Mathematical and Logical Foundations of Computer Science

Lecture 2 - Symbolic Logic

Vincent Rahli

(some slides were adapted from Rajesh Chitnis' slides)

University of Birmingham

Where are we?

- **Symbolic logic**
- Propositional logic
- Predicate logic

Today

We will introduce some useful concepts to deal with logical systems. Some of them will make more sense as we experience them during the course of this module.

- Symbolic logic
- Grammars
- (Meta)variables
- Axiom schemata
- Substitution

Symbolic Logics

Symbolic logics are **formal languages** that allow conducting logical reasoning through the **manipulation of symbols**.

> *"Symbolic logic is the development of the most general principles of rational procedure, in ideographic symbols, and in a form which exhibits the connection of these principles one with another." (Irving Lewis in A Survey of Symbolic Logic)*

Pioneered for example by Leibniz, Boole, Frege, etc.

For example:

- **Propositional logic**
- **Predicate logic**
- Higher-order logic

Grammars - BNFs

Two important aspects of a language are:

- its **syntax** describing the well-formed sequences of symbols denoting objects of the language;
- and its **semantics** assigning meaning to those symbols.

This lecture focuses on syntax.

The syntax of a language is defined through a **grammar**.

In particular, the language of a symbolic logic is defined by a grammar that allows deriving formulas from collections of symbols (we will see an example in a few slides).

Grammars - BNFs

The grammar of such a language is often defined using a **Backus Naur Form** (BNF). BNFs allow defining **context-free grammars** (i.e., where production rules are context independent). They are collections of **rules** of the form:

$$lhs ::= rhs_1 \mid \cdots \mid rhs_n$$

Meaning: this rule means that the left-hand-side lhs (a non-terminal symbol) can expand to any of the forms rhs_1 to rhs_n on the right-hand-side.

Each rhs_i is a sequence of non-terminal and terminal symbols.

The **arity** of a terminal symbol is the number of arguments it takes.

The **Fixity** of a terminal symbol is the place where it occurs w.r.t. its arguments: **infix** if it occurs in-between its arguments, **prefix** if it occurs before, and **postfix** if it occurs after.

Grammars - BNF example

Example of a BNF for (some) arithmetic expressions:

$$exp ::= num \mid exp + exp \mid exp \times exp$$

where a numeral num is a sequence of digits. Here exp is a non-terminal symbol and $+$, $\times$, $\mathit{0}$, 1, etc., are terminal symbols.

Arity & fixity:

- $\mathit{0}$, 1, etc. are nullary (arity 0) operators (they are called **constants**).
- $+$ and $\times$ are binary (arity 2) infix operators

Derivations:

$$exp \mapsto exp + exp \mapsto 1 + exp \mapsto 1 + 2$$
$$exp \mapsto exp \times exp \mapsto exp \times 0 \mapsto 2 \times 0$$

How to extend this language to allow for conditional expressions?

$$\begin{array}{lll} exp & ::= & num \mid exp + exp \mid exp \times exp \mid \texttt{if}\ b\ \texttt{then}\ exp\ \texttt{else}\ exp \\ b & ::= & \texttt{true} \mid \texttt{false} \mid b\ \&\ b \mid b \parallel b \end{array}$$

Fixity: all the above operators are infix.

Grammars - BNF example

Example of a BNF for propositional logic formulas:

$$P ::= a \mid P \to P \mid P \vee P \mid P \wedge P \mid \neg P$$

where a ranges over a set of atomic propositions (e.g., *“it is raining”*, or *“it is sunny”*). Here P is a non-terminal symbol and $\wedge$, $\vee$, $\to$, and $\neg$, as well as the atomic propositions, are terminal symbols.

Arity & Fixity: $\wedge$, $\vee$, $\to$ are binary infix operators, $\neg$ is a unary (arity 1) prefix operator

Example: let s stand for “it is sunny”, and r for “it is rainy”

Derivation:

$$P \mapsto P \vee P \mapsto r \vee P \mapsto r \vee \neg P \mapsto r \vee \neg s$$

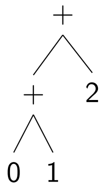
Grammars - abstract syntax trees

An expression derived from a BNF grammar can then be seen as a tree, called an **abstract syntax tree**.

