Assignment 1: A More Featureful Interpreter

In this assignment, you'll take what you've learned from the course so far and use it to build a more featureful interpreter for a language called *Dubdub*. The language will:

- have similar syntax as Racket
- use a left-to-right eager evaluation strategy
- have support for *checking of type contracts* (explained later)
- support automatic currying (like Haskell)

Please review this handout and the general assignment guidelines carefully before starting to write any code!

Due date: October 15, 2019 before 10:00pm

Starter code

- `dubdub.rkt`
- `dubdub_errors.rkt`
- `dubdub_test_sample.rkt`

Dubdub specification

Unlike in exercises 1-4, a Dubdub program will not be a single expression. Instead, the Dubdub grammar is defined as follows (`<prog>` indicates the form of an entire Dubdub program):

```
<prog> = <binding_or_contract> ... <expr> ;

<binding_or_contract> = <binding>    (* Name-value binding *)
                      | <contract>   (* Function contract (more on this later) *)
                      ;

<binding> = "(" "define" ID <expr> ")" ;

<expr> = ID                                      (* Identifier *)
       | INTEGER                                 (* Integer literal *)
       | BOOLEAN                                 (* Boolean literal, #t or #f *)
       | "(" "lambda" "(" ID ... ")" <expr> ")"  (* Function expression / definition *)
       | "(" <expr> <expr> ... ")"               (* Function call *)
       ;

<contract> = "(" "define-contract" ID "(" <con-expr> ... "->" <con-expr> ")" ")" ;
<con-expr> = <expr> | "any" ;
```

Notes on syntax:

- The language keywords are `define`, `define-contract`, `lambda`, `->`, and `any`; they must appear literally as specified by the above grammar.
- `ID` is any valid Racket identifier, except the language keywords.

- `x ...` in the grammar means zero or more `x`'s. For example, a program can have zero or more name bindings, and `lambda` expressions can have zero or more parameters. Note that a function call must have *one* or more subexpressions, since the first subexpression represents the function being called.
- Order matters. For example, all name bindings and function contracts must occur before the program's single top-level expression.
- Dubdub does not permit the `(define (ID <param> ...) <expr>)` function short form.
- For this assignment, a datum representing a program will always be a list, even if it only contains a single expression. See the sample tests in the starter code for details.

The semantics for Dubdub are similar to Racket (e.g., eager evaluation order, lexical scope) except for the following:

- Function calls are automatically curried (more on this later)
- Name bindings may not be recursive; this includes function definitions. (Exercise 4 cookie challenge applies to Exercise 4 only)
- Identifiers in function bodies are checked for whether they are bound only when the function is called.

For example, Racket raises an error when given function definition `(define (f) x)` when `x` is not in scope, even if `f` is never called. Dubdub should only raise an error when evaluating a call to `f`.

- Errors are reported differently (see below).
- Dubdub supports *optional contracts* for its functions. More on this below.

Dubdub builtin functions

Dubdub supports the following builtin functions. Each has the same name as an existing Racket function, and the same behaviour except for `procedure?`.

- `+`
- `equal?`
- `<`
- `integer?`
- `boolean?`
- `procedure?`

For simplicity, you are *not* required to do any error-checking for calls to the builtins. You are also *not* required to support automatic currying on calls to any of the builtins. Instead, when a builtin is called in a Dubdub expression, simply `apply` the corresponding Racket function to the given arguments (but see the exception below). If the arguments are invalid, your interpreter will raise a runtime error—this is **okay**.

Exception: the Racket `procedure?` function should return `#t` when called on a Dubdub function value (created with `lambda`) as well as a Dubdub builtin. Because you'll likely create a new data structure to represent Dubdub functions as closures, your interpreter should wrap calls to `procedure?` so that it properly returns `#t` for your closures.

