Functions

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, lets add functions to our language.

As always, lets look at some examples.

Example: Increment

For example, a function that increments its input:

```
def incr(x):
  x + 1

incr(10)
```

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

Example: Factorial

Here's a somewhat more interesting example:

```
def fac(n):
  let t = print(n) in
  if (n < 1):
    1
  else:
    n * fac(n - 1)

fac(5)
```

This program should produce the result

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

Suppose we modify the above to produce intermediate results:

```
def fac(n):
  let t   = print(n)
    , res = if (n < 1):
              1
            else:
              n * fac(n - 1)
  in
    print(res)

fac(5)
```

we should now get:

```
5
4
3
2
1
0
1
1
2
6
24
120
120
```

Example: Mutually Recursive Functions

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
    0
```

QUIZ What should be the result of executing the above?

1. `false true false true 0`
2. `true false true false 0`
3. `false false false false 0`
4. `true true true true 0`

Types

Lets add some new types to represent programs.

Bindings

Lets create a special type that represents places where **variables are bound**,

```
data Bind a = Bind Id a
```

A `Bind` is an `Id` *decorated with* an `a`

- to save extra *metadata* like **tags** or **source positions**
- to make it easy to 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

A **program** is a list of declarations and *main* expression.

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

Declarations

Each **function** lives is its own **declaration**,

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

Expressions

Finally, lets add *function application* (calls) to the source expressions:

```
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

Lets see how the examples above are represented:

```
>>> 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 ()] ()
     }

>>> 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 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 the architect of the C# and TypeScript compilers.

Static Well-formedness Checking

We will look at code linting and, later in the quarter, type systems in 131.

For the former, 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:6:13-14: Unbound variable 'm'

         6|      n * fac(m - 1)
                         ^

tests/input/err-fac.diamond:8:1-9: Function 'fact' is not defined

         8|  fact(5) + fac(3, 4)
             ^^^^^^^^

tests/input/err-fac.diamond:(8:11)-(9:1): Wrong arity of arguments a
t call of fac

         8|  fact(5) + fac(3, 4)
                       ^^^^^^^^^
```

We get *multiple* errors:

1. The variable `m` is not defined,
2. The function `fact` is not defined,
3. The call `fac` has the wrong number of arguments.

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

Types: An Error Reporting API

An *error message* type:

```
data UserError = Error
  { eMsg  :: !Text          -- ^ error message
  , eSpan :: !SourceSpan    -- ^ source position
  }
  deriving (Show, Typeable)
```

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

```
instance Exception [UserError]
```

We can **create** errors with:

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

We can **throw** errors with:

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

We **display errors** with:

```
renderErrors :: [UserError] -> IO Text
```

which takes something like:

```
Error
  "Unbound variable 'm'"
  { file      = "tests/input/err-fac"
  , startLine = 8
  , startCol  = 1
  , endLine   = 8
  , endCol    = 9
  }
```

and produces a **contextual message** (that requires reading the source file),

```
tests/input/err-fac.diamond:6:13-14: Unbound variable 'm'

         6|      n * fac(m - 1)
                         ^
```

We can put it all together by

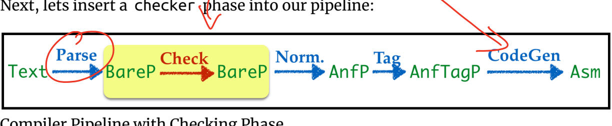
```
-- bin/Main.hs
main :: IO ()
main = runCompiler `catch` esHandle

esHandle :: [UserError] -> IO ()
esHandle es = renderErrors es >>= hPutStrLn stderr >> exitFailure
```

Which runs the compiler and if any `UserError` are thrown, `catch` -es and renders the result.

