Axioms as a Data Type

```
data Axiom  = Ax1 Prop Prop
            | Ax2 Prop Prop Prop
            | Ax3 Prop Prop
              deriving Eq
```

Axioms as a Data Type, cont'd

```
instance Show Axioms where
  show (Ax1 phi gamma)     = show (axiom1 phi gamma)
  show (Ax2 phi gamma psi) = show (axiom2 phi gamma psi)
  show (Ax3 phi gamma)     = show (axiom3 phi gamma)
```

Axioms as a Data Type, cont'd

```
instance Show Axioms where
  show (Ax1 phi gamma)     = show (axiom1 phi gamma)
  show (Ax2 phi gamma psi) = show (axiom2 phi gamma psi)
  show (Ax3 phi gamma)     = show (axiom3 phi gamma)
```

```
*PropLogic> Ax1 negp (Atom "q")
((-p) => (q => (-p)))
*PropLogic>
```

Axioms as a Data Type, cont'd

```
instance Show Axioms where
  show (Ax1 phi gamma)     = show (axiom1 phi gamma)
  show (Ax2 phi gamma psi) = show (axiom2 phi gamma psi)
  show (Ax3 phi gamma)     = show (axiom3 phi gamma)
```

```
*PropLogic> Ax1 negp (Atom "q")
((-p) => (q => (-p)))
*PropLogic>
```

What about the inference rule in propositional logic, namely *Modus Ponens*.

$$\frac{\varphi \quad \varphi \supset \gamma}{\gamma}\ \text{(MP)}$$

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2
2. $A \supset ((A \supset A) \supset A)$ Ax.1
3. $(A \supset (A \supset A)) \supset (A \supset A)$ MP2,1
4. $A \supset (A \supset A)$ Ax.1
5. $A \supset A$ MP3,4

Review: Proof as Tree, $A \supset A$

$$
\dfrac{\dfrac{}{A \supset (A \supset A)}\ (\text{Ax.1}) \qquad \dfrac{\dfrac{}{A \supset ((A \supset A) \supset A)}\ (\text{Ax.1}) \qquad \dfrac{}{\begin{array}{l}(A \supset ((A \supset A) \supset A)) \supset \\ \qquad ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\ (\text{Ax.2})}{(A \supset (A \supset A)) \supset (A \supset A)}\ (\text{MP})}{A \supset A}\ (\text{MP})
$$

Theorems

Hint: Theorem will be recursive.

```
data Theorem =
```

Theorems

Hint: Theorem will be recursive.

```
data Theorem = AxiomInst Axiom
```

Theorems

Hint: Theorem will be recursive.

```
data Theorem = AxiomInst Axiom
             | ModusPonens Theorem Theorem Prop
```

Theorems

Hint: Theorem will be recursive.

```
data Theorem = AxiomInst Axiom
             | ModusPonens Theorem Theorem Prop

instance Show Theorem where
  show (AxiomInst ax)            = show ax
  show (ModusPonens x y z)
     =        show x ++ "   " ++ show y
       ++ "\n--------------------------------\n    " ++
                    show z
```

Example

$$
\dfrac{
\dfrac{}{A \supset (A \supset A)}\,(\text{Ax.1})
\qquad
\dfrac{
\dfrac{}{A \supset ((A \supset A) \supset A)}\,(\text{Ax.1})
\quad
\dfrac{}{\begin{array}{c}(A \supset ((A \supset A) \supset A)) \supset \\ ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\,(\text{Ax.2})
}{(A \supset (A \supset A)) \supset (A \supset A)}\,(\text{MP})
}{A \supset A}\,(\text{MP})
$$

Example

$$
\dfrac{
\dfrac{}{A \supset (A \supset A)}\,(\text{Ax.1})
\qquad
\dfrac{
\dfrac{}{A \supset ((A \supset A) \supset A)}\,(\text{Ax.1})
\quad
\dfrac{}{(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))}\,(\text{Ax.2})
}{(A \supset (A \supset A)) \supset (A \supset A)}\,(\text{MP})
}{A \supset A}\,(\text{MP})
$$

```
a = Atom "A"

subproof = ModusPonens hyp1 hyp2 conc
  where hyp1 = AxiomInst (Ax1 a (Imply a a))
        hyp2 = AxiomInst (Ax2 a (Imply a a) a)
        conc = Imply (Imply a (Imply a a))
                     (Imply a a)
```

Example

$$
\dfrac{
\dfrac{}{A \supset (A \supset A)}\,(\text{Ax.1})
\qquad
\dfrac{
\dfrac{}{A \supset ((A \supset A) \supset A)}\,(\text{Ax.1})
\quad
\dfrac{}{\begin{array}{c}(A \supset ((A \supset A) \supset A)) \supset \\ ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\,(\text{Ax.2})
}{(A \supset (A \supset A)) \supset (A \supset A)}\,(\text{MP})
}{A \supset A}\,(\text{MP})
$$

```
a = Atom "A"

subproof = ModusPonens hyp1 hyp2 conc
  where hyp1 = AxiomInst (Ax1 a (Imply a a))
        hyp2 = AxiomInst (Ax2 a (Imply a a) a)
        conc = Imply (Imply a (Imply a a))
                     (Imply a a)
```

```
*PropLogic> subproof
(A => ((A => A) => A))   ((A => ((A => A) => A)) => ((A => (A => A)) => (A => A)))
---------------------------------------------------------------------------------
                      ((A => (A => A)) => (A => A))
```

Food for Thought

Is it possible to define a `Theorem` value in Haskell that is not a theorem?

Next Time

Finish up the specification of the Propositional Logic in Haskell