Dubdub function contracts 限制，可有可无

A *function contract* is a way to specify pre- and postconditions for a user-defined function. A Dubdub function contract takes the form `(define-contract ID (<expr> ... -> <expr>))`, where:

- `ID` is the name of the function this contract applies to (Dubdub does not support defining contracts for anonymous functions or builtins);
- each `<expr>` before the `->` is a unary *predicate* (a function with one argument that returns a boolean) that its corresponding function argument must satisfy, or the keyword `any` (we call these *contract preconditions*);
- the `<expr>` after the `->` is a unary predicate that the return value of the function must satisfy, or the keyword `any` (we call this the *contract postcondition*).

The keyword `any` is used to accept any value. For example, the contract

```
(define-contract f (integer? boolean? any -> procedure?)))
```

specifies that function `f` takes three arguments, an integer, a boolean, and any value, and returns a function.

Contracts aren't just limited to type information! Here is another example of a contract:

```
(define-contract f2
  ((lambda (x) (< 0 x)) (lambda (x) (< 0 x)) -> (lambda (x) (< 0 x))))
```

Function contracts are checked dynamically every time a function is called; you should think of them as a "wrapper" around the function body. Specifically, here are the semantics for a function call in Dubdub:

Function Call 的运算逻辑

1. The leftmost subexpression is evaluated. If its value isn't a function (builtin or closure), an error is raised (see next section for error descriptions).
2. The remaining subexpressions are evaluated in left-to-right order (these are the arguments to the function).
3. *If the function has a contract*, each contract precondition checked against its corresponding argument, in left-to-right order. If any predicate returns `#f`, an error is raised.
4. *If the number of arguments is the same as the number of parameters of the function*, then
   a. the function body is evaluated, with the parameter names bound to the argument values.
   b. *If the function has a contract*, the value obtained from evaluating the body is checked against the contract postcondition; if the predicate returns `#f`, an error is raised.
   c. The value of the function body is returned as the value of the original function call.
5. *If the number of arguments is less than the number of parameters of the function*, then a new function (closure) is created and returned, with the contract for the remaining arguments stored in the closure.
6. *If the number of arguments is greater than the number of parameters of the function*, then an error is raised.

匿名

Here are some notes about how function contracts can be specified:

1. Function contracts are *optional*. A named function doesn't need to have a contract. Anonymous functions (other than those created via currying) never have contracts.
2. A function contract must be defined *before* the definition of the function itself. Other name bindings and function contracts can appear in between a function contract and its definition.
3. The expressions in the function contract are evaluated when the `define-contract` is reached. Check that the expressions evaluate to a builtin or user-defined function. If not, raise an `'invalid-contract` error.

? ->

4. When a named function is defined, and that function has a contract, then The number of contract preconditions must equal the number of parameters from the function declaration. If not, raise an `'invalid-contract` error.
5. Only one contract may be defined per function. If a second contract is defined for the same function, raise an `'invalid-contract` error.

Dubdub Semantic Errors

For this entire assignment, you may assume that all programs are *syntactically* valid, i.e., they follow the Dubdub grammar. However, these programs may have the following semantic errors that you need to check for:

- `'unbound-name`: an identifier is evaluated, but is not bound to a value.

  Note: because recursive bindings are not allowed, trying to define a recursive function could result in this error.

- `'duplicate-name`: an identifier has two top-level name bindings (i.e., two `define`s for the same name), or a function has two parameters with the same name.

  Note: name shadowing *is* allowed, meaning the same name can be bound in more than one scope, e.g. top-level vs. function parameter, or even outer function parameter vs. inner function parameter. Shadowing of builtins is also allowed.

- `'not-a-function`: when a function call expression is evaluated, but the first expression is not a function. For example, the expression `(3 4 5)` should produce this error.

- `'arity-mismatch`: when a function is called with the wrong number of arguments.

- `'invalid-contract`: when a function definition is inconsistent with its contract (see note #3 and #4 from the previous section).

- `'contract-violation`: when a function is called, but its contract is violated (both precondition and postcondition violations raise this error).

When one of these errors is detected, the interpreter must raise an error by calling `report-error`, which you'll read more about in the `dubdub_errors.rkt` starter code.

For simplicity, you do *not* need to worry about the order in which errors are detected. So if a Dubdub program has two different semantic errors, your interpreter should raise just one of them, and you may choose which one.