Transforms

between parse

Next, lets insert a `checker` phase into our pipeline:



Compiler Pipeline with Checking Phase

In the above, we have defined the types:

```
type BareP   = Program SourceSpan        -- ^ source position metada
ta
type AnfP    = Program SourceSpan        -- ^ sub-exprs in ANF
type AnfTagP = Program (SourceSpan, Tag) -- ^ sub-exprs have unique
tag
```

Catching Multiple Errors

Its rather irritating to get errors one-by-one.

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

We will implement this by writing the functions

```
wellFormed  :: BareProgram -> [UserError]
```

which will *recursively traverse* 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:

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

Functions
- Defined behavior
- Static Checker
- Compile Fun /Call
- Tail Recursion

Well-formed Programs, Declarations and Expressions

The bulk of the work is done by three functions

```
-- Check a whole program
wellFormed  ::                 BareProgram -> [UserError]

-- Check a single declaration
wellFormedD :: FunEnv ->        BareDecl    -> [UserError]

-- Check a single expression
wellFormedE :: FunEnv -> Env -> Bare        -> [UserError]
```

Well-formed Programs

To check the whole program

```
wellFormed :: BareProgram -> [UserError]
wellFormed (Prog ds e)
  =  concat [wellFormedD fEnv d | d <- ds]
  ++ wellFormedE fEnv emptyEnv e
  where
    fEnv = funEnv ds

funEnv :: [Decl] -> FunEnv
funEnv ds = fromListEnv [(bindId f, length xs)
                        | Decl f xs _ _ <- ds]
```

This function,

1. **Creates** `FunEnv` , 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.

QUIZ

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

1. `wellFormed :: BareProgram -> [UserError]`
2. `wellFormedD :: FunEnv -> BareDecl -> [UserError]`
3. `wellFormedE :: FunEnv -> Env -> Bare -> [UserError]`
4. `1 and 2`
5. `2 and 3`

QUIZ

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

1. `wellFormed :: BareProgram -> [UserError]`
2. `wellFormedD :: FunEnv -> BareDecl -> [UserError]`
3. `wellFormedE :: FunEnv -> Env -> Bare -> [UserError]`
4. `1 and 2`
5. `2 and 3`

def fac(n):
let n = 5 in
if ....

QUIZ

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

1. `wellFormed :: BareProgram -> [UserError]`
2. `wellFormedD :: FunEnv -> BareDecl -> [UserError]`
3. `wellFormedE :: FunEnv -> Env -> Bare -> [UserError]`
4. `1 and 2`
5. `2 and 3`

QUIZ

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

1. `wellFormed :: BareProgram -> [UserError]`
2. `wellFormedD :: FunEnv -> BareDecl -> [UserError]`
3. `wellFormedE :: FunEnv -> Env -> Bare -> [UserError]`
4. `1 and 2`
5. `2 and 3`

Traversals

Lets look at how we might check for two 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.

```
wellFormedD :: FunEnv -> BareDecl -> [UserError]
wellFormedD fEnv (Decl _ xs e _) = wellFormedE fEnv vEnv e
  where
    vEnv                          = addsEnv xs emptyEnv
```

The helper function `wellFormedE` starts with the input

- `vEnv0` which has the function parameters, and
- `fEnv` that has the defined functions,

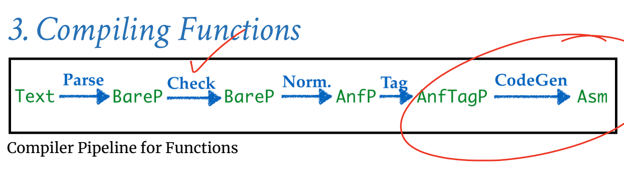
and traverses the expression:

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

3. Compiling Functions

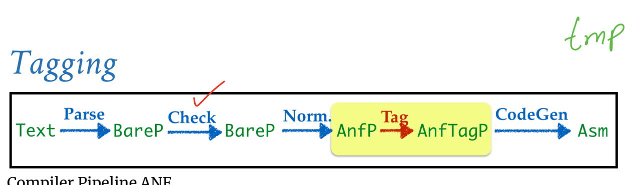


Compiler Pipeline for Functions

In the above, we have defined the types:

```
type BareP   = Program SourceSpan         -- ^ each sub-expression has source position metadata
type AnfP    = Program SourceSpan         -- ^ each function body in ANF
type AnfTagP = Program (SourceSpan, Tag) -- ^ each sub-expression has unique tag
```

