CS 463 - Homework 2

Untyped Lambda Calculus

Deadline: Thursday, February 1st 2024, 5pm

- Work must be uploaded to gradescope as a pdf, and then you label which page contains each question. It should be relatively quick, but please view the submission options ahead of time in case you have any questions about how to upload your work.
- **Recommendation**: type your solution up for this homework; the ability to cut-paste-edit while simplifying terms is well worth it! For simplicity, you can use either a backslash (\) or an actual lambda (λ) to represent lambdas, and perhaps -> instead of ⟶ for the evaluation symbol, and ==> instead of ⟹.

We will work on a few different tasks: simplifying some expressions; writing code ("encoding") in a specific calculus; and exploring an extension to our calculus.

---

§ 1. Simplifying terms down to values (40pts)

For each of the following, write out an entire simplification, showing every step and naming each rule that is used. Our class examples should be useful here. Remember, this is based on the shared calculus with multiple extensions: https://cs.gmu.edu/~marks/463/calculi/ulc_rules.pdf

While you are not expected to draw the trees as part of your answer, they are quite valuable in figuring out which rules to apply; you are encouraged to draw them as necessary. The parentheses have been colored in to help with readability a bit; you do not need to preserve the coloring.

1. `( (λn . n – 3)  8)`
2. `( (λa.a+1) (2+3) )`
3. `( ( (λx. (λy. if (x=y) x 0) ) (1+4)) (2*3))`
4. `(((λf. λn. (f n)) (λx.4*x)) 7)`
5. `((λxs. cons 3 xs) (if false (cons 2 nil) nil))`

6. *(extra credit +1):*
   `((λg.(λx.g x)) (λh.h))`

§ 2. Encoding terms (35pts)

For this section, we are only using the **core lambda calculus with no extensions.** This means there are no numbers, no true/false/if terms, no lists. Only the following:

```
t ::= x | λx.t | (t t)
v ::= λx.t
```
Three evaluation rules: `E-App1, E-App2, E-App-Abs`

We saw in class the following *encodings:*

```
true  = λx. λy. x
false = λx. λy. y
not = λa. (a false) true
and = λa. λb. (a b) a
or  = λa. λb. (a a) b
```

Your task is to use those encodings, in the core lambda calculus, and again perform simplification of the expressions below. Similar to our class examples, steps can now be "==> expand def'n of <foo>". I strongly recommend not expanding any definition until you need to use it *as a function*: it will be clearer to work with, less to write out, and easier to grade. Since it's a lambda, you know it's a value, even when it's not expanded, which can be useful information. Name the steps taken each time, as above in part 1.

1. `(not false)`

2. `((or true) false)`

3. `((and true) false)`

Next, create your own definition of `nand`, which answers true when not both of its arguments are true. Remember, we have the true/false/etc. *encodings* above, which are really just lambda terms. Thinking of the other encodings as being e.g. the true vaue is convenient for getting the logic sorted out, but ultimately it's merely a function that can behave true-ish when applied to arguments in the correct way. You can create helper functions as needed (more encodings), but make sure you don't use things from the extended version we had used in part 1 of this assignment. (For instance, there is no `if` term!) Then, simplify the next uses of it.

4. `nand = …<your definition goes here>`

5. `((nand true) false)`

a	b	((nand a) b)
false	false	true
false	true	true
true	false	true
true	true	false

§ 3. Language Extensions (25pts)

For the last part of this homework, we will perform an extension to the language much like the examples in class: we will write out new terms and values (given), write new rules, and use the language (with encodings).

We are now starting with the provided full language definition that has extensions, and (in green) we are going to add functionality for an "optional" type, similar to Java's Optional class.

```
t ::= x | λx.t | (t t) | true | false | if t t t | ~t | t = t
    | <#> | t + t | t-t | t*t | t<t | t>t
    | nil | cons t t | isnil t | head t | tail t
    | fix t
    | optional t | empty | orelse t t | ispresent t | get t

v ::= λx.t | true | false | <#> | nil | cons t t | fix t | optional t | empty
```

(Creating the new rules: this will be part of your task below.)