As we mentioned earlier, you do *not* need to do any error-checking for arguments to builtin functions. You also do not need to automatically curry builtin functions.

Final comment: in many ways this error-handling strategy is quite poor, and certainly wouldn't be used by a real interpreter! We've opted for a simple approach here to reduce your workload on this assignment, but we'll revisit error-handling more generally later in the course.

Part 1: A Racket-compatible Dubdub interpreter

Your first is to write an Racket program that is an interpreter for all parts of the Dubdub language *except* function contracts and currying, which are additional language feature that deviates from Racket semantics, so we've separated that into Part 2 and 3. More precisely, the `run-interpreter` function in the starter code takes a Dubdub program datum and *evaluates* it, returning the value that the program represents, or raising an appropriate error if the program is not semantically valid.

Technical requirements

While all the starter code provided should prove useful, depending on your design and approach, you may want to change some things, which is fine. However, in addition to correctly implementing the operational semantics of Dubdub, your program *must* follow these requirements:

1. Don't change the required signature for the `run-interpreter` or `interpret` functions. The former is is the only public function, and will be used for testing purposes. The latter is the recursive structure we require for this assignment (including using a hash table for the environment), and your TAs will be looking for this.

   However, you may add *optional* arguments to these functions, as long as your documentation gives a good justification for them.

   Also, we'll encourage (but not require) you to use pattern-matching—changing `define` to `define/match` keeps the function interface the same, it just changes the function body.

2. Your interpreter *must* make `lambda` expressions evaluate to a "closure" data structure that contains the function's parameters, body, enclosing lexical environment (necessary for correct implementation of static scope), and contract. It's up to you how you represent this (e.g., using a list, or Racket `struct`), but it should be clear to your grader that this requirement is satisfied.

3. While your interpreter should work correctly when the top-level expression evaluates to a function, because you have the freedom to choose exactly how you represent functions, our tests will only use Dubdub programs whose top-level expression evaluates to an integer or boolean. (But of course, such programs can still define and call plenty of functions as intermediate values!)

Also please note that all the general restrictions for the course apply, e.g., you still may not use mutation, `eval`, or looping constructs. You **are** welcome (and encouraged) to use higher-order functions like `map`, `filter`, `foldl`, `apply` as appropriate. You can also use functions assocsiated with Racket data structures `struct` and `hash`.

Suggested implementation roadmap

The major tip we have is to *follow the recursive structure of the grammar*. Don't get overwhelmed trying to implement everything at once; instead, start from plain integer literals and add more and more language features until you've met all the specifications of the Dubdub language. One easy way to keep track of this is through pattern-matching or good use of `cond`, where every branch (roughly) corresponds to a different grammar rule.

We suggest starting with evaluating a single expression, like in Exercise 4. Start with integers & boolean literals, then move on to built-in function calls, building closures, and function calls using closures. Then, continue to creating name bindings. This part is similar to exercise 3, but the syntax is a bit different.

Other tip: it may be tempting to add error-checking only at the very end. This is actually harder, because you'll have to go through all your code to find the right places to do the checking. The errors we're asking you to check for are tied to specific parts of the syntax, meaning it'll be much easier to implement error-checking as you go.

Part 2: Implementing Currying

Currying is implemented by changing the semantics of what happens when you call a function with fewer arguments than there are parameters. For example:

```
((lambda (x y z) (+ x y z)) 2 3)
```

should return a function (closure) in terms of the remaining argument.

Note: This implementation exercise is *different* from Exercise 3, Task 2. Your currying code from Exercise 3 won't help you. Instead, think about what closure to return. How will the list of parameters, the body of the function, and the environment change?

Make sure that you write good tests at this stage before continuing.

Part 3: Implementing Contracts

We strongly recommend making sure your interpreter works correctly in all other case from Part 1 and 2 before doing this part! This will increase the complexity of your implementation, so you want to ensure you're starting from a solid foundation.

Implement contracts for non-curried functions first, before moving on to having contracts work well with currying.

Submission instructions

Submit the file `dubdub.rkt` to MarkUs. Do *not* submit the other starter code files; we'll be supplying our own version of `dubdub_errors.rkt` when testing.