Functions
Owen Arden
UC Santa Cruz

Next, we’ll build diamondback which adds support for

- User-Defined Functions

In the process of doing so, we will learn about

- Static Checking
- Calling Conventions
- Tail Recursion

Plan

1. Defining Functions
2. Checking Functions
3. Compiling Functions
4. Compiling Tail Calls
1. Defining Functions

First, let's add functions to our language.

As always, let's look at some examples.

Example: Increment

For example, a function that increments its input:

```python
def incr(x):
    return x + 1
incr(10)
```

We have a function definition followed by a single "main" expression, which is evaluated to yield the program's result, which, in this case, is 11.

Example: Factorial

Here's a somewhat more interesting example:

```python
def fac(n):
    if n < 1:
        return 1
    else:
        return n * fac(n - 1)
fac(5)
```

This program should produce the result:

```
5
4
3
2
1
0
120
```

Example: Factorial

Suppose we modify the above to produce intermediate results:

```python
def fac(n):
    if n < 1:
        return 1
    else:
        res = fac(n - 1)
        print(res)
        return n * res
fac(5)
```

we should now get:

```
5
4
3
2
1
0
120
24
120
```

Example: Mutual Recursion

For this language, the function definitions are global: any function can call any other function. This lets us write mutually recursive functions like:

def even(n):
    if (n == 0):
        true
    else:
        odd(n - 1)

def odd(n):
    if (n == 0):
        false
    else:
        even(n - 1)

let t0 = print(even(0)),
t1 = print(even(1)),
t2 = print(even(2)),
t3 = print(even(3))
in

What should be the result of executing this program?

Example: Mutual Recursion

For this language, the function definitions are global: any function can call any other function. This lets us write mutually recursive functions like:

def even(n):
    if (n == 0):
        true
    else:
        odd(n - 1)

def odd(n):
    if (n == 0):
        false
    else:
        even(n - 1)

let t0 = print(even(0)),
t1 = print(even(1)),
t2 = print(even(2)),
t3 = print(even(3))
in

Bindings

Let's create a special type that represents places where variables are bound,

data Bind a = Bind Id

A Bind is basically just an Id decorated with an a which will let us save extra metadata like tags or source positions to help report errors

We will use Bind at two places:
1. Let bindings
2. Function parameters

It will be helpful to have a function to extract the Id corresponding to a Bind

bindId :: Bind a -> Id
bindId Bind x = x
Programs and Declarations

A program is a list of declarations and main expression.

```haskell
data Program a = Prog
  { pDecls :: [Decl a] -- ^ function declarations,
    pBody  :: (Expr a) -- ^ "main" expression }
```

Each function lives its own declaration.

```haskell
data Decl a = Decl
  { fName  :: (Bind a) -- ^ name,
    fArgs  :: [Bind a] -- ^ parameters,
    fBody  :: (Expr a) -- ^ body expression,
    fLabel :: a -- ^ metadata/tag }
```

Expressions

Finally, let's add function application (calls) to the source expressions:

```haskell
data Expr a = ... 
  | Let (Bind a) (Expr a) (Expr a) a
  | App Id (Expr a) a
```

An application or call comprises

- an `Id`, the name of the function being called,
- a list of expressions corresponding to the parameters, and
- a metadata/tag value of type `a`.

(Note: that we are now using `Bind` instead of plain `Id` at a `Let`.)

Examples Revisited

Finally, let's add function application (calls) to the source expressions:

```haskell
data Expr a = ... 
  | Let (Bind a) (Expr a) (Expr a) a
  | App Id (Expr a) a
```

An application or call comprises

- an `Id`, the name of the function being called,
- a list of expressions corresponding to the parameters, and
- a metadata/tag value of type `a`.