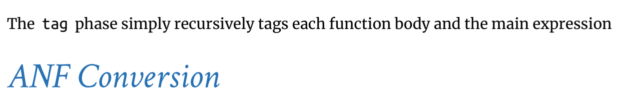
Tagging



Compiler Pipeline ANF

The `tag` phase simply recursively tags each function body and the main expression

ANF Conversion



Compiler Pipeline ANF

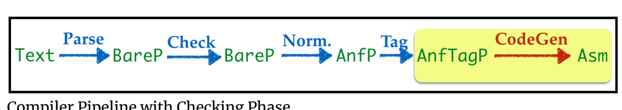
- The `normalize` phase (i.e. `anf`) is recursively applied to each function body.
- In addition to `Prim2` operands, each call's arguments should be transformed into an immediate expression

Generalize the strategy for *binary* operators

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

Strategy

Now, lets look at *compiling* function *definitions* and *calls*.



Compiler Pipeline with Checking Phase

We need a co-ordinated strategy for *definitions* and *calls*.

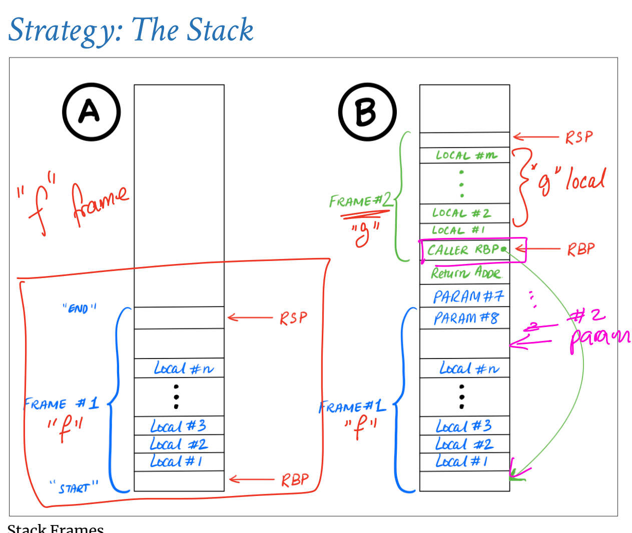
Function Definitions

- Each *definition* is compiled into a labeled block of `Asm`
- That implements the *body* of the definitions.
- (But what about the *parameters*)?

Function Calls

- Each *call* of `f(args)` will execute the block labeled `f`
- (But what about the *parameters*)?

Strategy: The Stack



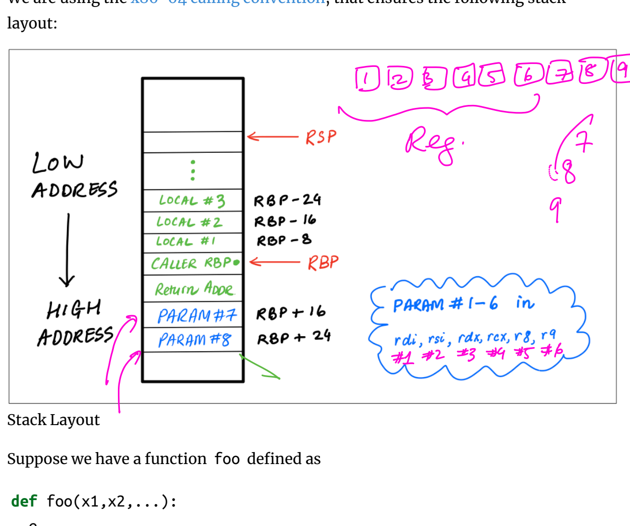
Stack Frames

We will use our old friend, the stack to

- pass *parameters*
- have *local variables* for called functions.

