Write You a Haskell

Building a modern functional compiler from first principles

Stephen Diehl

December 2015 (Draft)

Write You a Haskell
by Stephen Diehl

www.stephendiehl.com

Git commit: 23a5afe76a51032ecfd08ac8c69932364520906c

Contents

Introduction

Introduction

Goals

Off we go on our Adventure in Haskell Compilers! It will be intense, long, informative, and hopefully fun.

It's important to stress several points about the goals before we start our discussion:

a) This is not a rigorous introduction to type systems, it is a series of informal discussions of topics structured around a reference implementation with links provided to more complete and rigorous resources on the topic at hand. The goal is to give you an overview of the concepts and terminology as well as a simple reference implementation to play around with.

b) None of the reference implementations are industrial strength, many of them gloss over fundamental issues that are left out for simplicity reasons. Writing an industrial strength programming language involves work on the order of hundreds of person-years and is an enormous engineering effort.

c) You should not use the reference compiler for anything serious. It is intended for study and reference only.

Throughout our discussion we will stress the importance of semantics and the construction of core calculi. The frontend language syntax will be in the ML-family syntax out of convenience rather than principle. Choice of lexical syntax is arbitrary, uninteresting, and quite often distracts from actual substance in comparative language discussion. If there is one central theme it is that the *design of the core calculus should drive development*, not the frontend language.

Prerequisites

An intermediate understanding at the level of the *Real World Haskell* book is recommended. We will avoid advanced type-level programming that is often present in modern Haskell, and instead will make heavy use of more value-level constructs. A strong familiarity with monads, monad transformers, applicatives, and the standard Haskell data structures is strongly recommended.

Some familiarity with the standard 3rd party libraries will be useful. Many of these are briefly overviewed in What I Wish I Knew When Learning Haskell.

In particular we will use:

Library	Description
`containers`	Core data structures
`unordered-containers`	Core data structures
`text`	Text datastructure
`bytestring`	Text datastructure
`transformers`	
`mtl`	
`filepath`	
`directory`	
`process`	
`parsec`	Parser combinators
`happy`	Parser generator
`alex`	Lexer generator
`pretty`	Pretty print combinators
`ansi-wl-pprint`	Pretty print combinators
`pretty-show`	Haskell pretty printer
`graphscc`	Topological sorting
`haskeline`	GNU Readline integration
`repline`	Interactive shell builder
`cereal`	
`deepseq`	
`uniqueid`	
`uniplate`	
`optparse-applicative`	Commandline argument
`hoopl`	
`fgl`	
`llvm-general`	LLVM Codegen
`smtLib`	
`sbv`	

In later chapters some experience with C, LLVM and x86 Assembly will be very useful, although not strictly required.

Concepts

We are going to set out to build a *statically typed functional* programming language with a *native code generation backend*. What does all this mean?

Functional Languages

In mathematics a *function* is defined as a correspondence that assigns exactly one element of a set to each element in another set. If a function $f(x) = a$ then the function evaluated at x will always have

the value a. Central to the notion of all mathematics is the notion of *equational reasoning*, where if $a = f(x)$ then for an expression $g(f(x), f(x))$, this is always equivalent to $g(a, a)$. In other words, the values computed by functions can always be substituted freely at all occurrences.

The central idea of *pure functional programming* is to structure our programs in such a way that we can reason about them as a system of equations just like we can in mathematics. The evaluation of a pure function is one in which *side effects* are prohibited; a function may only return a result without altering the world in any *observable* way.

The implementation may perform effects, but central to this definition is the unobservability of such effects. A function is said to be *referentially transparent* if replacing a function with its computed value output yields the same observable behavior.

By contrast impure functions are ones which allow unrestricted and observable side effects. The invocation of an impure function always allows for the possibility of performing any functionality before yielding a value.

```
// impure: mutation side effects
function f() {
  x += 3;
  return 42;
}

// impure: international side effects
function f() {
  launchMissiles();
  return 42;
}
```

The behavior of a pure function is independent of where and when it is evaluated, whereas the behavior of an impure function is intrinsically tied to its execution order.

Static Typing

Types are a formal language integrated with a programming language that refines the space of allowable behavior and degree of expressible programs for the language. Types are the world's most popular formal method for analyzing programs.

In a language like Python all expressions have the same type at compile time, and all syntactically valid programs can be evaluated. In the case where the program is nonsensical the runtime will bubble up exceptions during evaluation. The Python interpreter makes no attempt to analyze the given program for soundness at all before running it.

```
>>> True & "false"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for &: 'bool' and 'str'
```

By comparison Haskell will do quite a bit of work to try to ensure that the program is well-defined before running it. The language that we use to prescribe and analyze static semantics of the program is that of *static types*.

```
Prelude> True && "false"

<interactive>:2:9:
    Couldn't match expected type 'Bool' with actual type '[Char]'
    In the second argument of '(&&)', namely '"false"'
    In the expression: True && "false"
    In an equation for 'it': it = True && "false"
```

Catching minor type mismatch errors is the simplest example of usage, although they occur extremely frequently as we humans are quite fallible in our reasoning about even the simplest of program constructions! Although this is just the tip of the iceberg, the gradual trend over the last 20 years goes toward more *expressive types* in modern type systems which are capable of guaranteeing a large variety of program correctness properties.

