Rackette, Records
Overall Rackette structure

- Raw program text
- Concrete program
- Abstract program
- Values Modified envt
- Printed program result

Steps:
- "read" Raw program text
- "parse" Concrete program
- "process" Abstract program
- "print" Values Modified envt

Flow:
- Raw program text → Concrete program → Abstract program → Values Modified envt → Printed program result
- Feedback loop: Values Modified envt → Initial envt
One step at a time

- Already done!
One step at a time

- Need to go from concrete program pieces (Num, Symbol, List) to abstract program pieces (definition, or-expression, ...)
- Almost entirely pattern matching
- Easier once you have the “types” defined
One step at a time

• Too tough to think about all at once!
One step at a time

- Build an initial environment
- Must contain all built-ins
- Must contain definitions for “true” and “false”
- Must be an “environment” so that we can add things and can look up things
One step at a time

• Again, too complicated
One step at a time

Raw program text → Concrete program → Abstract program → Values Modified envt → Printed program result

“read”  “parse”  “handle definitions”  “handle expressions”  “print”

Initial envt

• Again, too complicated
“Handling” an expression means taking as input an expression and an environment, and producing a value; that’s called “evaluation.”

The process is laid out almost algorithmically in the “rules of evaluation.”

Code one piece at a time.

Example: how do you evaluate a number-expression like `NumE 4`?

Produce as output the value `VNum 4`!

One line of your “evaluate” procedure:

```
| NumE n -> VNum n
```
“Handling” an definition means taking as input a definition and an environment, and producing as output a new environment that contains the defined thing.

Typical input (informally): `(ID “x“, ApplicationE (ID “+“, NumE 3, NumE 2), <some envt>)`

Typical output (informally): input envt with binding `(ID “x“, VNum 5)` appended
• Printing is nice and easy!
• VNum 2 → “2”
• Typical code fragment:
  \[ VNum \ n \rightarrow \text{string} \_\text{of} \_\text{int} \ n \]
• It’s such a nice piece that it’s a really good place to start the project, so that you can feel you’ve made progress.
Typedef reminders: the "read" part

type raw_program = string

type concrete_program_piece =
    Number of int
    | Symbol of string
    | List of concrete_program_piece list

type concrete_program = concrete_program_piece list
The “parsing” part: consume a concrete program, produce an abstract program

type abstract_program_piece =
| Definition of definition
| Expression of expression ;;
type abstract_program = abstract_program_piece list ;;
Details: definitions

type identifier = ID of string ;;
type definition = identifier * expression ;;
type expression =
| NumE of int
| IdentE of identifier
| AndE of expression * expression
| OrE of expression * expression
(* | IfE of your definition for if expr goes here *)
| CondE of (expression * expression) list
| QuoteE of concrete_program_piece
| LambdaE of identifier list * expression
| ApplicationE of expression list ;;
The “defining things” part: build environments. Need name-value pairs.

type value =
  | VNum of int
  | VBool of bool
  | VSymbol of string
  | VList of value list
  | VBuiltin of string * (value list -> value)
  | VClosure of identifier list * expression * environment

type binding = identifier * value

type environment = binding list ;;
Last time

- Things to put in the initial “top level environment” (TLE)

```ocaml
let initial_tle : environment = [
  (ID "true", VBool true);
  (ID "false", VBool false)
  ... ] ;;
```

Also need a binding for “not”

```
(ID "not", VBuiltin <string * (value list -> value>)
```
let not_func: <type> = function
...
(ID "not", VBuiltin <string * (value list -> value>)

(ID "+", ("<builtin:not>", not_func))

let not_func: value list -> value = function
| ... -> ...
| ... -> ...

...
let not_func: value list -> value = function
| [] -> failwith "Need an argument for 'not'"
| ... -> ...

(ID "not", VBuiltin <string * (value list -> value)>)

(ID "not", ("<builtin:not>", not_func))
(ID "not", VBuiltin <string * (value list -> value>))

(ID "not", ("<builtin:not>", not_func))

let not_func: value list -> value = function
   | [] -> failwith "Need an argument for 'not'
   | [VBool arg] -> ...
(ID "not", Vbuiltin <string * (value list -> value>)

(ID "not", ("<builtin:not>", not_func))

let not_func: value list -> value = function
| [] -> failwith "Need an argument for ‘not’"
| [VBool arg] -> VBool (not arg)
| ...
let not_func: value list -> value = function
| [] -> failwith "Need an argument for 'not'"
| [VBool arg] -> VBool (not arg)
| _ -> "Too many arguments given to 'not'" ;;

N.B.: This code incorrectly handles an input like (not 3)
Can you see why?
Records in OCaml

```ocaml
type point =
{
  x : float ;
  y : float ;
} ;;

let my_point = {x = 5.; y = 3.} ;;
my_point.x + my_point.y ;;
val - = 8.0 : float

• Really just “and types” (i.e., float * float) in disguise
• Can be much easier to read
• You might choose to use them in Rackette... or not.
```
Challenges with records

• If you make two records with similar field-names, it can be tough to remember what’s what
  • Also tough for OCaml, which can get confused without explicit type labels
• Pattern-matching has its own syntax, which you’ll need to learn (but can probably guess)

match p with
| { x = foo; y = bar} -> (* expr using foo and bar *)
Quiz

let string_of_value : value -> string = function
  | VNum n -> string_of_int n
  | VBool b -> ...
  | VSymbol s -> ...
  | VList -> ...
  | VBuiltin (s, _) -> ...
  | VClosure _ -> ... ;;
let string_of_value : value -> string = function
  | VNum n -> string_of_int n
  | VBool b -> string_of_bool b
  | VSymbol s -> s
  | VList -> ...
  | VBuiltin (s, _) -> ...
  | VClosure _ -> ... ;;