(Note: that we are now using `Bind` instead of plain `Id` at a `Let`.)
Examples Revisited

Let's see how the examples above are represented:

```
ghci> parseFile "tests/input/incr.diamond"
Prog {pDecls = [Decl { fName = Bind "incr" ()
, fArgs = [Bind "n" ()]
, fBody = Prim2 Plus (Id "n" ()) (Number 1 ()) ()
, fLabel = ()}
]
, pBody = App "incr" [Number 5 ()] ()
}
```

```
ghci> parseFile "tests/input/fac.diamond"
Prog { pDecls = [Decl {fName = Bind "fac" ()
, fArgs = [Bind "n" ()]
, fBody = Let (Bind "t" ()) (Prim1 Print (Id "n" ()) ())
(If (Prim2 Less (Id "n" ()) (Number 1 ()) ())
(Number 1 ())
(Prim2 Times (Id "n" ())
(App "fac" [Prim2 Minus (Id "n" ()) (Number 1 ()) ()] ())
()) ())
, fLabel = ()}
]
pBody = App "fac" [Number 5 ()] ()
}
```

2. Static Checking

Next, we will look at an increasingly important aspect of compilation, pointing out bugs in the code at compile time.

Called Static Checking because we do this without (i.e. before) compiling and running ("dynamicking") the code.

There is a huge spectrum of checks possible:

- Code Linting (JSLint, HLint)
- Static Typing
- Static Analysis
- Contract Checking
- Dependent or Refinement Typing

Increasingly, this is the most important phase of a compiler, and modern compiler engineering is built around making these checks lightning fast. For more, see this [Interview of Anders Hejlsberg](https://www.youtube.com/watch?v=Qz9Yd5ZdWME) the architect of the C# and TypeScript compilers.

Static Well-formedness Checking

Suppose you tried to compile:

```
def fac(n):
    let t = print(n) in
    if (n < 1):
        1
    else:
        n * fac(m - 1)
    fact(5) + fac(3, 4)
```

We would like compilation to fail, not silently, but with useful messages:

```
$ make tests/output/err-fac.result
```

```
Errors found:

  tests/input/err-fac.diamond:5:10-14: Unbound variable 'm'
  | n = fac(m - 1)
  tests/input/err-fac.diamond:8:1-9: Function 'fact' is not defined
  | fact(5) + fac(3, 4)
  tests/input/err-fac.diamond:8:11-10:12: Wrong arity of arguments at call of fac
  | fact(5) + fac(3, 4)
```

We make tests/output/err-fac.result
Static Well-formedness Checking

We get multiple errors:
1. The variable \( m \) is not defined,
2. The function \( \text{fact} \) is not defined,
3. The call \( \text{fac} \) has the wrong number of arguments.

Next, let's see how to update the architecture of our compiler to support these and other kinds of errors.

Types

An error message type:

```haskell
data UserError = Error
    { eMsg :: !Text
    , eSpan :: !SourceSpan
    }
  deriving (Show, Typeable)
```

We make it an exception (that can be thrown):

```haskell
instance Exception [UserError]
```

Types

We can create errors with:

```haskell
mkError :: Text -> SourceSpan -> Error
mkError msg l = Error msg l
```

We can throw errors with:

```haskell
abort :: UserError -> a
abort e = throw [e]
```
Types

We display errors with:

\[
\text{renderErrors :: [UserError] -> IO Text}
\]

which takes something like:

\[
\begin{aligned}
\text{Error}
\end{aligned}
\]

and produce a pretty message (that requires reading the source file),

\[
\text{tests/input/err-fac.diamond:6:13-14: Unbound variable 'm'}
\]

Which runs the compiler and if any \text{UserError} are thrown, \text{catch}-es and renders the result.

Transforms

Next, let's insert a \text{checker} phase into our pipeline:

In the above, we have defined the types:

\[
\begin{aligned}
type \text{BareP} = \text{Program} \times \text{SourceSpan} & \quad -- \text{sub-expressions have src position metadata} \\
type \text{AnfP} = \text{Program} \times \text{SourceSpan} & \quad -- \text{each function body in ANF} \\
type \text{AnfTagP} = \text{Program} \times (\text{SourceSpan}, \text{Tag}) & \quad -- \text{each sub-expression has unique tag}
\end{aligned}
\]
Catching Multiple Errors

To make using a language and compiler pleasant, we should return as many errors as possible in each run.

- It's rather irritating to get errors one-by-one.

We will implement this by writing the functions

```hs
wellformed :: BareProgram -> [UserError]
```

which will recursively walk over the entire program, declaration and expression and return the list of all errors.