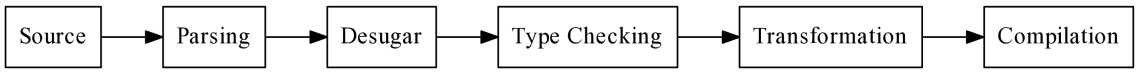
- Preventing resource allocation errors.
- Enforcing security in communication protocols.
- Side effect management.
- Preventing buffer overruns.
- Ensuring cryptographic properties for network protocols.
- Modeling and verifying theorems in mathematics and logic.
- Preventing data races and deadlocks in concurrent systems.

Even though type systems will never be able to capture all aspects of a program, more sophisticated type systems are increasingly able to model a large space of program behavior. They are one of the most exciting areas of modern computer science research. Put most bluntly, **static types let you be dumb** and offload the checking that you would otherwise have to do in your head to a system that can do the reasoning for you and work with you to interactively build your program.

Functional Compilers

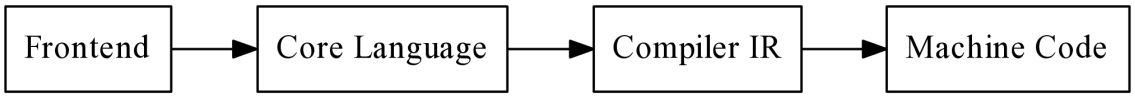
A *compiler* is a program for turning high-level representation of ideas in a human readable language into another form. A compiler is typically divided into parts, a *frontend* and a *backend*. These are loose terms but the frontend typically deals with converting the human representation of the code into some canonicalized form while the backend converts the canonicalized form into another form that is suitable for evaluation.

The high level structure of our functional compiler is described by the following *block diagram*. Each describes a *phase* which is a sequence of transformations composed to transform the input program.



Phase	
Source	The frontend textual source language.
Parsing	Source is parsed into an abstract syntax tree.
Desugar	Redundant structure from the frontend language is removed and canonicalized.
Type Checking	The program is type-checked and/or type-inferred yielding an explicitly typed form.
Transformation	The core language is transformed to prepare for compilation.
Compilation	The core language is lowered into a form to be compiled or interpreted.
(**Code Generation**)	Platform specific code is generated, linked into a binary.

A *pass* may transform the input program from one form into another or alter the internal state of the compiler context. The high level description of the forms our final compiler will go through is the following sequence:



Internal forms used during compilation are *intermediate representations* and typically any non-trivial language will involve several.

Parsing

The source code is simply the raw sequence of text that specifies the program. Lexing splits the text stream into a sequence of *tokens*. Only the presence of invalid symbols is checked; programs that are meaningless in other ways are accepted. Whitespace is either ignored or represented as a unique token in the stream.

```
let f x = x + 1
```

For instance the previous program might generate a token stream like the following:

```
[
  TokenLet,
  TokenSym "f",
  TokenSym "x",
  TokenEq,
  TokenSym "x",
  TokenAdd,
  TokenNum 1
]
```

We can then scan the token stream via dispatch on predefined patterns of tokens called *productions*, and recursively build up the syntax datatype for the *abstract syntax tree* (AST).

```
type Name = String

data Expr
  = Var Name
  | Lit Lit
  | Op PrimOp [Expr]
  | Let Name [Name] Expr

data Lit
  = LitInt Int

data PrimOp
  = Add
```

So for example the following string is parsed into the resulting `Expr` value.

```
let f x = x + 1
```

```
Let "f" ["x"] (Op Add [Var "x", Lit (LitInt 1)])
```

Desugaring

Desugaring is the process by which the frontend AST is transformed into a simpler form of itself by reducing the number of complex structures by expressing them in terms of a fixed set of simpler constructs.

Haskell's frontend is very large and many constructs are simplified down. For example `where` clauses and operator sections are the most common examples. Where clauses are effectively syntactic sugar for let bindings and operator sections are desugared into lambdas with the left or right hand side argument assigned to a fresh variable.

Type Inference

Type inference is the process by which the untyped syntax is endowed with type information by a process known as *type reconstruction* or *type inference*. The inference process may take into account explicit type annotations.

```
let f x = x + 1
```

```
Let "f" [] (Lam "x" (Op Add [Var "x", Lit (LitInt 1)]))
```

Inference will generate a system of constraints which are solved via a process known as *unification* to yield the type of the expression.

```
Int -> Int -> Int  ~  a -> b
b  ~  Int -> c

f :: Int -> Int
```

In some cases this type will be incorporated directly into the AST and the inference will transform the frontend language into an explicitly typed *core language*.

```
Let "f" []
  (Lam "x"
    (TArr TInt TInt)
    (App
      (App
        (Prim "primAdd") (Var "x"))
      (Lit (LitInt 1))))
```

Transformation

The type core representation is often suitable for evaluation, but quite often different intermediate representations are more amenable to certain optimizations and make various semantic properties of the language explicit. These kind of intermediate forms will often attach information about free variables, allocations, and usage information directly in the AST structure.

