Program Verification (CS1.303)

Monsoon 2021, IIIT Hyderabad
04 Jan, Tuesday (Lecture 1)

Taught by Prof. Venkatesh Choppella

Propositional Logic

Propositional logic is about how to combine propositions, or assertions. We have certain *atomic propositions* (for example, $2 + 2 = 5$), which can be combined using operators (as in $2 + 2 = 5$ or $1 + 1 = 2$).

There are two aspects of logic: its *syntax* (how to write expressions, a kind of grammar) and its *semantics* (what the expressions mean).

Syntax of Propositional Logic

In propositional logic, we have a denumerable set of *propositional variables*. Elements of this set are typically denoted by p, q, p_1, q_1, etc.:

$$\mathrm{Var} = \{p, q, p_1, q_1, \dots\}$$

Using this, we define the set of propositional formulas or expressions:

$$\mathrm{Exp} ::= \mathrm{Var} \mid \mathrm{Exp} \ \land \ \mathrm{Exp} \mid \mathrm{Exp} \ \lor \ \mathrm{Exp} \mid \mathrm{Exp} \ \rightarrow \ \mathrm{Exp} \mid \neg\mathrm{Exp}$$

This is a definition in BNF, or Backus-Naur form. This can be written alternatively in the inference form:

$$\textsc{Var}\quad \frac{p \in \mathrm{Var}}{p \in \mathrm{Exp}}$$

$$\textsc{And}\quad \frac{e_1 \in \mathrm{Exp} \qquad e_2 \in \mathrm{Exp}}{e_1 \land e_2 \in \mathrm{Exp}}$$

$$\textsc{Or}\quad \frac{e_1 \in \mathrm{Exp} \qquad e_2 \in \mathrm{Exp}}{e_1 \lor e_2 \in \mathrm{Exp}}$$

$$\textsc{Imp}\;\frac{e_1 \in \text{Exp} \qquad e_2 \in \text{Exp}}{e_1 \rightarrow e_2 \in \text{Exp}}$$

$$\textsc{Not}\;\frac{e \in \text{Exp}}{\neg e \in \text{Exp}}$$

The set of expressions is the smallest set satisfying these rules.
These rules give us a way to write the structure of propositional logic expressions as trees (abstract syntax trees).

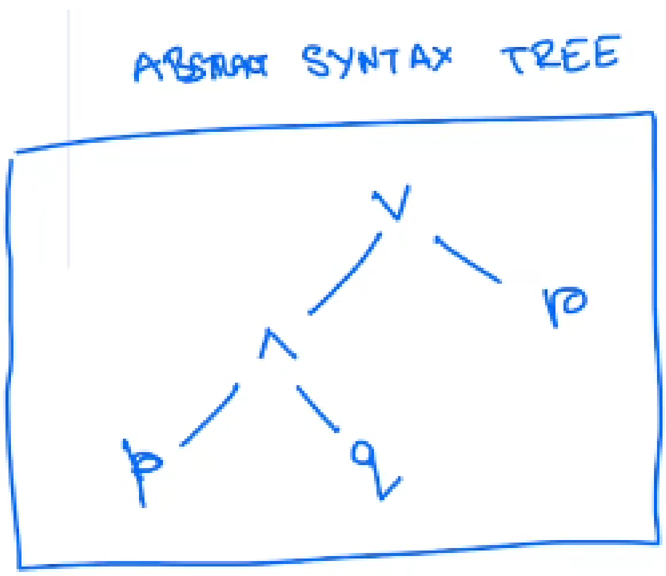

Figure 1: Abstract Syntax Tree of $(p \wedge q) \vee r$

This can be implemented in Haskell (mirroring the inference rules and the BNF grammar) in this way:

```haskell
data Exp = Var Char
         | And Exp Exp
         | Or  Exp Exp
         | Imp Exp Exp
         | Not Exp
```

assuming that the set of propositional variables can be mapped to the set of characters.

These definitions are *inductive*, in that they define the set of expressions by building up.