X86-64 Calling Convention

We are using the x86-64 calling convention, that ensures the following stack layout:



Stack Layout

Suppose we have a function `foo` defined as

```
def foo(x1,x2,...):
  e
```

When the function body **starts** executing

- the **first 6** parameters `x1` , `x2` , ... `x6` are at `rdi` , `rsi` , `rdx` , `rcx` , `r8` and `r9`
- the **remaining** `x7` , `x8` ... are at `[rbp + 8*2]` , `[rbp + 8*3]` , ...

When the function **exits**

- the **return** value is in `rax`

Pesky detail on Stack Alignment

At both *definition* and *call*, you need to also respect the 16-Byte Stack Alignment Invariant

> Ensure `rsp` is always a multiple of `16` .

i.e. pad to ensure an **even** number of arguments on stack

Strategy: Definitions

Thus to compile each definition

```
def foo(x1,x2,...):
  body
```

we must

1. **Setup Frame** to allocate space for local variables by ensuring that `rsp` and `rbp` are properly managed
2. **Copy parameters** `x1` , `x2` , ... from the registers & stack into stack-slots `1` , `2` , ... so we can access them in the `body`
3. **Compile body** `body` with initial `Env` mapping parameters `x1 => 1` , `x2 => 2` , ...
4. **Teardown Frame** to restore the caller's `rbp` and `rsp` prior to `ret` urn.

Strategy: Calls

As before we must ensure that the parameters actually live at the above address.

1. **Push** the parameter values into the registers & stack,
2. **Call** the appropriate function (using its label),
3. **Pop** the arguments off the stack by incrementing `rsp` appropriately.

Types

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

Lets just add a new kind of `Label` for each user-defined function:

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

Implementation

Lets can refactor our `compile` functions into:

```
-- Compile the whole program
compileProg :: AnfTagP -> Asm

-- Compile a single function declaration
compileDecl :: Bind -> [Bind] -> Expr -> Asm

-- Compile a single expression
compileExpr :: Env -> AnfTagE -> Asm
```

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

Compiling Programs

To compile a `Program` we compile

- the *main* expression as `Decl` with no parameters and
- each *function* declaration

```
compileProg (Prog ds e) =
     compileDecl (Bind "" ()) [] e
  ++ concat [ compileDecl f xs e | (Decl f xs e _) <- ds ]
```

QUIZ

Does it matter whether we put the code for `e` before `ds`?

1. Yes

2. No

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 `compileBody` to fill in the assembly code for the body, using the initial `Env` obtained from the function's formal parameters.

```
compileDecl :: Bind a -> [Bind a] -> AExp -> [Instruction]
compileDecl f xs body =
 -- 0. Label for start of function
    [ ILabel (DefFun (bindId f)) ]
 -- 1. Setup  stack frame RBP/RSP
 ++ funEntry n
 -- label the 'body' for tail-calls
 ++ [ ILabel (DefFunBody (bindId f)) ]
 -- 2. Copy parameters into stack slots
 ++ copyArgs xs
 -- 3. Execute 'body' with result in RAX
 ++ compileEnv initEnv body
 -- 4. Teardown stack frame & return
 ++ funExit n
  where
             -- space for params + locals
    n       = length xs + countVars body
    initEnv = paramsEnv xs
```

Setup and Tear Down Stack Frame

(As in `cobra`)

Setup frame

```
funEntry :: Int -> [Instruction]
funEntry n =
   [ IPush (Reg RBP)                       -- save caller's RBP
   , IMov  (Reg RBP) (Reg RSP)             -- set callee's RBP
   , ISub  (Reg RSP) (Const (argBytes n))  -- allocate n local-vars
   ]
```

Teardown frame

```
funExit :: Int -> [Instruction]
funExit n =
   [ IAdd (Reg RSP) (Const (argBytes n))    -- un-allocate n local-v
ars
   , IPop (Reg RBP)                         -- restore callee's RBP
   , IRet                                   -- return to caller
   ]
```