The most important form we will use is called the *Spineless Tagless G-Machine* (STG), an abstract machine that makes many of the properties of lazy evaluation explicit directly in the AST.

Code Generation

After translating an expression to the core language we will either evaluate it with a high-level interpreter written in Haskell itself, or translate it to another intermediate language (such as LLVM IR or GHC's Cmm) which can be compiled into native code. This intermediate language abstracts over the process of assigning values to, and moving values between CPU registers and main memory.

As an example, the `let` statement below would be compiled into some intermediate representation, like LLVM IR.

```
let f x = x + 1
```

```
define i32 @f(i32 %x) {
entry:
  %add = add nsw i32 %x, 1
  ret i32 %add
}
```

From the intermediate representation the code can be compiled into the system's assembly language. Any additional code that is required for evaluation is *linked* into the resulting module.

```
f:
        movl    %edi, -4(%rsp)
        movl    -4(%rsp), %edi
        addl    $1, %edi
        movl    %edi, %eax
        ret
```

And ultimately this code will be assembled into platform specific instructions by the *native code generator*, encoded as a predefined sequence of CPU instructions defined by the processor specification.

```
0000000000000000 <f>:
   0:   89 7c 24 fc             mov    %edi,-0x4(%rsp)
   4:   8b 7c 24 fc             mov    -0x4(%rsp),%edi
   8:   81 c7 01 00 00 00       add    $0x1,%edi
   e:   89 f8                   mov    %edi,%eax
  10:   c3                      retq
```

Haskell Basics

Haskell Basics

Let us now survey a few of the core concepts that will be used throughout the text. This will be a very fast and informal discussion. If you are familiar with all of these concepts then it is very likely you will be able to read the entirety of this tutorial and focus on the subject domain and not the supporting code. The domain material itself should largely be accessible to an ambitious high school student or undergraduate; and requires nothing more than a general knowledge of functional programming.

Functions

Functions are the primary building block of all of Haskell logic.

```
add :: Integer -> Integer -> Integer
add x y =  x + y
```

In Haskell all functions are pure. The only thing a function may do is return a value.

All functions in Haskell are curried. For example, when a function of three arguments receives less than three arguments, it yields a partially applied function, which, when given additional arguments, yields yet another function or the resulting value if all the arguments were supplied.

```
g :: Int -> Int -> Int -> Int
g x y z = x + y + z

h :: Int -> Int
h = g 2 3
```

Haskell supports higher-order functions, i.e., functions which take functions as arguments and yield other functions. For example the `compose` function takes two functions as arguments f and g and returns the composite function of applying f then g.

```
compose f g = \x -> f (g x)

iterate :: (a -> a) -> a -> [a]
iterate f x = x : (iterate f (f x))
```

Datatypes

Constructors for datatypes come in two flavors: *sum types* and *product types*.

A sum type consists of multiple options of *type constructors* under the same type. The two cases can be used at all locations the type is specified, and are discriminated using pattern matching.

```
data Sum = A Int | B Bool
```

A product type combines multiple fields into the same type.

```
data Prod = Prod Int Bool
```

Records are a special product type that, in addition to generating code for the constructors, generates a special set of functions known as *selectors* which extract the values of a specific field from the record.

```
data Prod = Prod { a :: Int , b :: Bool }

-- a :: Prod -> Int
-- b :: Prod -> Bool
```

Sums and products can be combined.

```
data T1
  = A Int Int
  | B Bool Bool
```

The fields of a datatype may be *parameterized*, in which case the type depends on the specific types the fields are instantiated with.

```
data Maybe a = Nothing | Just a
```

Values

A list is a homogeneous, inductively defined sum type of linked cells parameterized over the type of its values.

```
data List a = Nil | Cons a (List a)

a = [1,2,3]
a = Cons 1 (Cons 2 (Cons 3 Nil))
```

List have special value-level syntax:

```
(:) = Cons
[]  = Nil

(1 : (2 : (3 : []))) = [1,2,3]
```

A tuple is a heterogeneous product type parameterized over the types of its two values.

Tuples also have special value-level syntax.

```
data Pair a b = Pair a b

a = (1,2)
a = Pair 1 2

(,) = Pair
```

Pattern matching

Pattern matching allows us to discriminate on the constructors of a datatype, mapping separate cases to separate code paths and binding variables for each of the fields of the datatype.

```
data Maybe a = Nothing | Just a

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just a) = f a
```

Top-level pattern matches can always be written identically as case statements.

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f x = case x of
  Nothing -> n
  Just a  -> f a
```

Wildcards can be placed for patterns where the resulting value is not used.