For example, given the grammar:

$$exp ::= num \mid exp + exp \mid exp \times exp$$

an abstract syntax tree corresponding to $0 + 1 + 2$ is:



Grammars - associativity

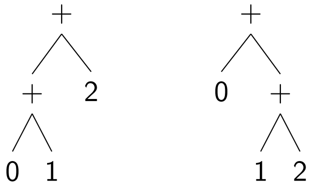
Note the **ambiguity** in our example: $0 + 1 + 2$.
Does it stand for $(0 + 1) + 2$ or $0 + (1 + 2)$?

We need to define the **associativity** of the terminal symbols to avoid ambiguities.

- left associativity: $(0 + 1) + 2$
- right associativity: $0 + (1 + 2)$

We will consider the first but we will sometimes use parentheses to avoid ambiguities.

Those have different abstract syntax trees:



Grammars - precedence

What about: $0 + 1 \times 2$? This is again ambiguous.

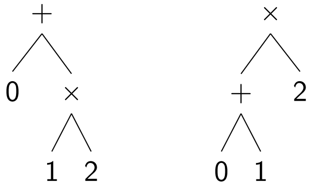
Does it stand for $(0 + 1) \times 2$ or $0 + (1 \times 2)$?

We need to define the **precedence** of the terminal symbols to avoid ambiguities.

- $\times$ has higher precedence: $0 + (1 \times 2)$
- $+$ has higher precedence: $(0 + 1) \times 2$

We will consider the first.

Those have different abstract syntax trees:



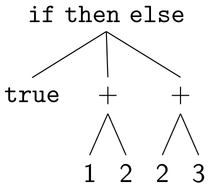
Grammars - example

What is the abstract syntax tree for?

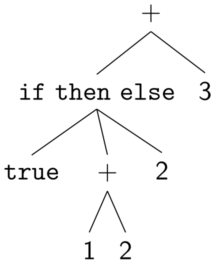
$$\texttt{if true then } 1+2 \texttt{ else } 2+3$$

Again this is ambiguous. Without knowing which operator has precedence over the other, it could be either of the two:

$\texttt{if true then } (1+2) \texttt{ else } (2+3)$



$(\texttt{if true then } 1+2 \texttt{ else } 2)+3$



Grammars - associativity, precedence, parentheses

To avoid ambiguities:

- define the associativity of symbols
- define the precedence between symbols
- use parentheses to avoid ambiguities or for clarity

Parentheses are sometimes necessary:

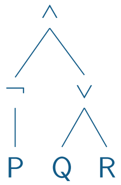
- using left associativity $0 + 1 + 2$ stands for $(0 + 1) + 2$
- we need parentheses to express $0 + (1 + 2)$

Grammars - example

Given the grammar:

$$P ::= a \mid P \to P \mid P \vee P \mid P \wedge P \mid \neg P$$

what is the abstract syntax tree for $(\neg P) \wedge (Q \vee R)$?



(Meta)variables

Some of these concepts will start making more sense when we come to experience them during the course of this module

We sometimes want to write down expressions/formulas such as $exp + exp$ or $P \rightarrow P$, where exp and P are non-terminals.

In that case exp and P act as **variables** that can range over **all possible** expressions/formulas.

Such variables are typically called, **metavariables** or **schematic variables**, and act as placeholders for any element derivable from a given grammar rule.

For example, we might write $P \rightarrow P$ to mean that P implies P whatever the proposition P is: "it is rainy" → "it is rainy" is true; "it is sunny" → "it is sunny" is true; etc.

(Meta)variables

Notation. Given the grammar:

$$exp ::= num \mid exp + exp \mid exp \times exp$$

one typically allows exp, exp_0, exp_1, . . . , exp', exp'', . . . , as variables ranging over all possible arithmetic expressions derivable using the above rule.

Technical details:

- The expressions of the language captured by the above grammar, has all the ones that cannot be derived further, i.e., that do not contain **non-terminal** symbols.
- $exp + exp$ is not part of this language but is useful to capture a **collection** of expressions.
- Why is it called a “metavariable”? A metavariable is a variable within the language, called the **metatheory**, used to describe and study a theory at hand.

(Meta)variables

For example, let us consider the following grammar:

$$\begin{array}{lcl} exp & ::= & num \mid exp + exp \mid exp \times exp \\ eq & ::= & exp = exp \end{array}$$

where equalities are used to state that two expressions are equal.

