Objectives

- In this lecture, we’ll talk about the sequence of steps and the skills you’ll need to do Rackette

1 Introduction to Rackette

Most of today’s lecture notes will be on Rackette. There are two reasons we need to start this. You’re going to see how some of what you’ll doing in Eliza will be relevant for the next project. Also, you have a lab this weekend that does a little part of Rackette as a warmup. We want to give you an introduction in class first.

Let’s start with the main rackette program. The type signature for the main rackette procedure will be: rackette:: string -> string.

It will take in Racket code in the form of a string, and output what the interactions pane of DrRacket would output, likewise in the form of a string.

A typical input to Rackette may be:
Note, however, you are writing a Racket interpreter, not a Racket interpreter; there are some simplifications in Rackette. A program in Rackette is a sequence of definitions followed by zero or one expression. In Rackette, the data types are int, bool, proc. There are no strings and no comments, and there are relatively few builtins. You will have to implement arithmetic builtins such as + and -, boolean builtins such as not, and list builtins such as cons and first. Builtins such as + need to only take two arguments in your implementation. Additionally, you will implement syntax including define, true, false, empty, if-expressions, cond-expressions, and-expressions, or-expressions, lambda-expressions, and procedure-application expressions. You don’t have to include let* or letrec (but it’s fun!). There will probably be more simplifications that you’ll see later.

Processing a Rackette program involves a sequence of transformations from raw text which represents the program as a string to a final printed value.

The sequence takes several steps:

1. read: convert raw text into a concrete program, described in OCaml by

   ```
   type concrete_program_piece = Number of int | Symbol of string | List of (concrete_program_piece list)
   type concrete_program = concrete_program_piece list
   ```

   The type signature of read is `read: string -> concrete_program`.

2. parse: convert a concrete program into an abstract program, which is described in OCaml by a substantially more complex set of datatypes. But at the highest level, an abstract program is a list of abstract program pieces, each of which is either a definition or an expression. And parsing will work on a per-piece basis, i.e., to parse a concrete program, we will apply a parse_piece procedure to each concrete program piece to get a corresponding abstract program piece.

3. process: run the rules of processing/evaluation on the abstract program to gradually update various environments, etc., and to compute values for top-level expressions, producing as a result a list of values, one per top-level expression.

4. print: for each value produced by processing, print out a printed representation of that value.

2 Parse

The following examples show strings being transformed to concrete programs by `read`, which may or may not succeed. Which ones work out? (Answers are in comments at the right.)

```scheme
(read "(+ 1 2)"; (* [List [Symbol "+"; Number 1; Number 2] ] *))
(read "(define a 5)"; (* Failure: syntax error *))
(read "(define a 5)"; (* [List [Symbol "define"; Symbol "a"; Number 5] ] *))
(read "(define a 5) a"; (* [List [Symbol "define"; Symbol "a"; Number 5]; Symbol "a"] *))
```
Let’s look at the type `concrete_program`, which is defined for you in the file containing the definition of `read`:

```haskell
  type concrete_program_piece =
  | Number of int
  | Symbol of string
  | List of concrete_program_piece list ;;
  type concrete_program = concrete_program_piece list
```

The `parse` procedure has to handle a concrete program, which is a list of concrete program pieces, each of which should parse as either a definition or an expression. The real work is therefore in the `parse_piece` procedure.

From the given definition, we know `parse` should look something like:

```haskell
  parse (input: concrete_program) : abstract_program =
  List.map parse_piece input;;
```

```haskell
  parse_piece (input: concrete_program_piece) : abstract_program_piece =
  match input with:
  | Number n -> ...
  | Symbol s -> ...
  | List lst -> ...
```

What goes on the right hand side of our matching depends on what an `abstract_program` looks like, so let’s define it and make sure it’s a reflection of the BNF that defines Rackette:

```haskell
  type ident = Ident of string
  type expression = ...
  type definition = ident * expression
  type abstract_program_piece = Def of definition | TLExp of expression
  type abstract_program = abstract_program_piece list
```

So … what’s an expression? Well, there’s a lot of them. There are `if` expressions, there are `and` expressions, there are lambda expressions, there are `let` expressions, there are constants, there are identifiers, … there’s a lot of stuff!