What is an "Optional"?
This new type of values allows us to *either* have the optional target value tucked away inside the object, or else it is currently missing (like a null pointer hiding inside the overall **`Optional`** object if you're thinking about Java's implementation. (See: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Optional.html). This is the representation of "value-missing"). Rather than resorting to manually checking for these null pointers or doing a lot of exception handling, we can acknowledge that "value-missing" scenario and directly code for it, because we always have the Optional-object and can use it with support code that addresses the empty-case directly.

- When building up more terms for our lambda calculus to extend it to include optionals, we also add the terms that will be using optionals (much like addition and if-terms did for numbers and booleans).

- Much like how `true` and `false` are the two kinds of values we have for Booleans, `optional t` and `empty` are the two new shapes of values we have for this Optional extension:
    - `optional t` represents an Optional value that *has* the value present, which it stores in the only subterm.
    - `empty` represents an Optional value that *does not* have the value present (thus no subterm).

- Much like how we also extended the core calculus to include operations over the Boolean values and integers (such as `if`, `+`, etc.), we include terms that operate over our new optional values:
    - `orelse t t` represents a term whose first term is an Optional, and we would like to use the value that's maybe tucked away in there; we provide a backup "default" value in the second subterm. Simplifying it means to simplify the first subterm down to a value, and when it's an `optional t` term, we extract the value; when it was an `empty` term, we instead use the second subterm (our backup plan).
    - `ispresent t` represents a question: assuming the subterm eventually yields an optional value (either shape), answer `true` when it's a `optional t` value, and `false` when it's an `empty` value.
    - `get t` represents a term that will unsafely unpack what had better be a `optional t` term. Dynamic checks like this can fail, such as indexing into a list with an invalid index.

Finally, on to the work of this Section Three:

3.1. (16/25pts) Write out evaluation rules that complete the extension of optionals to our lambda calculus. You task is to provide ways to simplify from any of our new terms (including `orelse`, `ispresent`, and `get` functionality) down to values in the newly extended language. Remember to focus on which subterm should be evaluated (if at all), and what needs to be evaluated along the way (stuff above the line). Understanding why the rules for e.g. E-If* and various list rules behave the way they do may be a good checkpoint before writing these out. Here are some design notes on required behavior:

a. An `orelse` term should evaluate its first subterm until it becomes either an `optional`-term or an `empty`-term, and only then will it either extract the value-that-is-present or use the default (`orelse`'s second subterm).
b. A `get` term can simplify its only subterm until it becomes an `optional`-term, and then extracts the value. There is no backup plan for if a different kind of value shows up (think of those like runtime errors, akin to an array indexing error).
c. An `ispresent` term simplifies its one subterm sufficiently until it sees either a `optional`-term or an `empty`-term, and returns `true` or `false` accordingly. *It does not try to handle all the other "types" of terms, just like `get true` has no next step, neither does `ispresent 5` have any further evaluation possible.*
d. Naming your rules similar to the provided ones can help explain your intent (no specific names are required). You can give some short commentary to the side if you'd like (also not required).

Hint: My solution has eight evaluation rules: three each for orelse and ispresent, and another two for evaluating get terms.

3.2. (9/25pts) Encode the following in your newly extended language:

- `more`, a function that can be applied to two arguments; the first is a number, the second is an optional (of a number). Returns the first argument, plus the number that is optionally hidden in the optional (or, adds zero when the optional was empty).
- `isbig,` a function that accepts an optional number and returns an optional Boolean (three possible outcomes): `optional true` or `optional false` representing that there *was* a value present that is bigger than 1000 or not, or an `empty` value when there was no option number to check its size.

§4. Extra Credit (+4)

Using your Section 3 language, encode a function, `addAnyPresent`, that accepts a list of optional numbers, and adds up all the numbers that are present. (hint: you will need to write this recursively). You do not need to evaluate it! (as we saw in class, this gets a bit out of hand quickly).