OVERVIEW FOR A9

• Implement an interpreter for a specified language, JoCalf
  • Much of the interpreter is built: no work required on lexer, parser, REPL, Main
  • Your work:
    • Design some of the AST that represents JoCalf expressions
    • Implement short functions that aid in generating the AST
    • **Implement evaluation, converting expressions into values (98% of assignment)**
  • Suggested: Do each of these 3 tasks for one syntactic form at a time, rather than writing all of
    the AST types, then all of the Ast_factory functions, and then all evaluation code
    • You can do this in scope order
A9 DELIVERABLES

- Zip file, created by `make zip`, which requires work in:
  - `[ast.ml]`: Define types for different structures in the JoCalf AST
  - `[ast_factory.ml]`: Convert parser output to your (newly defined) AST types
  - `[eval.ml]`: Implement evaluation of all different forms of JoCalf expressions; define value type
  - `[test.ml]`: Test suite for everything you implement
  - `[authors.ml]` and `[authors.mli]`: Assignment metadata.
JoCalf Overview

- Top-level statements can be *definitions* (like top-level let assignments) or *expressions*

Language Features Include:

- Integer, boolean, and string constants; a special value “undefined” that comes up a lot
- Overloaded binary operations (e.g. +, -, *, >, =); short-circuiting && and ||
- Variables; variable assignment via let expressions and functions
- If statements, with an optional else case
- Looping via recursive let expressions and while loops
- Exceptions that carry one value and can be manually thrown and caught; “finally” syntax on catching
- Mutability via references and objects (mutable mapping from fields to values)
- Built-in functions (externs), for type reflection, string length, and object field checking
• Make some variants to represent different types of AST nodes
  • The abstract syntax in BNF form can act as a rough guide for the types of forms required
  • Try to combine similar / identical syntactic forms into one variant case
    • e.g. do you need separate ones for if else vs. if with no else?
  • That being said, some similar-looking syntactic forms cannot be combined
  • Make decisions that make your life easiest

• It's totally fine to have repetitive variable names
  • Just don’t have duplicate constructor names as the OCaml compiler will get confused
Ast_factory Module

• Convert parser output into AST nodes
• Most functions should be very straightforward
• The inputs coming from the parser are usually “pre-processed” – for example:
  • Do not have to handle hex, octal, binary conversion
  • Object field access syntactic forms e1.e2 and e1[e2] are combined
• Look at <ast_factory.mli> to see what outputs from the parser you get as input
• Most difficult part here is handling integers, which are given as strings from the parser:
  • Make sure you can handle min_int, (which is not out of bounds)
  • Make sure you can handle -(-5), or that sort of structure
Eval Module

• Implement the JoCalf big step semantics to convert an AST node into a [value]
  • You will have to expand the [value] type definition

• Behold the power of variants and recursion: the overall structure of Eval should be very clean

• Big step semantics depend on an environment and a state, for which you must make types
  • Environment is mapping between variable names and values (mutable for backpatching)
    • Altered by let definitions, let expressions, and functions calls
    • Functions are stored as closures: function body + argument names + environment at its definition
  • State is mapping between locations (abstraction of memory locations) and values, for mutability
    • Altered by references
Eval Module – Tips

• Abstract functionality into helper functions

• When implementing a single syntactic form:
  • Read through the relevant section in the JoCalf Manual
  • Consult the formal semantics for more precise definitions

• Some syntactic forms have a lot of edge cases. Testing should help here

• When implementing recursive let statements, you use a “backpatching” strategy
  • See section 8.12 of the textbook
  • Try to implement a recursive function in OCaml without the let keyword first via this strategy
  • Then, do the same sort of process but in JoCalf evaluation
Final Tips

- JoCalf exceptions behave a lot like exceptions in other languages, including OCaml
  - If an exception is thrown in JoCalf or OCaml, it propagates “upward” until it is caught
    - You can look at the formal semantics to verify this behavior
    - Maybe you can leverage this similarity to greatly reduce the complexity of exception handling
- There is a simple way to implement locations
  - Just make sure distinct locations are represented differently (have different values)
- Some syntactic forms are grindy to implement; others are more conceptually difficult
  - Plan accordingly