```
const :: a -> b -> a
const x _ = x
```

Subexpression in the pattern can be explicitly bound to variables scoped on the right hand side of the pattern match.

```
f :: Maybe (Maybe a) -> Maybe a
f (Just x @ (Just _)) = x
```

List and tuples have special pattern syntax.

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)

fst :: (a, b) -> a
fst (a,b) = a
```

Patterns may be guarded by predicates (functions which yield a boolean). Guards only allow the execution of a branch if the corresponding predicate yields True.

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred []     = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      =     filter pred xs
```

Recursion

In Haskell all iteration over data structures is performed by recursion. Entering a function in Haskell does not create a new stack frame, the logic of the function is simply entered with the arguments on the stack and yields result to the register. In the case where a function returns an invocation of itself invoked in the *tail position* the resulting logic is compiled identically to `while` loops in other languages, via a `jmp` instruction instead of a `call`.

```
sum :: [Int] -> [Int]
sum ys = go ys 0
  where
    go (x:xs) i = go xs (i+x)
    go [] i = i
```

Functions can be defined to recurse mutually on each other.

```
even 0 = True
even n = odd (n-1)

odd 0 = False
odd n = even (n-1)
```

Laziness

A Haskell program can be thought of as being equivalent to a large directed graph. Each edge represents the use of a value, and each node is the source of a value. A node can be:

- A *thunk*, i.e., the application of a function to values that have not been evaluated yet
- A thunk that is currently being evaluated, which may induce the evaluation of other thunks in the process
- An expression in *weak head normal form*, which is only evaluated to the outermost constructor or lambda abstraction

The runtime has the task of determining which thunks are to be evaluated by the order in which they are connected to the main function node. This is the essence of all evaluation in Haskell and is called *graph reduction*.

Self-referential functions are allowed in Haskell. For example, the following functions generate infinite lists of values. However, they are only evaluated up to the depth that is necessary.

```
-- Infinite stream of 1's
ones = 1 : ones

-- Infinite count from n
numsFrom n = n : numsFrom (n+1)

-- Infinite stream of integer squares
squares = map (^2) (numsfrom 0)
```

The function `take` consumes an infinite stream and only evaluates the values that are needed for the computation.

```
take :: Int -> [a] -> [a]
take n _   | n <= 0 =  []
take n []           =  []
take n (x:xs)       =  x : take (n-1) xs

take 5 squares
-- [0,1,4,9,16]
```

This also admits diverging terms (called *bottoms*), which have no normal form. Under lazy evaluation, these values can be threaded around and will never diverge unless actually forced.

```
bot = bot
```

So, for instance, the following expression does not diverge since the second argument is not used in the body of `const`.

```
const 42 bot
```

The two bottom terms we will use frequently are used to write the scaffolding for incomplete programs.

```
error :: String -> a
undefined :: a
```

Higher-Kinded Types

The "type of types" in Haskell is the language of kinds. Kinds are either an arrow (`k -> k'`) or a star (`*`).

The kind of Int is `*`, while the kind of `Maybe` is `* -> *`. Haskell supports higher-kinded types, which are types that take other types and construct a new type. A type constructor in Haskell always has a kind which terminates in a `*`.

```
-- T1 :: (* -> *) -> * -> *
data T1 f a = T1 (f a)
```

The three special types `(,)`, `(->)`, `[]` have special type-level syntactic sugar:

```
(,) Int Int   =  (Int, Int)
(->) Int Int  =  Int -> Int
[] Int        =  [Int]
```

Typeclasses

A typeclass is a collection of functions which conform to a given interface. An implementation of an interface is called an instance. Typeclasses are effectively syntactic sugar for records of functions and nested records (called *dictionaries*) of functions parameterized over the instance type. These dictionaries are implicitly threaded throughout the program whenever an overloaded identifier is used. When a typeclass is used over a concrete type, the implementation is simply spliced in at the call site. When a typeclass is used over a polymorphic type, an implicit dictionary parameter is added to the function so that the implementation of the necessary functionality is passed with the polymorphic value.

Typeclasses are "open" and additional instances can always be added, but the defining feature of a typeclass is that the instance search always converges to a single type to make the process of resolving overloaded identifiers globally unambiguous.

For instance, the Functor typeclass allows us to "map" a function generically over any type of kind (`* -> *`) and apply it to its internal structure.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```haskell
instance Functor [] where
  fmap f []     = []
  fmap f (x:xs) = f x : fmap f xs

instance Functor ((,) a) where
  fmap f (a,b) = (a, f b)
```

Operators

In Haskell, infix operators are simply functions, and quite often they are used in place of alphanumerical names when the functions involved combine in common ways and are subject to algebraic laws.

```haskell
infixl 6 +
infixl 6 -
infixl 7 /
infixl 7 *

infixr 5 ++
infixr 9 .
```

Operators can be written in section form:

```haskell
(x+) =  \y -> x+y
(+y) =  \x -> x+y
(+)  =  \x y -> x+y
```

Any binary function can be written in infix form by surrounding the name in backticks.

```haskell
(+1) `fmap` [1,2,3] -- [2,3,4]
```

Monads

A monad is a typeclass with two functions: `bind` and `return`.

```haskell
class Monad m where
  bind   :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

The bind function is usually written as an infix operator.

