Contents

1 Recursive Variant Types ................................................................. 1

2 Birth environments of procedures ................................................ 1
   2.1 Closure .................................................................................. 2

3 Introduction to Rackette ................................................................. 3
   3.1 Parsing .................................................................................. 3

4 Summary ......................................................................................... 4

1 Recursive Variant Types

Last time we saw examples of variant type definition, e.g.

type transport =
    Van
    | Pickup_truck
    | Freight_truck of int

There was one more thing you can do with variant type definitions: the definition can be recursive. You see this in the current homework:

type primitive =
    | Add
    | Sub

type expression_tree =
    Int of int
    | Prim of primitive * expression_tree

This reflects the fact that an expression can be built out of smaller expressions. You will see this in the project coming out, Rackette.
2 Birth environments of procedures

From last time:

input: increment, an integer
output: a procedure that, given an integer x, returns x+increment

Answer: (lambda (increment) (lambda (x) (+ increment x)))

We think a procedure value has two parts:

- the argument list, in this case (increment)
- the body, in this case (+ increment x)

value is a procedure with the following spec:

- input: an integer x
- output: x + 7

Answer: ((lambda (increment) (lambda (x) (+ increment x))) 7)

What are the two parts of the procedure that is the value of this expression?

- the argument list is (x)
- the body is (+ increment x)

When this procedure is applied to something, say, 10, what happens? The symbol x is bound to 10, and the body is evaluated:

> (( (lambda (increment) (lambda (x) (+ increment x))) 7) 10)
17

Is there something a little mysterious about this? The symbol increment in the body—we know it was bound to 7 but how does that binding manage to affect the evaluation of the expression (+ increment x)?

The answer is that a procedure data object has not two but three parts: the arg list, the body, and the birth environment.

The birth environment of a procedure object is the environment consisting of the local variables that are the procedure’s arguments, which are defined locally within the procedure’s scope, but not outside of it. The concept of birth environments will be very important as we begin our next project: Rackette!
2.1 Closure

Let’s say we want to create a procedure increase_salary which adds 10 to the salary of the CS17 TAs:

\[
\text{(define increase_salary}
\quad \text{(lambda (L) (map (lambda (x) (+ x 10) L ))))}
\]

So we don’t have to constantly change 10 to whatever amount we desire, let’s change this to a procedure that can add any amount to the list, so have it take in two arguments:

\[
\text{(define increase_salary}
\quad \text{(lambda (increment L) (map (lambda (x) (+ x increment) L ))))}
\]

Here we are using the map procedure to carry out the computation, and that procedure is given a procedure value.

Map does not know anything about increment. So, when map is applied, how does the procedure know what to add? It is because the environment includes the binding of increment to 20. Here, we are using the notion of closure.

3 Introduction to Rackette

Let’s start with the main rackette program. The type signature for the main rackette procedure will be: \text{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:

"(define x 3) (+ x 1)"

=> "4"

Note, however, you are writing a Rackete interpreter, not a Racket interpreter; there are some simplifications in Rackete. For example, the only numbers are integers. You also don’t have to include every “built-in” that we’ve used in Racket. Built-ins such as + need to only take two arguments in your implementation. You will implement \text{if}-expressions, \text{cond}-expressions, \text{lambda}-expressions, and procedure-application expressions.

3.1 Parsing

The first step in breaking down something that takes in text as input is a process calling \text{parsing.}
Parsing involves taking a sentence or a program and identifying the major parts.

In the case of a sentence, you might identify the subject and predicate, for example. In Rackette, “parsing” means looking at the input text and seeing how it lines up with the syntax of a Scheme definition or expression.

Experience shows that this is best done as a multi-step process. First, check that parens match, and identify numbers, strings, and non-number entities. Then, parse the resulting things to be sure they’re legal Rackette.

Parsing ensures that the provided string looks like our definition of what the syntax of Rackette looks like. To simplify this, we’ve written a procedure for you called read. Its type signature is `read: raw_program -> concrete_program_piece`.

Here’s a sample input and output for read:

Input: "(define (f x) (+ x 1)) (f 3)"
Output: `[List [Symbol "define" ; List [Symbol "f" ; Symbol "x"] ; List [Symbol "+" ; Symbol "x" ; Number 1]] ; List [Symbol "f" ; Number 3]]`

Your task will be to take something like this and essentially ask, “Is that a legal Rackette program?”

Each item in a Rackette program is either a definition or an expression. So specifically, we’ll want to ask: does that first thing in the list match the notion of a definition? We know that a definition should begin with the keyword `define`, followed by an identifier or a list. Our example output does in fact begin with `define`, and is followed by a list! (And, as you probably guessed, you can use pattern matching to check this in your implementation!)

4 Summary

- Birth environments will be very important for Rackette and consists of the local variables that are a procedure’s arguments (defined in the scope of the procedure).
- Parsing breaks down parts of a program into the key components.

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]