Let’s write that all down:

```haskell
  type expression =
  | IfE of if_expr
  | ...
  | ...
  | ...
  | ... and
  if_expr = expression * expression * expression
```

Why is an if-expression made up of three expressions? Look at `((= x 3)2 5)`. The keyword ‘if’ tells us that this is an if-expression, but it’s then followed by three further expressions. It’s these three (or representations of them) that we store in our representation of an if-expression!

**Note:** You may be wondering what that `and` is doing in the middle there. Don’t worry, we’ll come back to that.
Let’s return for a moment to that parser. How is it going to “recognize” an if-expression, or a definition, or anything else?

```
parse_piece (input: concrete_program_piece) : abstract_program_piece =
    match input with:
    | Number n -> TLExp(Num(n))
    | Symbol s -> TLExp(...)
    | List lst -> <this is the interesting part!>
```

So what happens when we encounter a “list” (i.e., something in parens in the input string)? It’s probably either an if-expression, or a definition, or a proc-app expression, or ... How can we decide which? We look at the very first item in the list.

```
| List lst -> match lst with
|   [] -> failwith "Empty parenthesized expression not allowed"
|   (Symbol "define") :: tl -> match tl with ...
|   (Symbol "if") :: exp1 :: exp2 :: exp3 :: [] -> ...
|   (Symbol "if") :: _ -> failwith "Malformed if-expression!"
|   ...
```

And in that “if” case: what do we do with the three exps? Note that the names are a little misleading. They are really pieces of concrete program that we expect will turn out to be expressions. The answer is that we want to form an if_expr to represent this thing (assuming all works out).

```
| List lst -> match lst with
|   ...  
|   (Symbol "if") :: exp1 :: exp2 :: exp3 -> IfE ((parse_piece exp1), (parse_piece exp2), (parse_piece exp3))
|   ...  
```

And to do that...we recursively call parse_piece.

## 2.1 What’s tough about parsing?

What makes this part of Rackette tough is that there are a lot of cases to deal with: if-expressions, and-expressions, cond-expressions, etc. The good news is that you can test them one at a time: you get parsing of if-expressions working, and if you feed in a definition, you just get back a report that it’s (as of yet) not a valid program. The other challenge is that you have to keep straight the difference between a piece of a concrete program and a piece of an abstract program. That Number that showed up as a result of read is not an expression. You need to convert Number 3 into NumE 3, where NumE is the constructor for number expressions, which are one of the cases of expression, but Number is the constructor for one of the kinds of concrete_program_piece.

## 2.2 What’s easy about parsing?

You don’t have to worry about what anything means during parsing. If an if-expression starts with the keyword if and is followed by three things that can all be parsed as expressions, you don’t have to worry about whether the first one will evaluate to a boolean or not: that’s the job of the “processing” phase! You just have to check that there are three expression in the place where they should be.
3 Recursive Type Definitions

You may be wondering what that “and” is doing in the middle of our type definition for expressions:

```
type expression =
  | IfE of if_expr
  | AndE of and_expr
  | ...

if_expr = expression * expression * expression
```

Well, the type definition for `if_expr` needs to know what an `expression` is, but `expression` needs to know what `if_expr` is. These two types are mutually recursive! Using the keyword “and” allows for mutually recursive types.

You can use the keyword `and` for writing mutually recursive `functions` too, and we’ll be doing that a lot with evaluation and parsing.

```
let eval ... =
  match ... with
  | If i -> eval_if_exp(i)
  | And a -> eval_and_exp(a)
  and

  eval_if_exp (i: if_expr) =
  match i with
  | (test, yes, no) -> let v = eval(test) in (match v with ...
```

Between every pair of functions in a collection of mutually recursive functions, you need an “and.” And there needs to not be a doublesemicolon before the “and”, or OCaml will get very confused.

3.1 Testing Parse and Eval

Both “parse” and “eval” are somewhat complex procedures. Testing them a step at a time is a big help. The project handout recommends a strategy for this, but here’s one hint:

Make sure you first test your base cases! Test on things like constants in order to ensure your base cases are correct. There’s going to be a lot of recursion going on, so it’s crucial to work bottom up.

