Today:

- Writing a computer language, part 2
  - Evaluating the AST
  - Environments and scoping
Where we're at

- We've implemented the first part of a language interpreter
  - source code $\rightarrow$ tokens (lexing)
  - tokens $\rightarrow$ S-expressions (parsing)
  - S-expressions $\rightarrow$ abstract syntax trees (ASTs) (also part of parsing)

- This is the boring (routine) part of writing an interpreter
Where we're going

- Today, we'll look at the process of evaluating the ASTs produced by the lexing/parsing process
- Our programs will go through the parser and will be transformed into a sequence of AST expressions
- We will write an evaluator that can generate a value for any AST expression
Overview (1)

- Program $\rightarrow$ [parsing] $\rightarrow$ sequence of AST expressions
- For each AST expression,
  - evaluate the AST expression to give a value
- That's all there is for a simple interpreter!
- More complex interpreters/compilers may transform the AST into simpler representations (often called intermediate representations or IRs)
  - compilers may go all the way to machine language
Overview (2)

- Type signature of evaluator (in `eval.mli`):

  ```plaintext
  val eval : Ast.expr \rightarrow Env.env \rightarrow Env.value
  ```

  This says: take an AST expression and an "environment" and produce a "value"

- What are environments?
- What are values?
Environments and values (1)

- **Values** are the possible legal values that AST expressions can evaluate to.
- **Environments** are a data structure that stores the mappings (bindings) between identifiers in the language and their values.
Environments and values (2)

Values and environments are mutually-recursive types:

```ocaml
type id = string (* identifiers *)
type value = (* values *)
  | Val_unit
  | Val_bool of bool
  | Val_int of int
  | Val_prim of (value list -> value)
  | Val_lambda of env * id list * Ast.expr list
and env = (* environments *)
  { parent: env option;
    bindings: (id, value) Hashtbl.t }```

Values (1)

- Values represent the different possible results of a computation:
  - `Val_unit` -- unit value (#u)
  - `Val_bool` -- boolean value (#t or #f)
  - `Val_int` -- integer value
  - `Val_prim` -- built-in (primitive) function
  - `Val_lambda` -- user-defined function
Values (2)

Val_primin of (value list -> value)

- Represents built-in functions:
  - +, −, *, /, <, >, etc.

- Built-in functions take a list of values (evaluated arguments) and return a single value.
Values (3)

- **Val_lambda** (lambda expression) is particularly interesting:
  
  ```scheme
  Val_lambda of env * id list * Ast.expr list
  ```

- **Ast.expr list** is just a list of Scheme expressions in the body of the lambda
  - usually just one expression
  - if more than one, evaluate them in order

- **id list** is the list of identifiers making up the formal argument list of the function

- **env ...**
Values (3)

- **Val_lambda** (lambda expression) is particularly interesting:
  - `env` is the environment in which the lambda expression was defined
  - lambda expressions "carry their own environments around with them"
  - This is called **lexical scoping** and has many uses
Lexical scoping (example)

(define adder
  (lambda (n)
    (lambda (i) (+ n i))))
(define add3 (adder 3))

- Here, \texttt{add3} is bound to the lambda expression \texttt{(lambda (i) (+ n i))}

- This wouldn't make sense unless there is an environment that maps \texttt{n} to something

- That environment is the one that was active when \texttt{(lambda (i) (+ n i))} was defined
Environments (1)

- Recall:
  
  ```ocaml
  and env =
  
  { parent: env option;
  bindings: (id, value) Hashtbl.t }
  ```

- Environments bind names (identifiers, \texttt{id}) to values (\texttt{value})
  
  - here, we use an Ocaml hash table in the implementation

- Environments may have a "parent environment"
  
  - here, we use an \texttt{env option} type
Environments (2)

- Environments are used to store bindings between identifiers and values and to look up the value corresponding to a given identifier.

- How to look up a value in an environment:
  1) Look it up in the *bindings* hash table
  2) If it's found there, return the corresponding value
  3) If it isn't found there, search the *parent* environment
  4) If there is no parent environment, signal an error (raise an exception)
Environments (3)

- Ocaml hash tables are a data structure in the Ocaml standard library.
- Look up hash tables in the Ocaml documentation.
- Hash tables are **not** a functional data structure.
  - They are imperative.
- In lab 6, the only part of the code that cares about hash tables is inside the file `env.ml`.
- `env.mli` has the interface to the `env` type, which doesn't mention hash tables at all.
  - `env` is an abstract data type.
Writing the evaluator (1)

- Type of the evaluator function (from `eval.mli`):
  ```ocaml
  val eval : Ast.expr -> Env.env -> Env.value
  ```
- `Ast.expr` is the expression to be evaluated
- `Env.env` is the environment in which the expression is evaluated
  - This provides bindings for any free (unbound) variables
  - *Evaluation only makes sense in the context of some environment!* We call this the "current environment"
- `Env.value` is the result of evaluating the expression
Writing the evaluator (2)

- Type of AST expressions:

