Objectives

This course studies principles underlying the design of programming languages. It has four main objectives:

1. Learning principles of programming languages: elements of programming languages; abstraction means in programming languages; formal definition of programming languages – concrete syntax, abstract syntax, operational semantics; program correctness – type checking and type inference systems.
2. Describing program execution by studying evaluators: interpreters, transformers and compilers.
3. Comparing programming paradigms: Functional programming, Logic programming and Imperative programming.
4. Learning principles of program design: Abstraction, contracts, modular architecture, testing.

The course is a mixture of theory and practice. Theoretical topics are supported by implemented software, and course assignments involve programming. The course interleaves two main threads: (1) learning new programming techniques (functional programming, logic programming) using practical examples in JavaScript, TypeScript and Prolog; (2) learning meta-programming by developing parsers, interpreters, and program transformers (illustrated by a type inference system for functional programming). The course uses the Scheme language (Racket) and TypeScript for teaching the general theory and practice of language design and implementation. Meta-programming tools are developed to demonstrate the advantage of a formal definition of programming languages syntax and semantics to "reason about code". These software tools are also used as practical examples of good software design, using functional programming.

Topics

1. Applied Functional Programming
   1. Benefits of Functional programming (FP): safety, concurrency, asynchronous code
   2. Characteristics of FP: pure functions, immutability, higher-order functions, composition
   3. FP in JavaScript: function, map, filter, chain, flatmap, reduce, zip.
   4. Types in TypeScript: incremental typing, basic types, interfaces, contracts
   5. Hierarchical data types: Arrays, objects, JSON
   6. Higher-order procedures
   7. Polymorphic procedures: Generic types
2. Syntax, Semantics, Types
   1. Concrete and abstract syntax: comparing JavaScript and Scheme
   2. Operational semantics: the substitution model
   3. Types: basic types, composite types, type constructors, algebraic data types
   4. Higher-Order Functions
      1. Functions and the processes they generate
      2. Anonymous functions, closures, lexical scope
      3. Partial evaluation, currying and function composition
      4. Primitive and user-defined procedures, derived expressions, special operators
3. Abstraction on Data and on Control
   1. Data abstraction: Abstract Data Types (ADTs), Interfaces, Immutable data structures
   2. ADTs and Algebraic Data Types to model abstract syntax
   3. From iterators to streams to observables - push and pull data flow
   4. Control abstraction: Promises and asynchronous programming - reactive programming model
   5. Coroutines: generators, yield, async and await in JavaScript
   6. Continuation passing style programming in Scheme
   7. Tail Recursion Optimization, stack execution model
4. Evaluators for Functional Programming
   1. Abstract syntax parser
   2. An evaluator for the substitution model
   3. The environment model evaluation algorithm
   4. Evaluator for the environment-based operational semantics
5. Type Correctness
   1. Type checking and inference
   2. An algorithm for type inference
   3. Implementing a program transformer: Scheme to Typed Scheme.
6. Logic Programming
   1. Relational logic programming
   2. Logic programming
   3. Meta-tools: Backtracking optimization (cuts); Unify;
   4. An evaluator for Prolog
7. Conclusion: Techniques, Principles and Languages Comparison