- If this list is empty, we just return the source unchanged,
- Otherwise, we throw the list of found errors (and exit.)

Thus, our `check` function looks like this:

```hs
check :: BareProgram -> BareProgram
check p = case wellformed p of
  [] -> p
  es -> throw es
```

Well-formed Programs

The bulk of the work is done by:

```hs
wellformed :: BareProgram -> [UserError]
wellformed (Prog ds e) =
  duplicateFunErrors ds ++
  concatMap (wellformedD fEnv) ds ++
  wellformedE fEnv emptyEnv e
```

This function,
1. creates `fEnv`, a map from function names to the function-arity (number of params),
2. computes the errors for each declaration (given functions in `fEnv`),
3. concatenates the resulting lists of errors.

Traversals

Let's look at how we might find three types of errors:

1. "unbound variables"
2. "undefined functions"

(In your assignment, you will look for many more.)

The helper function `wellformedD` creates an initial variable environment `vEnv` containing the functions parameters, and uses that (and `fEnv`) to walk over the body-expressions.

```hs
wellformedD :: FunEnv -> BareDecl -> [UserError]
```

```hs
wellformedD fEnv (Decl _ _ e) =
  wellformedE fEnv vEnv e
  where
    vEnv = addsEnv xs emptyEnv
```
Traversals

The helper function `wellFormedE` starts with the input `vEnv0` (which has just) the function parameters, and `fEnv` that has the defined functions, and traverses the expression:

- At each definition `let x e1 e2`, the variable `x` is added to the environment used to check `e2`.
- At each use `Id x` we check if `x` is in `vEnv` and if not, create a suitable `UserError`.
- At each call `App f es` we check if `f` is in `fEnv` and if not, create a suitable `UserError`.

### WellFormedE

```haskell
wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
wellFormedE fEnv vEnv0 e
  where
gos vEnv es = concatMap (gos vEnv) es
  go _ (Boolean b) = [b]  
go _ (Number n l) = [n]  
go vEnv (Id x l) = unboundVarErrors vEnv x l  
go vEnv (Prim1 _ e l) = go vEnv e  
go vEnv (Prim2 _ e1 e2 _ l) = gos vEnv [e1, e2]
go vEnv (If e1 e2 e3 _ l) = gos vEnv [e1, e2, e3]
go vEnv (Let x e1 e2 _ l) = go vEnv e1 ++ go (addEnv x vEnv) e2 ++ gos vEnv es
  go vEnv (App f es _ l) = unboundFunErrors fEnv f l ++ gos vEnv es
```

You should understand the above and be able to easily add extra error checks.

Quiz

Which function(s) would we have to modify to add large number errors (i.e. errors for numeric literals that may cause overflow)?

A. `wellFormed :: BareProgram -> [UserError]`
B. `wellFormedD :: FunEnv -> BareDecl -> [UserError]`
C. `wellFormedE :: FunEnv -> Env -> Bare -> [UserError]`
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform-ind
Quiz

Which function(s) would we have to modify to add large number errors (i.e. errors for numeric literals that may cause overflow)?

A. wellFormed :: BareProgram -> [UserError]
B. wellFormedD :: FunEnv -> BareDecl -> [UserError]
C. wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform-grp

Quiz

Which function(s) would we have to modify to add variable shadowing errors?

A. wellFormed :: BareProgram -> [UserError]
B. wellFormedD :: FunEnv -> BareDecl -> [UserError]
C. wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform2-ind
Quiz

Which function(s) would we have to modify to add duplicate parameter errors?

A. wellFormed :: BareProgram -> [UserError]
B. wellFormedD :: FunEnv -> BareDecl -> [UserError]
C. wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform3-ind

Quiz

Which function(s) would we have to modify to add duplicate parameter errors?

A. wellFormed :: BareProgram -> [UserError]
B. wellFormedD :: FunEnv -> BareDecl -> [UserError]
C. wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform3-grp

Quiz

Which function(s) would we have to modify to add duplicate function errors?

A. wellFormed :: BareProgram -> [UserError]
B. wellFormedD :: FunEnv -> BareDecl -> [UserError]
C. wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform4-ind
Quiz
Which function(s) would we have to modify to add duplicate function errors?