```haskell
infixl 1 >>=

class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

This defines the structure, but the monad itself also requires three laws that all monad instances must satisfy.

Law 1

```haskell
return a >>= f = f a
```

Law 2

```haskell
m >>= return = m
```

Law 3

```haskell
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Haskell has a level of syntactic sugar for monads known as do-notation. In this form, binds are written sequentially in block form which extract the variable from the binder.

```haskell
do { a <- f ; m } = f >>= \a -> do { m }
do { f ; m } = f >> do { m }
do { m } = m
```

So, for example, the following are equivalent:

```haskell
do
  a <- f
  b <- g
  c <- h
  return (a, b, c)

f >>= \a ->
  g >>= \b ->
    h >>= \c ->
      return (a, b, c)
```

Applicatives

Applicatives allow sequencing parts of some contextual computation, but do not bind variables therein. Strictly speaking, applicatives are less expressive than monads.

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Applicatives satisfy the following laws:

```
pure id <*> v = v                            -- Identity
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w  -- Composition
```

For example:

```
example1 :: Maybe Integer
example1 = (+) <$> m1 <*> m2
  where
    m1 = Just 3
    m2 = Nothing
```

Instances of the `Applicative` typeclass also have available the functions `*>` and `<*`. These functions sequence applicative actions while discarding the value of one of the arguments. The operator `*>` discards the left argument, while `<*` discards the right. For example, in a monadic parser combinator library, the `*>` would discard the value of the first argument but return the value of the second.

Monoids

Monoids provide an interface for structures which have an associative operation (`mappend`, there is also the synonym `<>`) and a neutral (also: unit or zero) element (`mempty`) for that operation.

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

The canonical example is the list type with concatenation as the operation and the empty list as zero.

```haskell
import Data.Monoid

a :: [Integer]
a = [1,2,3] <> [4,5,6]

b :: [Integer]
b = ([1,2,3] <> mempty) <> (mempty <> [4,5,6])
```

Deriving

Instances for typeclasses like `Read`, `Show`, `Eq` and `Ord` can be derived automatically by the Haskell compiler.

```haskell
data PlatonicSolid
  = Tetrahedron
  | Cube
  | Octahedron
  | Dodecahedron
  | Icosahedron
  deriving (Show, Eq, Ord, Read)

example = show Icosahedron
example = read "Tetrahedron"
example = Cube == Octahedron
example = sort [Cube, Dodecahedron]
```

IO

A value of type `IO a` is a computation which, when performed, does some I/O before returning a value of type `a`. The notable feature of Haskell is that IO is still functionally pure; a value of type `IO a` is simply a value which stands for a computation which, when invoked, will perform IO. There is no way to peek into its contents without running it.

For instance, the following function does not print the numbers 1 to 5 to the screen. Instead, it builds a list of IO computations:

```haskell
fmap print [1..5] :: [IO ()]
```

We can then manipulate them as an ordinary list of values:

```haskell
reverse (fmap print [1..5]) :: [IO ()]
```

We can then build a composite computation of each of the IO actions in the list using `sequence_`, which will evaluate the actions from left to right. The resulting `IO` computation can be evaluated in `main` (or the GHCi repl, which effectively is embedded inside of `IO`).

```
>> sequence_ (fmap print [1..5]) :: IO ()
1
2
3
4
5

>> sequence_ (reverse (fmap print [1..5])) :: IO ()
5
4
3
2
1
```

The IO monad is wired into the runtime with compiler support. It is a special case and most monads in Haskell have nothing to do with effects in this sense.

```
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()
```

The type of `main` is always `IO ()`.

```
main :: IO ()
main = do
  putStrLn "Enter a number greater than 3: "
  x <- readLn
  print (x > 3)
```

The essence of monadic IO in Haskell is that *effects are reified as first class values in the language and reflected in the type system*. This is one of foundational ideas of Haskell, although it is not unique to Haskell.

Monad Transformers

Monads can be combined together to form composite monads. Each of the composite monads consists of *layers* of different monad functionality. For example, we can combine an error-reporting monad with a state monad to encapsulate a certain set of computations that need both functionalities. The use of monad transformers, while not always necessary, is often one of the primary ways to structure modern Haskell programs.

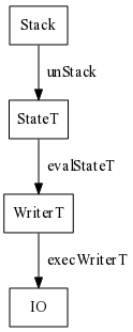
```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

The implementation of monad transformers is comprised of two different complementary libraries, `transformers` and `mtl`. The `transformers` library provides the monad transformer layers and `mtl` extends this functionality to allow implicit lifting between several layers.

