Propositional logic and recursive datatypes

Patrick D. Elliott

May 2, 2023

- Haskell's type system can be used to represent grammars as datatypes.
- Grammars of non-trivial formal languages/natural languages typically involve *recursive* rules for well-formed formulas/sentences
- An example from a CFG for natural language:

```
S -> NP VP
VP -> V S
```

- "She is upset."
- "Zach said she is upset."
- "Jenna believes Zach said she is upset."

Formal grammars in Haskell cont.

- In order to implement recursive grammars in Haskell, we use *recursive datatypes*.
    - Operations that apply to inhabitants of recursive datatypes are (typically) recursive functions.
    - Today we'll learn about recursive datatypes/functions in Haskell, using the simple example of propositional logic.

- Next week we'll talk about semantics, again using propositional logic as an example.
- For homework, read chapter 5 of (van Eijck, Jan and Unger, Christina, 2010) "Formal semantics for fragments".

Propositional logic

Sentences of propositional logic:

- $p \wedge q$
- $p \wedge (q \rightarrow r)$
- $p \vee \neg(q \wedge p)$
- $\bot \rightarrow (\top \vee (q \wedge r))$

To state a grammar for propositional logic, we first need a set of variables $\mathbf{Var} := \{ p, q, r, \ldots \}$, and a set of constants $\{ \top, \bot \}$.

- If $p \in \mathbf{Var} \cup \{ \top, \bot \}$, then p is a sentence of propositional logic.
- If ϕ, ψ are sentences of propositional logic, then $(\phi \wedge \psi)$ is a sentence of propositional logic.
- If ϕ, ψ are sentences of propositional logic, then $(\phi \vee \psi)$ is a sentence of propositional logic.
- If ϕ, ψ are sentences of propositional logic, then $(\phi \rightarrow \psi)$ is a sentence of propositional logic.
- If ϕ is a sentence of propositional logic, then so is $\neg\phi$

We'll use a *sum type* to implement the grammar of propositional logic. We can start with just the following:

```haskell
data PropL = PVar String deriving (Eq,Show)
```

```
ghci> :t (PVar "p1")
PropL
ghci> :t (PVar "p3")
PropL
```

Adding negation: recursive datatypes!

```
data PropL = PVar String | PNot PropL deriving (Eq,Show)
```

Note that we *reuse* `PropL` in the constructor for negative sentences; this means that if an expression `p` is of type `PropL`, then `PNot p` is of type `PropL`.

```
ghci> :t (PNot (PVar "p1"))
PropL
ghci> :t (PNot (PNot (PNot (PVar "p1")))
PropL
```

Note that this already gives us infinite inhabitants of type `PropL`.

Adding connectives

```haskell
data PropL = PVar String | PNot PropL | PAnd PropL PropL | POr
↪  PropL PropL deriving (Eq,Show)
```

```
ghci> :t (PAnd (PVar "p1") (POr (PVar "p2") (PVar "p3")))
PropL
ghci> :t (PAnd (PVar "p1"))
error
```

Bonus: infix constructors!

Functions that take two arguments can be used as *infix operators* by enclosing them in backticks. This also goes for constructors:

```
ghci> :t PAnd
  PropL -> PropL -> PropL
ghci> :t ((PVar "p1") `PAnd` (PVar "p2"))
PropL
```

You can even use infix constructors in the data declaration:

```
data PropL = PVar String | PNot PropL | PropL `PAnd` PropL | PropL
 ↪  `POr` PropL deriving (Eq,Show)
```

Implementing a custom **Show** instance for **PropL** simply amounts to defining a function **show** of type **PropL -> String**.

```
data PropL = PVar String | PNot PropL | PropL `PAnd` PropL | PropL
↪  `POr` PropL deriving Eq

instance Show PropL where
  show (PVar s) = s
  show (PNot p) = "~" ++ show p
  show (p `PAnd` q) = "(" ++ show p ++ " & " ++ show q ++ ")"
  show (p `POr` q) = "(" ++ show p ++ " | " ++ show q ++ ")"
```

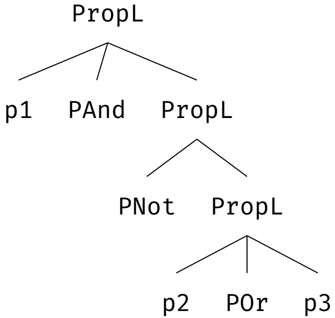
The **Show** instance we just declared will automatically be used by ghci.

```
ghci> ((PVar "p1") `PAnd` (PNot ((PVar "p1") `POr` (PVar "p3")))
(p1 & ~(p2 | p3))
```

We can also use it explicitly by calling `show` on something of type `PropL`.

ASTs

Recursive datatypes are used to create an Abstract Syntax Tree (AST) for sentences of propositional logic.



Let's say that we want to compute the number of operators in a formula.

In order to do so we'll need a recursive function `opsNr`.

First, we define the base of the recursion (where the recursion halts):

```
opsNr :: PropL -> Int
opsNr (PVar _) = 0
```

For all other cases we need recursion:

```
opsNr :: PropL -> Int
opsNr (PVar _) = 0
opsNr (PNot p) = 1 + opsNr p
opsNr (PAnd p q) = 1 + opsNr p + opsNr q
opsNr (POr p q) = 1 + opsNr p + opsNr q
```

Another recursive function: depth

```haskell
depth :: PropL -> Int
depth (PVar _) = 0
depth (PNot p) = 1 + depth p
opsNr (PAnd p q) = undefined
opsNr (POr p q) = undefined
```

Depth cont.

```haskell
depth :: PropL -> Int
depth (PVar _) = 0
depth (PNot p) = 1 + depth p
depth (PAnd p q) = 1 + max (depth p) (depth q)
depth (POr p q) = 1 + max (depth p) (depth q)
```

Exercise: gather names

- Exercise: write a recursive function that returns a list of all of the variables that occur in a formula.
- As a bonus, remove duplicates and sort the output alphabetically.

$\mathcal{F}in$

van Eijck, Jan and Unger, Christina (2010). *Computational Semantics with Functional Programming*, Cambridge University Press.