4 Some Rackette Built-ins

These are some of the built-in procedures you’re going to need to be able to implement to make Rackette work:

- `+`, `-`, `*`, `/`
- `empty?`, `cons`
- `cons`
- =, <=, >=, <, >
- equal? not

And here are some keywords that you’ll have to watch out for:

- empty
- true, false
- define
- if, and, or
- lambda
- let
- cond

Things like map, reverse, and append won’t be built in, but you’re more than welcome to define them yourself if you’d like to.

5 Processing

Once you’ve got an abstract program

\[
\text{type abstract\_program\_piece} = \begin{cases} \text{Def of definition} \\ \text{TLExp of expression} \end{cases} \\
\text{type abstract\_program} = \text{abstract\_program\_piece list}
\]

how do you process it?

Similar to parsing, we process it once piece at a time. However, it’s not quite as simple as parsing, because now, we need both a program and an environment. This is because each step of processing either changes the environment (through definitions) or produces a value (through evaluating a top-level expression). Handling definitions involves evaluating expressions, so let’s talk about evaluation first.

We evaluate expressions in the presence of an environment in which we can look things up. Generally, this is the top level environment, but sometimes it’s the top level environment plus a local environment that has additional bindings. In our eval procedure, we’ll pass in two environments: the top level environment, and the local environment.

\[
\text{eval: environment * environment * expr -> value}
\]

Here’s a simple case of evaluation with the environments omitted.

\[
\text{let rec eval envs-omitted (input: abstract\_program) =}
\]
match input with
| NumE (n) -> VNum n
| ProcAppE (...) ->

6 Printing

Printing involves printing out a printed representation of each value produced by processing.

Here's what the print procedure looks like

let rec print (myVal: value) =
match input with
| VNum(n) -> string_of_int n
| VBuiltin ... ->
| ...

6.1 Unit

Print takes in a value and produces a string. But what if we wanted to write a print procedure that doesn’t produce a value, but just prints something out?

What type does “print something out” have? This type would be unit. unit is the type for “no value at all”.

The type signature for a procedure print_int that takes in an integer and prints it out instead of producing a value would be print_int: int -> unit

print_int is different from other procedures we’ve seen because it has a side effect, which is displaying something in the window in which you’re running OCaml.

If we start the OCaml REPL in terminal, and start typing

print_int 5;;
=> 5: unit = ()
3 + (print_int 5);;
=> Error: this expression has type unit but an expression was expected of type int

When we type print_int 5;, OCaml prints out the printed representation of 5 on the screen, and then it prints out the type of the result of evaluating print_int, which is unit. Then when we try to type 3 + (print_int 5);;, this results in an error, because we tried to add an int to an unit.

Notice that when we typed print_int 5;, it printed 5: unit = () without a new line. How would we print 5 followed by a new line?

print_int 5; print_string "\n"
=> 5
    -: unit ()
The ; here is an operator in OCaml. The type signature for ; is 'a -> 'b -> 'b. It takes two arguments and forgets its first argument and produces just its second argument. Although it can be applied to any 'a or 'b, it gives a warning if the first argument is not of type unit.

```
5; 3;;
=> Warning: this expression should have type unit.
-.: int = 3
```

The purpose of ; is to print sequences of strings in OCaml. This can be used along with printing functions such as print_int that print out things as side effects to debug procedures in OCaml and will be helpful in future projects like Game.

### 7 Summary

#### Ideas

- Parsing takes a concrete_program (which is a string of raw text representing a Rackette program) and turns it into an abstract_program, which reflects the actual structure of our Rackette program (i.e. what expressions and definitions it is made of).

- Evaluation converts an abstract_program to our representation of values by looking at each expression and following the rules of evaluation until a value is produced.

- If you are trying to determine what kinds of expressions should be valid abstract_program_pieces in your language, the BNF for the language is a great place to start. For instance, taking a look at Rackette's BNF would tell us that an if-expression needs to be a valid abstract_program_piece, and that an if-expression is really just the keyword if followed by three more expressions.

#### Skills

- The keyword and can be used within a recursive procedure to allow for mutual recursion (where two procedures can call each other recursively)

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)