To use transformers, we simply import the *Trans* variants of each of the layers we want to compose and then wrap them in a newtype.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Monad.Trans
import Control.Monad.Trans.State
import Control.Monad.Trans.Writer

newtype Stack a = Stack { unStack :: StateT Int (WriterT [Int] IO) a }
  deriving (Monad)

foo :: Stack ()
foo = Stack $ do
  put 1                   -- State layer
  lift $ tell [2]         -- Writer layer
  lift $ lift $ print 3   -- IO Layer
  return ()

evalStack :: Stack a -> IO [Int]
evalStack m = execWriterT (evalStateT (unStack m) 0)
```

As illustrated by the following stack diagram:



Using `mtl` and `GeneralizedNewtypeDeriving`, we can produce the same stack but with a simpler forward-facing interface to the transformer stack. Under the hood, `mtl` is using an extension called `FunctionalDependencies` to automatically infer which layer of a transformer stack a function belongs to and can then lift into it.

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Monad.Trans
import Control.Monad.State
import Control.Monad.Writer

newtype Stack a = Stack { unStack :: StateT Int (WriterT [Int] IO) a }
  deriving (Monad, MonadState Int, MonadWriter [Int], MonadIO)

foo :: Stack ()
foo = do
  put 1             -- State layer
  tell [2]          -- Writer layer
  liftIO $ print 3  -- IO Layer
  return ()

evalStack :: Stack a -> IO [Int]
evalStack m = execWriterT (evalStateT (unStack m) 0)
```

StateT

The state monad allows functions within a stateful monadic context to access and modify shared state.

```haskell
put    :: s -> State s ()          -- set the state value
get    :: State s s                -- get the state
gets   :: (s -> a) -> State s a    -- apply a function over the state, and return the result
modify :: (s -> s) -> State s ()   -- set the state, using a modifier function
```

Evaluation functions often follow the naming convention of using the prefixes `run`, `eval`, and `exec`:

```haskell
execState :: State s a -> s -> s         -- yield the state
evalState :: State s a -> s -> a         -- yield the return value
runState  :: State s a -> s -> (a, s)    -- yield the state and return value
```

For example:

```haskell
import Control.Monad.State

test :: State Int Int
test = do
  put 3
  modify (+1)
  get

main :: IO ()
main = print $ execState test 0
```

ReaderT

The Reader monad allows a fixed value to be passed around inside the monadic context.

```
ask   :: Reader r r                            -- get the value
asks  :: (r -> a) -> Reader r a                -- apply a function to the value, and return the r
local :: (r -> r) -> Reader r a -> Reader r a  -- run a monadic action, with the value modified b
```

For example:

```
import Control.Monad.Reader

data MyContext = MyContext
  { foo :: String
  , bar :: Int
  } deriving (Show)

computation :: Reader MyContext (Maybe String)
computation = do
  n <- asks bar
  x <- asks foo
  if n > 0
    then return (Just x)
    else return Nothing

ex1 :: Maybe String
ex1 = runReader computation $ MyContext "hello" 1

ex2 :: Maybe String
ex2 = runReader computation $ MyContext "haskell" 0
```

WriterT

The writer monad lets us emit a lazy stream of values from within a monadic context. The primary function `tell` adds a value to the writer context.

```
tell :: (Monoid w) => w -> Writer w ()
```

The monad can be evaluated returning the collected writer context and optionally the returned value.

```
execWriter :: (Monoid w) => Writer w a -> w
runWriter  :: (Monoid w) => Writer w a -> (a, w)

import Control.Monad.Writer
```

```haskell
type MyWriter = Writer [Int] String

example :: MyWriter
example = do
  tell [1..5]
  tell [5..10]
  return "foo"

output :: (String, [Int])
output = runWriter example
```

ExceptT

The Exception monad allows logic to fail at any point during computation with a user-defined exception. The exception type is the first parameter of the monad type.

```haskell
throwError :: e -> Except e a
runExcept  :: Except e a -> Either e a
```

For example:

```haskell
import Control.Monad.Except

type Err = String

safeDiv :: Int -> Int -> Except Err Int
safeDiv a 0 = throwError "Divide by zero"
safeDiv a b = return (a `div` b)

example :: Either Err Int
example = runExcept $ do
  x <- safeDiv 2 3
  y <- safeDiv 2 0
  return (x + y)
```

Kleisli Arrows

The additional combinators for monads (`(>=>)`, `(<=<)`) compose two different monadic actions in sequence. `(<=<)` is the monadic equivalent of the regular function composition operator `(.)` and `(>=>)` is just `flip (<=<)`.

```haskell
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

The monad laws can be expressed equivalently in terms of Kleisli composition.

```haskell
(f >=> g) >=> h   =   f >=> (g >=> h)
return >=> f      =   f
f >=> return      =   f
```

Text

The usual `String` type is a singly-linked list of characters, which, although simple, is not efficient in storage or locality. The letters of the string are not stored contiguously in memory and are instead allocated across the heap.

The `Text` and `ByteString` libraries provide alternative efficient structures for working with contiguous blocks of text data. `ByteString` is useful when working with the ASCII character set, while `Text` provides a text type for use with Unicode.

The `OverloadedStrings` extension allows us to overload the string type in the frontend language to use any one of the available string representations.