A. wellFormed :: BareProgram -> [UserError]
B. wellFormedD :: FunEnv -> BareDecl -> [UserError]
C. wellFormedE :: FunEnv -> Env -> Bare -> [UserError]
D. 1 and 2
E. 2 and 3

http://tiny.cc/cse110a-wellform4-grp

Compiling Functions

In the above, we have defined the types:

```haskell
type BareP = Program SourceSpan  -- ^ sub-expressions have src position metadata
type AnfP = Program SourceSpan  -- ^ each function body in ABF
type AnfTagP = Program (SourceSpan, Tag)  -- ^ each sub-expression has unique tag
```

Tagging

The `Tag` phase simply recursively tags each function body and the main expression
**ANF Conversion**

- The **normalize** phase (i.e. \textit{anf}) is recursively applied to each function body.
- In addition to \textit{Prim2} operands, each call’s arguments should be transformed into an immediate expression (Why?)

Generalize the strategy for binary operators from Boa

- from (2 arguments) to \(n\)-arguments.

---

**Strategy**

Now, let’s look at compiling function definitions and calls. We need a co-ordinated strategy.

- **Definitions** — Each definition is compiled into a labeled block of \texttt{Asm} that implements the body of the definitions. (But what about the parameters?)
- **Calls** — Each call of \texttt{f(args)} will execute the block labeled \texttt{f} (But what about the parameters?)

---

**Strategy: The Stack**

We will use our old friend, the stack to

- pass parameters
- have local variables for called functions
Calling Convention

Recall that we are using the C calling convention that ensures the above stack layout.

Strategy: Definitions

When the function body starts executing, the parameters $x_1, x_2, \ldots, x_n$ are at $[ebp + 4*2], [ebp + 4*3], \ldots, [ebp + 4*(n+1)]$.

1. Ensure that enough stack space is allocated i.e. that $esp$ and $ebp$ are properly managed.
2. Compile body with initial Env mapping parameters to $-2, -3, \ldots, -(n+1)$.

Strategy: Calls

As in Cobra, we must ensure that the parameters actually live at the above address.

1. Before the call, push the parameter values onto the stack in reverse order,
2. Call the appropriate function (using its label),
3. After the call, clear the stack by incrementing $esp$ appropriately.

NOTE: At both definition and call, if you are compiling on MacOS, you need to also respect the 16-Byte Stack Alignment Invariant.
Types

We already have most of the machinery needed to compile calls.

Let's just add a new kind of `Label` for each user-defined function:

```haskell
data Label = ...
   | DefFun Id
```

We will also extend the `Arg` type to include information about size directives:

```haskell
data Arg = ...
   | Sized Size Arg
```

We will often need to specify that an `Arg` is a double word
(the other possibilities are `single` word and `byte`) which we needn't worry about.

```haskell
data Sized = DWordPtr
```

Implementation

Let's refactor our `compile` functions into:

```haskell
compileProg :: AnfTagP -> Asm
compileDecl :: AnfTagD -> Asm
compileExpr :: Env -> AnfTagE -> Asm
```

that respectively compile `Program`, `Decl` and `Expr`.

In order to simplify stack management as in Cobra let's have a helper function that compiles the body of each function:

```haskell
compileBody :: Env -> AnfTagE -> Asm
```

`compileBody env e` will wrap the `Asm` generated by `compileExpr env e` with the code that manages `esp` and `ebp`.

Compiling Programs

To compile a `Program` we compile each `Decl` and the main body expression

```haskell
compileProg (Prog ds e) = compileBody emptyEnv e ++ concatMap compileDecl ds
```

QUIZ: Does it matter whether we put the code for `e` before `ds`?
1. Yes
2. No
Compiling Programs

To compile a Program we compile each Decl and the main body expression:

\[
\text{compileProg} \; (\text{Prog} \; ds \; e) = \text{compileBody} \; \text{emptyEnv} \; e \; \mathbin{++} \; \text{concatMap} \; \text{compileDecl} \; ds
\]

QUIZ: Does it matter what order we compile the ds?
1. Yes
2. No

Compiling Declarations

To compile a single Decl we:
1. Create a block starting with a label for the function's name (so we know where to call),
2. Invoke \text{compileBody} to fill in the assembly code for the body, using the initial Env obtained from the function's formal parameters.