Functions on Propositional Logic Expressions We can define a number of functions on propositional logic expressions.

For example, the function vars : Exp → Var, which gives the set of all variables contained in an expression, can be defined recursively:

$$\begin{aligned}
\text{vars}(p) &= \{p\} \\
\text{vars}(e_1 \wedge e_2) &= \text{vars}(e_1) \cup \text{vars}(e_2) \\
\text{vars}(e_1 \vee e_2) &= \text{vars}(e_1) \cup \text{vars}(e_2) \\
\text{vars}(e_1 \to e_2) &= \text{vars}(e_1) \cup \text{vars}(e_2) \\
\text{vars}(\neg e) &= \text{vars}(e)
\end{aligned}$$

Similarly, we can define the functions size : Exp → $\mathbb{N}$ and height : Exp → $\mathbb{N}$.

Substitutions In the AST of an expression, each node has an "address", given by how to reach it from the root.

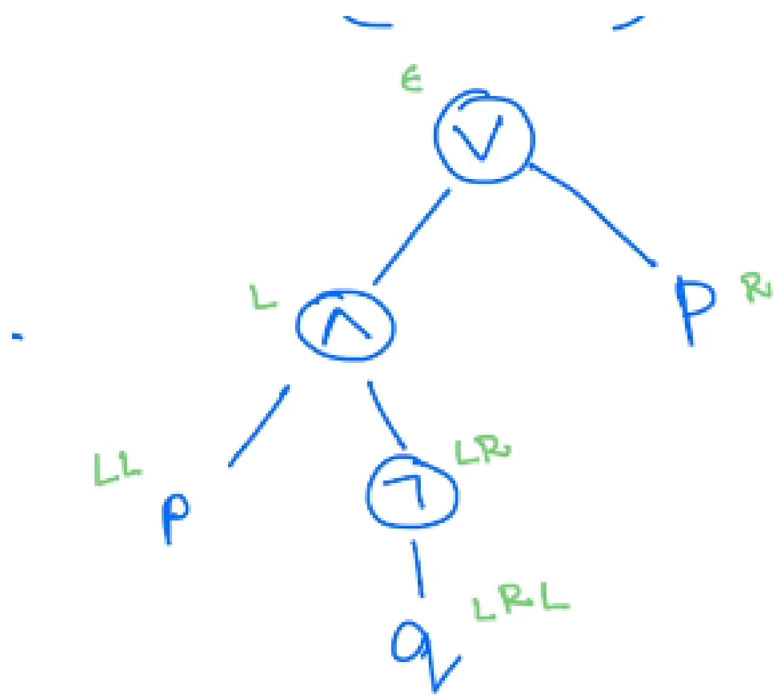

Figure 2: The AST with Addresses for $(p \wedge \neg q) \vee r$

Thus, given an expression, there is a set of addresses associated with it, for which we can define a function addr : Exp → $\mathcal{P}$(Addr). The set Addr of all possible addresses is clearly $(L + R)^*$.

If e is an expression and $p \in \mathrm{addr}(e)$, then there is a unique expression associated with e and p called the subexpression of e at address p. Thus we can define a function $\mathrm{subexp} : \mathrm{Exp} \times \mathrm{Addr} \to \mathrm{Exp}$.
We can also define $\mathrm{graft} : \mathrm{Exp} \times \mathrm{Addr} \times \mathrm{Exp} \to \mathrm{Exp}$, which removes a subexpression at a particular address in an expression, and replaces it with a different expression.
A related function $\mathrm{occurrences} : \mathrm{Exp} \times \mathrm{Exp} \to \mathcal{P}(\mathrm{Addr})$ gives the set of addresses in an expression at which a given subexpression can be found.

Finally, we can define $\mathrm{substitute} : \mathrm{Exp} \times \mathrm{Exp} \times \mathrm{Exp} \to \mathrm{Exp}$, which replaces all occurrences of a subexpression in an expression with another expression.