```
class IsString a where
  fromString :: String -> a

pack :: String -> Text
unpack :: Text -> String
```

So, for example:

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T

str :: T.Text
str = "bar"
```

Cabal & Stack

To set up an existing project with a sandbox, run:

```
$ cabal sandbox init
```

This will create the `.cabal-sandbox` directory, which is the local path GHC will use to look for dependencies when building the project.

To install dependencies from Hackage, run:

```
$ cabal install --only-dependencies
```

Finally, configure the library for building:

```
$ cabal configure
```

Now we can launch a GHCi shell scoped with the modules from the project in scope:

```
$ cabal repl
```

Resources

If any of these concepts are unfamiliar, there are some external resources that will try to explain them. The most thorough is the Stanford course lecture notes.

- Stanford CS240h by Bryan O'Sullivan, David Terei
- Real World Haskell by Bryan O'Sullivan, Don Stewart, and John Goerzen

There are some books as well, but your mileage may vary with these. Much of the material is dated and only covers basic programming and not "programming in the large".

- Introduction to Functioanl Programming by Richard Bird and Philip Wadler
- Learn you a Haskell by Miran Lipovača
- Programming in Haskell by Graham Hutton
- Thinking Functionally by Richard Bird

Parsing

Parsing

Parser Combinators

For parsing in Haskell it is quite common to use a family of libraries known as *parser combinators* which let us compose higher order functions to generate parsers. Parser combinators are a particularly expressive pattern that allows us to quickly prototype language grammars in an small embedded domain language inside of Haskell itself. Most notably we can embed custom Haskell logic inside of the parser.

NanoParsec

So now let's build our own toy parser combinator library which we'll call **NanoParsec** just to get the feel of how these things are built.

```
{-# OPTIONS_GHC -fno-warn-unused-do-bind #-}

module NanoParsec where

import Data.Char
import Control.Monad
import Control.Applicative
```

Structurally a parser is a function which takes an input stream of characters and yields a parse tree by applying the parser logic over sections of the character stream (called *lexemes*) to build up a composite data structure for the AST.

```
newtype Parser a = Parser { parse :: String -> [(a,String)] }
```

Running the function will result in traversing the stream of characters yielding a value of type `a` that usually represents the AST for the parsed expression, or failing with a parse error for malformed input, or failing by not consuming the entire stream of input. A more robust implementation would track the position information of failures for error reporting.

```
runParser :: Parser a -> String -> a
runParser m s =
```

```
  case parse m s of
    [(res, [])] -> res
    [(_, rs)]   -> error "Parser did not consume entire stream."
    _           -> error "Parser error."
```

Recall that in Haskell the String type is defined to be a list of `Char` values, so the following are equivalent forms of the same data.

```
"1+2*3"
['1', '+', '2', '*', '3']
```

We advance the parser by extracting a single character from the parser stream and returning in a tuple containing itself and the rest of the stream. The parser logic will then scrutinize the character and either transform it in some portion of the output or advance the stream and proceed.

```
item :: Parser Char
item = Parser $ \s ->
  case s of
   []     -> []
   (c:cs) -> [(c,cs)]
```

A bind operation for our parser type will take one parse operation and compose it over the result of second parse function. Since the parser operation yields a list of tuples, composing a second parser function simply maps itself over the resulting list and concat's the resulting nested list of lists into a single flat list in the usual list monad fashion. The unit operation injects a single pure value as the result, without reading from the parse stream.

```
bind :: Parser a -> (a -> Parser b) -> Parser b
bind p f = Parser $ \s -> concatMap (\(a, s') -> parse (f a) s') $ parse p s

unit :: a -> Parser a
unit a = Parser (\s -> [(a,s)])
```

As the terminology might have indicated this is indeed a Monad (also Functor and Applicative).

```
instance Functor Parser where
  fmap f (Parser cs) = Parser (\s -> [(f a, b) | (a, b) <- cs s])

instance Applicative Parser where
  pure = return
  (Parser cs1) <*> (Parser cs2) = Parser (\s -> [(f a, s2) | (f, s1) <- cs1 s, (a, s2) <- cs2 s1]
```

```
instance Monad Parser where
  return = unit
  (>>=)  = bind
```

Of particular importance is that this particular monad has a zero value (`failure`), namely the function which halts reading the stream and returns the empty stream. Together this forms a monoidal structure with a secondary operation (`combine`) which applies two parser functions over the same stream and concatenates the result. Together these give rise to both the Alternative and MonadPlus class instances which encode the logic for trying multiple parse functions over the same stream and handling failure and rollover.

