Lecture 22: More Rackette
10:00 AM, Oct 25, 2019

Contents

1 Introduction to Rackette 1

2 Parse 2
   2.1 What’s tough about parsing? 4
   2.2 What’s easy about parsing? 4

3 Processing 5

4 Printing 5

5 VSCode 6

6 Summary 6

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.

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.

Doing this is difficult and none of you should know how to do this already. This means the handout for rackette is very long because it is guidance for every single step you have to take.

A typical input to Rackette may be:

\begin{verbatim}
(define (f x) (+ x 1))
(f 3)
\end{verbatim}

Note, however, you are writing a Rackette 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. There is a REPL: a read-eval-print loop.

The sequence takes several steps:

1. read: For Racket reading can be done in two steps: check input matches BNF for Rackette then refine. Trying to read programs in other languages is challenging but the parentheses in Racket makes it easier. We provide read for you it converts raw text into a concrete program, described in OCaml by

```ocaml
type concrete_program_piece = Number (int) | Symbol (string) | List ( list concrete_program_piece ) ;

type concrete_program = list (concrete_program_piece);
```

The type signature of read is read: string -> concrete_program. What happens if you type the declaration of Independence into Drracket? An error should occur, similarly, Rackette should only deal with valid code.

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.)

```ocaml
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`:

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

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:

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

```plaintext
parse_piece (input: concrete_program_piece) : abstract_program_piece =
    switch (input) {
        | 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:

```plaintext
type name = | ID(string);
type expression = ...
type definition = name * expression
type abstract_program_piece = def(definition) | TExp(expression)
type abstract_program = list (abstract_program_piece)
```

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:

```plaintext
expression =
    | Num(int)
    | Bool(bool)
    | Name(name)
```
And(expression, expression)
| Or(expression, expression)
| If(expression, expression, expression);

Why is an if-expression made up of three expressions? Look at (if (= 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 =
  switch (input) {
  | 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 => switch (lst) {
  | [] => failwith "Empty parenthesized expression not allowed"
  | [(Symbol "define"), ... tl] => switch (tl) { ...
  | [(Symbol "if"), exp1, exp2, exp3, ... []] => ...
  | [(Symbol "if"), ... _ ] => failwith ("Malformed if-expression!");
  | ... ;

Why are we switching on tl? We have to parse the tl separately and specifically it has to have two parts: a name and an 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 => switch (lst) {
  ... 
  | [(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 Processing

Once you’ve got an abstract program

```scheme
type abstract_program_piece =
  | DEF(definition)
  | TLExp(expression) ;
type abstract_program = list (abstract_program_piece);
```

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.

```scheme
eval: (environment, environment, expr) => value
```

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

```scheme
let rec eval envts-omitted (input: abstract_program)=

switch (input) {
  | NumE (n) => VNum(n)
  | ProcAppE (...) =>};
```

In your eval, you will have to look things up in the local environment and then the top level
environment.
4 Printing

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

Here’s what the print procedure looks like

```racket
let rec print (myVal: value) =
  switch (input) {
  | VNum(n) => string_of_int(n)
  | VBool(true) => "true"
  | VBool(false) => "false"
  | VBuiltin ... =>
  | ... };
```

5 VSCode

Spike demoed how to use VScode in class. Note that bsb stands for bucklescript build. We can use this to create reason projects. After you initialize your new project, we care about the folder called: src where we write programs. After editing, make sure you save the program. Then in the terminal write "bsb -make-world" and the terminal will show errors if they exist or build the program and create a javascript file. To run it, write node <file name>. Also good to know if you put your cursor over a certain part of the program it will tell you what type it is. This is helpful for debugging.

6 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.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).