Copy Parameters into Frame

`copyArgs xs` returns the instructions needed to copy the parameter values

- **From** the combination of `rdi`, `rsi`, ...
- **To** this function's frame, `rdi -> [rbp - 8]`, `rsi -> [rbp - 16]`,...

```
copyArgs :: [a] -> Asm
copyArgs xs     = copyRegArgs   rXs -- copy upto 6 register args
               ++ copyStackArgs sXs -- copy remaining stack args
  where
    (rXs, sXs) = splitAt 6 xs

-- Copy upto 6 args from registers into offsets 1..
copyRegArgs :: [a] -> Asm
copyRegArgs xs = [ IMov (stackVar i) (Reg r) | (_,r,i) <- zipWith3
xs regs [1..] ]
  where regs   = [RDI, RSI, RDX, RCX, R8, R9]

-- Copy remaining args from stack into offsets 7..
copyStackArgs :: [a] -> Asm
copyStackArgs xs = concat [ copyArg src dst | (_,src,dst) <- zip3 xs
[-2,-3..] [7..] ]

-- Copy from RBP-offset-src to RBP-offset-dst
copyArg :: Int -> Int -> Asm
copyArg src dst =
  [ IMov (Reg RAX) (stackVar src)
  , IMov (stackVar dst) (Reg RAX)
  ]
```

Execute Function Body

(As in cobra)

`compileEnv initEnv body` generates the assembly for `e` using `initEnv`, the initial `Env` created by `paramsEnv`

```
paramsEnv :: [Bind a] -> Env
paramsEnv xs = fromListEnv (zip xids [1..])
  where
    xids     = map bindId xs
```

`paramsEnv xs` returns an `Env` mapping each parameter to its stack position

(Recall that `bindId` extracts the `Id` from each `Bind`)

Compiling Calls

Finally, lets extend code generation to account for calls:

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

EXERCISE The hard work in compiling calls is done by:

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

which implements the strategy for calls. 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 above 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:

```
  mov  rdi 24
  mov  rsi 14
  call label_def_add2
```

4. 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):

```
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

```
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:

```
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

Lets compare the execution of the two versions of `sumTo`

Plain Recursion

```
sumTo(5)
==> 5 + sumTo(4)
        ^^^^^^^^
==> 5 + [4 + sumTo(3)]
             ^^^^^^^^
==> 5 + [4 + [3 + sumTo(2)]]
                  ^^^^^^^^
==> 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
```

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

Tail Recursion

```
sumTo(5)
==> loop(0, 5)
==> loop(5, 4)
==> loop(9, 3)
==> loop(12, 2)
==> loop(14, 1)
==> loop(15, 0)
==> 15
```

- 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

Here's the code for `sumTo`

Tail Recursion Strategy

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

1. **Copy** the *call's* arguments to the (same) stack position (as current args),
   - first six in `rdi`, `rsi` etc. and rest in `[rbp+16]`, `[rbp+18]` …
2. **Jump** to the *start* of the function
   - but *after* the bit where setup the stack frame (to not do it again!)

That is, here's what a *naive* implementation would look like:

```
mov rdi, [rbp - 8]        # push rr
mov rsi, [rbp - 16]       # push ii
call def_loop
```

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

```
mov rdi, [rbp - 8]        # push rr
mov rsi, [rbp - 16]       # push ii
jmp def_loop_body
```

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

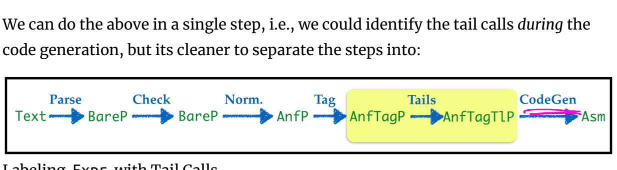
Requirements

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 its cleaner to separate the steps into:



Labeling `Expr` with Tail Calls

In the above, we have defined the types:

```
type BareP     = Program SourceSpan              -- ^ each sub-ex
pression has source position metadata
type AnfP      = Program SourceSpan              -- ^ each functi
on body in ANF
type AnfTagP   = Program (SourceSpan, Tag)       -- ^ each sub-ex
pression has unique tag
type AnfTagTlP = Program ((SourceSpan, Tag), Bool)  -- ^ each call i
s 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:

```
tails :: Program a -> Program (a, Bool)
```

2. To Compile tail calls, by extending `compileEnv`

Labeling Tail Calls

```
def facTR(acc, n):
  if (n < 1):
    acc
  else:
    if (n == 2):
      2 * facTR(n - 1, n - 1)    Not Tail
    else:
      facTR(acc * n, n - 1)      Tail
```

```
data Expr
  = Number  Integer
  | Boolean Bool
  | Id      Id
  | Prim1   Prim1  Expr
  | Prim2   Prim2  Expr  Expr
  | If      Expr   Expr  Expr
  | Let     Bind   Expr  Expr
  | App     Id     [Expr]
```

Which Calls are Tail Calls?

The `Expr` in *non tail positions*

- `Prim1`
- `Prim2`
- `Let` ("bound expression")
- `If` ("condition")

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

However, the `Expr` in *tail positions*

- `If` ("then" and "else" branch)
- `Let` ("body")

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

Algorithm: Traverse `Expr` using a `Bool`

- Initially `True` but
- Toggled to `False` under *non-tail positions*,
- Used as "tail-label" at each call.

NOTE: All non-calls get a default tail-label of `False`.

```
tails :: Expr a -> Expr (a, Bool)
tails = go True                                  -- initially
flag is True
  where
    noTail l z              = z (l, False)
    go _ (Number n l)       = noTail l (Number n)
    go _ (Boolean b l)      = noTail l (Boolean b)
    go _ (Id     x l)       = noTail l (Id x)

    go _ (Prim2 o e1 e2 l) = noTail l (Prim2 o e1' e2')
      where
        [e1', e2']          = go False <$> [e1, e2]      -- "prim-arg
s" is non-tail

    go b (If c e1 e2 l)     = noTail l (If c' e1' e2')
      where
        c'                  = go False c                 -- "cond" is
non-tail
        e1'                 = go b     e1                -- "then" ma
y be tail
        e2'                 = go b     e2                -- "else" ma
y be tail

    go b (Let x e1 e2 l)    = noTail l (Let x e1' e2')
      where
        e1'                 = go False e1                -- "bound-ex
pr" is non-tail
        e2'                 = go b     e2                -- "body-exp
r" may be tail

    go b (App f es l)       = App f es' (l, b)           -- tail-labe
l is current flag
      where
        es'                 = go False <$> es            -- "call arg
s" are non-tail
```

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

Compiling Tail Calls

Finally, to generate code, we need only add a special case to `compileExpr`

```
compileExpr :: Env -> AnfTagTlE -> [Instruction]
compileExpr env (App f vs l)
  | isTail l  = tailcall (DefFun f)     [immArg env v | v <- vs]
  | otherwise = call     (DefFunBody f) [immArg env v | v <- vs]
```

That is, *if* the call is *not labeled* as a tail call, generate code as before. Otherwise, use `tailcall` which implements our tail recursion strategy

```
tailcall :: Label -> [Arg] -> [Instruction]
tailcall l args
  = copyRegArgs       regArgs       -- copy into RDI, RSI,...
 ++ copyTailStackArgs stkArgs       -- copy into [RBP + 16], [RBP + 2
4] ...
 ++ [IJmp l]                        -- jump to start label
 where
   (regArgs, stkArgs) = splitAt 6 args
```

Recap

We just saw how to add support for **first-class function**

- **Definitions**, and
- **Calls**

and a way in which an important class of

> ***Tail Recursive** functions can be compiled as **loops**.*

Later, we'll see how to represent **functions as values** using **closures**.