\[
\text{compileDecl} :: \text{ADcl} \rightarrow \text{[Instruction]}
\]

\[
\text{compileDecl} \; (\text{Decl} \; f \; xs \; e \; _) = \text{ILabel} \; (\text{DefFun} \; (\text{bindId} \; f)) : \text{compileBody} \; (\text{paramsEnv} \; xs) \; e
\]

Compiling Declarations

The initial Env is created by \text{paramsEnv} which returns an Env mapping each parameter to its stack position:

\[
\text{paramsEnv} :: [\text{Bind} \; a] \rightarrow \text{Env}
\]

\[
\text{paramsEnv} \; xs = \text{fromListEnv} \; (\text{zip} \; \text{xids} \; [-2, -3..])
\]

where
\[
\text{xids} = \text{map} \; \text{bindId} \; xs
\]

(Recall that \text{bindId} extracts the \text{Id} from each \text{Bind})
Compiling Declarations

Finally, as in cobra, `compileBody env e` wraps the assembly for `e` with the code that manages `esp` and `ebp`.

```
compileBody :: Env -> AnfTagE -> Anm
compileBody env e = entryCode e
  ++ compileExpr env e
  ++ exitCode e

entryCode :: AnfTagE -> Asm
entryCode e = [IPush (Reg EBP), IMov (Reg EBP) (Reg ESP), ISub (Reg ESP) (Const 4 * n)]
  where n = countVars e

exitCode :: AnfTagE -> Asm
exitCode = [IMov (Reg ESP) (Reg EBP), IPop (Reg EBP)]
```

Compiling Calls

Finally, let’s extend code generation to account for calls:

```
compileExpr :: Env -> AnfTagE -> [Instruction]
compileExpr env (App f vs _) = call (DefFun f) [param env v | v <- vs]

The function `param` converts an immediate expressions (corresponding to function arguments)

```
param :: Env -> ImmE -> Arg
param env v = Sized DWordPtr (immArg env v)
```

The `Sized DWordPtr` specifies that each argument will occupy a double word (i.e. 4 bytes) on the stack.

EXERCISE

The hard work compiling calls is done by:

```
call :: Label -> [Arg] -> [Instruction]
```

Fill in the implementation of `call` yourself. As an example of its behavior, consider the (source) program:

```
def add2(x, y):
  x + y
add2(12, 7)
```

The call `add2(12, 7)` is represented as:

```
App "add2" (Number 12, Number 7)
```

The code for the call is generated by

```
call (DefFun "add2") (arg 12, arg 7)
```

where `arg` converts source values into assembly `Arg` which should generate the equivalent of the assembly:

```
push DWORD 14
push DWORD 24
call label_def_add2
add esp, 8
```
Compiling Tail Calls

Our language doesn’t have loops. While recursion is more general, it is more expensive because it uses up stack space (and requires all the attendant management overhead). For example (the python program):

```python
def sumTo(n):
r = 0
i = n
while (0 <= i):
r = r + i
i = i - 1
return r
sumTo(10000)
```

- Requires a single stack frame
- Can be implemented with 2 registers

But, the “equivalent” diamond program:

```python
def sumTo(n):
  if (n == 0):
    0
  else:
    n + sumTo(n - 1)
sumTo(10000)
```

- Requires 10000 stack frames ...
- One for `fac(10000)`, one for `fac(9999)` etc.

Tail Recursion

Fortunately, we can do much better.

A tail recursive function is one where the recursive call is the last operation done by the function, i.e. where the value returned by the function is the same as the value returned by the recursive call.

We can rewrite `sumTo` using a tail-recursive `loop` function:

```python
def loop(r, i):
  if (0 <= i):
    let rr = r + i
    , ii = i - 1
  in
  loop(rr, ii) "# tail call"
else:
  r

def sumTo(n):
  loop(0, n)
sumTo(10000)
```

Visualizing Tail Calls

Plain Recursion

- Each call pushes a frame onto the call-stack;
- The results are popped off and added to the parameter at that frame.