This defines the syntax of a simple symbolic logic to reason about arithmetic expressions.

We use this language to state laws of arithmetic by describing what equalities hold using variables that act as placeholders for any possible expressions.

Some equalities are assumed to hold in our simple logic through **axioms**, such as $0 + 0 = 0$, $1 + 0 = 1$, $2 + 0 = 2$, etc.

Axiom schemata

For example, as part of a "number theory" one may want to assume that the following equality holds:

$$exp + 0 = exp$$

A standard law of arithmetic that states: 0 is an additive identity.

It stands for an infinite number of axioms, which can be obtained by **instantiating** the variable exp with any arithmetic expression. This is called an **axiom schemata**.

For example, the following equality is such an instance:

$$1 + 0 = 1$$

Other examples of instances?

- $2 + 0 = 2$
- $(1 + 2) + 0 = 1 + 2$
- etc.

Axiom schemata

As another example, take again propositional logic, whose syntax is:

$$P ::= a \mid P \to P \mid P \vee P \mid P \wedge P \mid \neg P$$

Variables are useful to state axioms of the logic.

For example, we can state:

$$(P \wedge Q) \to P$$

using the variables P and Q.

By replacing P by "2 is prime" and Q by "2 is even", we can obtain the following instance of this formula:

$$(2 \text{ is prime} \wedge 2 \text{ is even}) \to 2 \text{ is prime}$$

Substitution

How do we obtain the equality:

$$1 + 0 = 1$$

from the axiom schema:

$$exp + 0 = exp$$

This is done by instantiating the schema, i.e., by substituting the variable exp with an arithmetic expression. For example here, we substituted exp with 1.

A **substitution** is a mapping (e.g., a key/value map), that maps metavariables to arithmetic expressions.
The **substitution operation** is the operation that replaces all occurrences of the keys by the corresponding values (the 1st key/value pair is considered if a key occurs more than once).

Substitution

We write $k_0\backslash v_0, \ldots, k_n\backslash v_n$ for the substitution that maps k_i to v_i for $i \in \{0, \ldots, n\}$.

For example:

- The substitution $exp\backslash 1$ maps exp to 1.
- $exp_1\backslash 0, exp_2\backslash 1$ maps exp_1 to 0 and exp_2 to 1.
- $exp_1\backslash 0, exp_2\backslash 1, exp_1\backslash 1$ also maps exp_1 to 0 and exp_2 to 1.

The substitution operation, written $eq[s]$, takes an equality eq and a substitution s, and replaces all occurrences of the keys of s by the corresponding values in eq.

For example: $(exp + 0 = exp)[exp\backslash 1]$ returns $1 + 0 = 1$.

Substitution - formally

Formally, the substitution operation is defined recursively on the syntactic forms they are applied to.

For example, the substitution operation computes as follows on arithmetic expressions:

$$
\begin{array}{lcl}
num[s] & = & num \\
(exp_1 + exp_2)[s] & = & exp_1[s] + exp_2[s] \\
(exp_1 \times exp_2)[s] & = & exp_1[s] \times exp_2[s] \\
(exp_1 = exp_2)[s] & = & exp_1[s] = exp_2[s]
\end{array}
$$

and as we allow variables in expressions:

$$
\begin{array}{lcl}
v[s] & = & v\text{, if } v \text{ is not a key of } s \\
v[s] & = & e\text{, if } s \text{ maps } v \text{ to } e
\end{array}
$$

Substitution - further examples

Consider the following commutativity schema:

$$exp_1 + exp_2 = exp_2 + exp_1$$

What does $(exp_1 + exp_2 = exp_2 + exp_1)[exp\backslash 1]$ return?
$exp_1 + exp_2 = exp_2 + exp_1$

What does $(exp_1 + exp_2 = exp_2 + exp_1)[exp_1\backslash 1]$ return?
$1 + exp_2 = exp_2 + 1$

What does $(exp_1 + exp_2 = exp_2 + exp_1)[exp_1\backslash 1, exp_2\backslash 2]$ return?
$1 + 2 = 2 + 1$

Conclusion

What did we cover today?

- A formal language such as a symbolic logic has a syntax captured by a grammar (e.g., a BNF).
- (Meta)variables are used to capture collections of axioms (as axiom schemata) of symbolic logics.
- Substitution is used to derive instances of axiom schemata.

Next time?

- Propositional logic - Syntax