Subtitle: An Introduction to Formal Methods.
Instructor: Dr. Lutz Hamel
Email: lutzhamel@uri.edu
Office: Tyler, Rm 251
There are no required books in this course; however, occasionally I will assign readings based on material available on the web.
The aim of this course is to

- Familiarize you with the basic techniques of applying formal methods to programming languages.
- This includes constructing models for programming languages and using these models to prove properties such as correctness and equivalence of programs.
- Look at all major programming language constructs including assignments, loops, type systems, and procedure calls together with their models.
- Introduce mechanical theorem provers so that we can test and prove properties of non-trivial programs.
**Definition:** In programming language semantics we are concerned with the **rigorous mathematical study** of the **meaning** of programming languages. The meaning of a language is given by a **formal system** that describes the possible computations expressible within that language.
Some Definitions

**Definition:** In computer science and software engineering, formal methods are techniques for the specification, development and verification of software and hardware systems based on formal systems.
Definition: A formal system consists of a formal language and a set of inference rules. The formal language is composed of primitive symbols that make up well-formed formulas and the inference rules are used to derive expressions from other expressions within the formal system. A formal system may be formulated and studied for its intrinsic properties, or it may be intended as a description (i.e., a model) of external phenomena.\(^1\)

In order to be truly useful in computer science, we require our formal systems to be machine executable.

\(^1\)Wikipedia
Uses of Formal Methods

**Implementation Issues**  Formally specified models can be considered machine-independent specifications of program behavior. They can act as “yard sticks” for the correctness of program implementations, transformations, and optimizations.

**Verification**  Basis of methods for reasoning about program properties (e.g. equivalence) and program specifications (program correctness).

**Language Design**  Can bring to light ambiguities and unforeseen subtleties in programming language constructs.
When programming we can observe two mental activities:

- We construct *correct looking* programs - *syntactically* correct programs.

- We construct *models* of the intended computation in our minds. Consider,

  ```
  x := 1
  while (x <= 10) do
    writeln(x)
    x := x + 1
  end while
  ```

Any person with some familiarity of programming immediately has a mental picture that this program will generate a list of integers from 1 through 10.
Mirroring our intuition, language definitions consist of two parts:

**Syntax** The formal description of the *structure* of well-formed expressions, phrases, programs, etc.

**Semantics** The formal description of the *meaning* of the syntactic features of a programming language usually understood in terms of the runtime *behavior* each syntactic construct evokes. The formal description of the behavior of all the syntactic features of a language is considered a *model* of the language.
Syntax and semantics of a programming language are usually related via an *evaluation relation* or *interpretation*, say $h$. Then we say that the interpretation $h$ takes each syntactic element and maps it into the appropriate semantic construct.

We often represent this with the diagram

```
Semantics

h

h

Syntax
```

**Note:** In order for the interpretation $h$ to make any sense we will have to define the syntax and semantics in terms of sets.
The formal systems we will be using in this course are:

- First-order logic extended with natural deduction – natural semantics.
- The *first order predicate calculus* (often also called first order logic) to construct semantics of programming languages.
Read Chapter 0 in "Denotational Semantics" by David Schmidt (available from the course website).
Read Sections 2.1 and 2.2 in "Denotational Semantics" by David Schmidt.