```python
sumTo(5)
--> 5 + sumTo(4)
--- 5 + (4 + sumTo(3))
---- 5 + (4 + (3 + (2 + sumTo(1))))
----- 5 + (4 + (3 + (2 + (1 + sumTo(0)))))
------ 5 + (4 + (3 + (2 + (1 + 0))))
------- 5 + (4 + (3 + (2 + 1)))
-------- 5 + (4 + (3 + 3))
--------- 5 + (4 + 6)
---------- 5 + 10
----------- 15
```
Visualizing Tail Calls

Tail Recursion
- Accumulation happens in the parameter (not with the output),
- Each call returns its result without further computation
No need to use call-stack, can make recursive call in place. * Tail recursive calls can be compiled into loops!

Tail Recursion Strategy

Instead of using `call` to make the call, simply:

1. Move the call’s arguments to the (same) stack position (as current args),
2. Free current stack space by resetting `esp` and `ebp` (as just prior to `ret` c.f. `exitCode`),
3. Jump to the start of the function.

That is, here’s what a naïve implementation would look like:

```
push [ebp - 8]  # push ii
push [ebp - 4]  # push rr
call def_loop
```

Tail Recursion Strategy

but a tail-recursive call can instead be compiled as:

```
mov eax, [ebp - 8]  # overwrite i with ii
mov [ebp + 12], eax
mov eax, [ebp + 4]  # overwrite r with rr
mov [ebp + 8], eax
mov esp, ebp       # "free" stack frame (as before `ret`)
pop ebp
jmp def_loop        # jump to function start
```

which has the effect of executing `loop` literally as if it were a while-loop!
**Requirements**

To implement the above strategy, we need a way to:

1. Identify tail calls in the source `Expr` (AST),
2. Compile the tail calls following the above strategy.

**Types**

We can do the above in a single step, i.e., we could identify the tail calls during the code generation, but it’s cleaner to separate the steps into:

<table>
<thead>
<tr>
<th>Parse</th>
<th>Check</th>
<th>Name</th>
<th>Tag</th>
<th>Tail</th>
<th>CodeGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td><code>BareP</code></td>
<td><code>BareP</code></td>
<td><code>AnfP</code></td>
<td><code>AnfTagP</code></td>
<td><code>AnfTagTP</code></td>
</tr>
</tbody>
</table>

In the above, we have defined the types:

- `type BareP = Program SourceSpan` — “sub-expressions have src position metadata”
- `type AnfP = Program SourceSpan` — “each function body in ANF”
- `type AnfTagP = Program (SourceSpan, Tag)` — “each sub-expression has unique tag”
- `type AnfTagTP = Program ((SourceSpan, Tag), Bool)` — “each call is marked as “tail” or not

**Transforms**

Thus, to implement tail-call optimization, we need to write two transforms:

1. To Label each call with `True` (if it is a tail call) or `False` otherwise:
   
   ```haskell
   tails :: Program a -> Program (a, Bool)
   ```

2. To Compile tail calls, by extending `compileExpr`
Labeling Tail Calls

The \texttt{Expr} in non tail positions

- \texttt{Prim1}
- \texttt{Prim2}
- \texttt{Let} ("bound expression")
- \texttt{If} ("condition")

cannot contain tail calls; all those values have some further computation performed on them.

However, the \texttt{Expr} in tail positions

- \texttt{If} ("then" and "else" branch)
- \texttt{Let} ("body")

can contain tail calls (unless they appear under the first case)

Transforms

Algorithm: Traverse \texttt{Expr} using a \texttt{Bool}

- Initially \texttt{True} but
- Toggled to \texttt{False} under non-tail positions,
- Used as "tail-label" at each call.

\textbf{NOTE:} All non-calls get a default tail-label of \texttt{False}. 

Transforms

```
Transforms

```

```
Transforms

```

```
Transforms

```

```
Compiling Tail Calls

```

```
Compiling Tail Calls

```

```
EXERCISE: How could we modify tails
to only mark tail-recursive calls, i.e. to
the same function (whose declaration is
being compiled?)

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?
```

```
EXERCISE: How could we modify tails
to only mark tail-recursive calls, i.e. to
the same function (whose declaration is
being compiled?)

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?

EXERCISE: Does the above strategy work
always? Can you think of situations where
it may go horribly wrong?