```ocaml
type id = string

type expr =
  | Expr_unit
  | Expr_bool of bool
  | Expr_int of int
  | Expr_id of id
  | Expr_define of id * expr
  | Expr_if of expr * expr * expr
  | Expr_lambda of id list * expr list
  | Expr_apply of expr * expr list
```
Writing the evaluator (3)

- Literal expressions:
  - `Expr_unit`
  - `Expr_bool of bool`
  - `Expr_int of int`

- These are easy to evaluate
  - `Expr_unit` always evaluates to `Val_unit`
  - `Expr_bool` evaluates to corresponding `Val_bool`
  - `Expr_int` evaluates to corresponding `Val_int`

- These expressions don't depend on the environment
id and define expressions do depend on the environment

| Expr_id of id
| Expr_define of id * expr

To evaluate an Expr_id expression:

- look up the identifier (id) in the current environment and return the value
- if the identifier isn't found, an exception will be raised
id and define expressions do depend on the environment

| Expr_id of id
| Expr_define of id * expr

To evaluate an Expr_define expression:

- evaluate expr in the current environment to get a value
- add a binding between the identifier id and this value in the environment
- return a unit value (Val_unit)
Writing the evaluator (6)

- `id` and `define` expressions `do` depend on the environment
  
  | Expr_id of id |
  | Expr_define of id * expr |

- **NOTE:**

- The evaluator doesn't contain any code for searching environments or adding new bindings to environments
  
  That code is in `env.ml` and `env.mli`

- The evaluator code simply calls those functions
Writing the evaluator (7)

- `Expr_if` of `expr * expr * expr`

To evaluate an `Expr_if` expression:

- evaluate the first `expr` (which should evaluate to a boolean (Val_bool) value)
- if the first expr evaluated to `Val_bool true`, evaluate the second expr; that value is the value of the entire `Expr_if` expression
- if the first expr evaluated to `Val_bool false`, evaluate the third expression; that value is the value of the entire `Expr_if` expression
- Never evaluate both the second and third `exprs`!
To evaluate an `Expr_lambda` expression:

- create a `Val_lambda` value with the same `id list`, the same `expr list`, and the current environment as the environment `(env)` part

- That's all!
Writing the evaluator (9)

<table>
<thead>
<tr>
<th>Expr_apply of expr * expr list</th>
</tr>
</thead>
<tbody>
<tr>
<td>This represents a function application (applying a function to its arguments)</td>
</tr>
<tr>
<td>This is by far the most complex case</td>
</tr>
<tr>
<td>expr represents the function, which is either a built-in function or a lambda expression</td>
</tr>
<tr>
<td>expr list represents the arguments to the function</td>
</tr>
</tbody>
</table>
Writing the evaluator (10)

```
Expr_apply of expr * expr list
```

- First step: evaluate the `expr list` by evaluating each `expr` in the current environment and making a list of the results in the same order as the `exprs`
- The result will be a list of values
- This is called strict evaluation: all function arguments are evaluated before applying the function to its arguments, even if the function doesn't need all of the values
Writing the evaluator (11)

- `Expr_apply` of `expr * expr list`

- Second step: evaluate the `expr`

- The result will be either
  - a built-in function (`Val_prim`)
  - a lambda value (`Val_lambda`)
  - some other value

- If the result is anything other than a `Val_prim` or a `Val_lambda`, it's an error and a `Type_error` exception should be raised
If the `expr` evaluates to a `Val_prim`:
- recall that `Val_prim` values have the function type `value list → value`
- the result of evaluating the `expr list` is a list of values (`value list`)
- so just apply the `Val_prim` function to the `value list` to get the `value` (the result)
Writing the evaluator (13)

| Expr_apply of expr * expr list
| If the expr evaluates to a Val_lambda:
  | create a new environment with these attributes:
    | the parent environment is the env of the Val_lambda (not the current environment!)
    | the bindings consist of the identifiers in the id list of the Val_lambda bound to the list of values from the evaluated arguments to the function (so, if the id list is x, y, and z and the values are 1, 2, and 3 then the bindings would be x \rightarrow 1, y \rightarrow 2, and z \rightarrow 3)
Writing the evaluator (14)

- Expr_apply of expr * expr list

  - If the expr evaluates to a Val_lambda:
    - evaluate the expr list of the Val_lambda in the context of the new environment you just created
    - return the value of the last expr in the expr list
  - That's all!
Lab 6

- Lab 6 is basically identical to the material in this lecture
- A lot of code is provided for you, as in lab 5
- You'll need to copy your working parser from lab 5 into your lab 6 submission
- Other than that, there's only about 40 lines of code to write, in two files