CS351
Fall 2025

Final Exam Study Guide

Structure

- Same as always
- Mostly multiple choice questions
- 1-2 Open Response (3pts each)
- Total ~32pts
- 120m (but designed for 80m; meaning ~1/2 done in about 40m)
- Default: closed book
- In-person, on Kodiak; proctor must be able to see your screen.

Topics

- Call Semantics
    - **Pass-by-value**: formal bound to a new reference containing the result of evaluating the passed expression in the calling environment.
    - **Pass-by-reference**: (only applies when passing a variable) formal bound to the reference of the passed variable, sharing the reference with that variable.
    - **Pass-by-name**: (only applies when passing a non-variable expression) formal bound to a thunk reference that captures the unevaluated, passed expression and the calling environment. Each time the thunk is dereferenced, its captured expression is evaluated in the captured calling environment, and the result is returned.
    - **Pass-by-need**: (only applies when passing a non-variable expression) formal bound to a thunk reference that captures the unevaluated, passed expression, and the calling environment. The first time the thunk is dereferenced, the expression is evaluated in the calling environment, and the resulting value is cached and returned. When the thunk is subsequently dereferenced, the cached value is returned.
- V6
    - All symbols are bound to values.
    - Procs are **closures** that capture formals, an expression (body), and the defining environment. When applied, extends the captured environment binding formals to passed arguments (values resulting from evaluating actual parameter expressions in the calling environment). **Closures** are also **higher-ordered functions** which can be passed into or out of functions.
    - Environments are extended with new bindings only by LetExp, LetrecExp, and AppExp (call to proc).
    - LetExp and AppExp evaluate RHS/passed expressions in the current environment, and then extend the current environment binding the LHS/formals with the results of those expressions, and then evaluate their body in this new environment.

    - LetrecExp extends the current environment with an empty Bindings, evaluates each RHS in order in the new environment adding a new binding to the Bindings before evaluating the next RHS, and then evaluates its body in the new environment.
- SET
    - Like V6 with the following changes:
    - All symbols are now bound to references. A reference represents a mutable space in memory. Each has two operations: deRef() and setRef(Val).
    - Side-effect are now possible through a SetExp ::= SET <VAR> EQ <Exp>
    - Call semantics:
        - Exp -> Pass-by-value
- REF
    - Like V6 with the following changes:
    - Call semantics:
        - VarExp -> Pass-by-reference
        - Other Exp -> Pass-by-value
- NAME
    - Like V6 with the following changes:
    - Call semantics:
        - VarExp -> Pass-by-reference
        - LitExp and ProcExp -> Pass-by-value
        - Other Exp -> Pass-by-name
    - Pass-by-name: a ThunkRef created that captures calling environment and the expression. Each deRef of ThunkRef evaluates its expression inside the calling environment. ThunkRefs are read-only, so setRef raises an exception.

- NEED
    - Call semantics
        - VarExp -> pass-by-reference
        - LitExp and ProcExp -> pass-by-value + READ-ONLY via ValRORef
        - Other -> pass-by-need (AKA lazy-evaluation)
    - pass-by-need: memoization
        - Unevaluated expression and calling environment captured into a ThunkRef which is bound to formal
        - On first dereference, expression evaluated and result is **memoized** (saved in the ThunkRef)
        - On subsequent dereferences, the **memoized** value is returned without re-evaluating the expression.
        - **memoization**: saving the result of an expression so that it can be used without having to re-evaluate the expression. It is an optimization technique that trades memory for time.
- TYPE

    - Be able to properly declare/annotate a proc's types.

    - Be able to determine a procs defined type.
    - Be able to identify type errors detected by TYPE.
    - Be able to determine the type of an expression.
    - Be able to properly use, and know when to use, declare to forward declare the type of a symbol.
- OBJ
    - Be able to evaluate and write code in OBJ
    - Be able to resolve symbols in the context of inheritance and these reserved words:
        - superclass
        - myclass
        - !@
        - super
        - this
        - self
    - Be able to identify what each of the above reserved words points to.
    - Be able to identify which reserved words are available in a static method vs an object's methods.
    - Be able to craft an expression to access a specific, possibly shadowed symbol.