Homework 8: Optimizations

In this homework, you'll implement some optimizations in your compiler. You'll also come up with benchmark programs and see how well your optimizations do on a collaboratively-developed benchmark suite.

You'll implement the following optimizations (all of which we discussed in class):

- Constant propagation
- Inlining
- Common subexpressic

In order to make inlining an
implement, you'll also write -> `program`) to make sure
all variable names are globa

Note: Task 1 (as defined be ie
tasks. It has a separate ass

Note: There are no "subtas has
exactly one task to go with .

Starter code

The starter code is the sam ILB
syntax. In particular, lambda expressions and function pointers are not supported.

You will write all your optimizations in the file `lib/optimize.ml`. **You will not need to modify any other files.**

Testing

There is no reference solution for this homework. This is because everyone's optimizations will be slightly different!

That being said, we still encourage you to write tests using the `OUnit2` framework we used for Homework 1. For a refresher on how that works, check out

Section 2 on the course website. In any case, we will *not* be grading any tests you write for this assignment *except for* the benchmarks that you explicitly submit in Task 1. It's up to you what kinds of tests you write!

Running the optimizer and compiler

You can run the compiler with specific optimization passes enabled using the `bin/compile.exe` executable, by passing the `-p` argument one or more times. For instance:

```
dune exec bin/compile.exe -- examples/ex1.lisp output -r -p propagate-constants -p uniq
```

will execute the compiler with constant propagation, globally unique names, and inlining enabled. You can al[...] n once—for instance, doing c[...] nt propagation again.

You can also pass `-o` inst[...]

1. Benchmarks

We've set up a repository for benchmarks for this homework assignm[...] how much your optimizations im[...].

Note: We will **not** be grading your optimizations based on a competition of any kind with the benchmarks submitted. Instead, we will just be grading your optimizations on the basis of whether or not they behave as prescribed (based on our own internal tests).

Task 1 (DUE 1 WEEK EARLY): In the Gradescope assignment `hw8-benchmarks` , upload *at least three* interesting benchmark programs. These must be programs that the Homework 8 starter code can actually run! For instance, since the Homework 8 language doesn't include variadic functions or let expressions that bind multiple variables, your benchmarks should not use these features.

We will periodically take some of these submissions from Gradescope and upload them to the *public* benchmarking repository, so please **DO NOT INCLUDE ANY IDENTIFYING**

INFORMATION IN YOUR BENCHMARKS (e.g. name, email, date of birth, social security number, password...).

This also means that if you're interested in testing your optimizations on benchmarks contributed by others, you should periodically do a `git pull` on the `hw8-benchmarks` repo, to get the latest benchmark suite! This is totally optional. Again, the grade for your optimizations will not have anything to do with the benchmarks submitted by your peers. We have our own set of tests that we'll use to evaluate your optimizations. But if you want access to benchmarks that your peers have created, just for your own purposes of assessing your optimizations, pulling from `hw8-benchmarks` is the way to get a whole suite!

2. Globablly unique names

Many optimizations can be bally
unique.

Task 2: Implement this pas

This pass should be run be ation,
and both of those optimiza his is
an exception to the usual p t
matter for correctness). The nsures
that this optimization is exe on
elimination.

Hint: You should implement
inlining or common subexpression elimination!

3a. Constant propagation

Constant propagation is a crucial optimization in which as much computation as possible is done at *compile time* instead of at *run time*.

We implemented a sketch of a simple version of constant propagation in class.

Task 3a: Implement constant propagation, which should support:

- Replacing the primitive operations `add1` , `sub1` , `plus` , `minus` , `eq` , and `lt` with their statically-determined result when possible;

- Replacing `let` -bound names with constant boolean or number values when possible;
- Eliminating `if` expressions where the test expression's value can be statically determined.

Optional extension (for no additional credit): You can also implement re-associating binary operations (possibly in a separate pass) to find opportunities for constant propagation. For instance, consider the expression

```
(+ 5 (+ 2 (read-num)))
```

This expression won't be modified by the constant propagation algorithm described above, but with re-association it could be optimized to

```
(+ 7 (read-num))
```

3b. Inlining

In this task, you will implem

In general, inlining function ... sider the following code:

```
(define (f x y) (+ x y

(let ((x 2))
  (let ((y 3))
    (f y x)))
```

A naive inlining implementation might result in code like this:

```
(let ((x 2))
  (let ((y 3))
    (let ((x y))
      (let ((y x))
        (+ x y)))))
```

This expression, however, is not equivalent!

This problem can be solved by adding a simultaneous binding form like the one you implemented in Homework 3. *It can also be solved by just ensuring that all variable and parameter names are globally unique.*

You should implement a heuristic for when to inline a given function. This heuristic should involve both (1) the number of static call sites and (2) the size of the function body. For example, you could multiply some measure of the size of the function body by the number of call sites and see if this exceeds some target threshold. We recommend implementing your inliner as follows:

1. Find a function to inline. This function should satisfy your heuristics and be a *leaf* function, i.e., one that doesn't contain any function calls.
2. Inline the function, and remove the function's definition.
3. Go back to Step 1. No may now be leaf functions.

This process will never inlin ive functions.

Task 3b: Implement functio describe your heuristic in a `optimizations.ml` file.

3c. Common s

In this task, you will implem

This optimization pass should find common subexpressions, add names for those subexpressions, and replace the subexpressions with variable references.

This optimization is more challenging to implement than inlining is. Our suggested approach is to:

- Optimize each definition (including the top-level program body) independently. For each definition:
  - Make a list of *all* of the subexpressions in the program that don't include calls to `(read-num)` or `(print)`
  - Find any such subexpressions that occur more than once
  - Pick a new variable name for each expression that occurs more than once
  - Replace each subexpression with this variable name