The core operator introduced here is the (`<|>`) operator for combining two optional paths of parser logic, switching to the second path if the first fails with the zero value.

```
instance MonadPlus Parser where
  mzero = failure
  mplus = combine

instance Alternative Parser where
  empty = mzero
  (<|>) = option

combine :: Parser a -> Parser a -> Parser a
combine p q = Parser (\s -> parse p s ++ parse q s)

failure :: Parser a
failure = Parser (\cs -> [])

option :: Parser a -> Parser a -> Parser a
option  p q = Parser $ \s ->
  case parse p s of
    []     -> parse q s
    res    -> res
```

Derived automatically from the Alternative typeclass definition are the `many` and `some` functions. Many takes a single function argument and repeatedly applies it until the function fails and then yields the collected results up to that point. The `some` function behaves similar except that it will fail itself if there is not at least a single match.

```
-- | One or more.
some :: f a -> f [a]
some v = some_v
  where
```

```
    many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v

-- | Zero or more.
many :: f a -> f [a]
many v = many_v
  where
    many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v
```

On top of this we can add functionality for checking whether the current character in the stream matches a given predicate (i.e is it a digit, is it a letter, a specific word, etc).

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = item `bind` \c ->
  if p c
  then unit c
  else (Parser (\cs -> []))
```

Essentially this 50 lines code encodes the entire core of the parser combinator machinery. All higher order behavior can be written on top of just this logic. Now we can write down several higher level functions which operate over sections of the stream.

`chainl1` parses one or more occurrences of `p`, separated by `op` and returns a value obtained by a recursing until failure on the left hand side of the stream. This can be used to parse left-recursive grammar.

```
oneOf :: [Char] -> Parser Char
oneOf s = satisfy (flip elem s)

chainl :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op a = (p `chainl1` op) <|> return a

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = do {a <- p; rest a}
  where rest a = (do f <- op
                     b <- p
                     rest (f a b))
                 <|> return a
```

Using `satisfy` we can write down several combinators for detecting the presence of specific common patterns of characters (numbers, parenthesized expressions, whitespace, etc).

```
char :: Char -> Parser Char
char c = satisfy (c ==)
```

```haskell
natural :: Parser Integer
natural = read <$> some (satisfy isDigit)

string :: String -> Parser String
string [] = return []
string (c:cs) = do { char c; string cs; return (c:cs)}

token :: Parser a -> Parser a
token p = do { a <- p; spaces ; return a}

reserved :: String -> Parser String
reserved s = token (string s)

spaces :: Parser String
spaces = many $ oneOf " \n\r"

digit :: Parser Char
digit = satisfy isDigit

number :: Parser Int
number = do
  s <- string "-" <|> return []
  cs <- some digit
  return $ read (s ++ cs)

parens :: Parser a -> Parser a
parens m = do
  reserved "("
  n <- m
  reserved ")"
  return n
```

And that's about it! In a few hundred lines we have enough of a parser library to write down a simple parser for a calculator grammar. In the formal Backus–Naur Form our grammar would be written as:

```
number = [ "-" ] digit { digit }.
digit  = "0" | "1" | ... | "8" | "9".
expr   = term { addop term }.
term   = factor { mulop factor }.
factor = "(" expr ")" | number.
addop  = "+" | "-".
mulop  = "*".
```

The direct translation to Haskell in terms of our newly constructed parser combinator has the following form:

```haskell
data Expr
  = Add Expr Expr
  | Mul Expr Expr
  | Sub Expr Expr
  | Lit Int
  deriving Show

eval :: Expr -> Int
eval ex = case ex of
  Add a b -> eval a + eval b
  Mul a b -> eval a * eval b
  Sub a b -> eval a - eval b
  Lit n   -> n

int :: Parser Expr
int = do
  n <- number
  return (Lit n)

expr :: Parser Expr
expr = term `chainl1` addop

term :: Parser Expr
term = factor `chainl1` mulop

factor :: Parser Expr
factor =
      int
  <|> parens expr

infixOp :: String -> (a -> a -> a) -> Parser (a -> a -> a)
infixOp x f = reserved x >> return f

addop :: Parser (Expr -> Expr -> Expr)
addop = (infixOp "+" Add) <|> (infixOp "-" Sub)

mulop :: Parser (Expr -> Expr -> Expr)
mulop = infixOp "*" Mul

run :: String -> Expr
run = runParser expr

main :: IO ()
main = forever $ do
  putStr "> "
  a <- getLine
```

```
  print $ eval $ run a
```

Now we can try out our little parser.

```
$ runhaskell parsec.hs
> 1+2
3
> 1+2*3
7
```

Generalizing String

The limitations of the String type are well-known, but what is particularly nice about this approach is that it adapts to different stream types simply by adding an additional parameter to the Parser type which holds the stream type. In its place a more efficient string data structure (`Text`, `ByteString`) can be used.

```
newtype Parser s a = Parser { parse :: s -> [(a,s)] }
```

For the first couple of simple parsers we will use the String type for simplicity's sake, but later we will generalize our parsers to use the `Text` type. The combinators and parsing logic will not change, only the lexer and language definition types will change slightly to a generalized form.

Parsec

Now that we have the feel for parser combinators work, we can graduate to the full Parsec library. We'll effectively ignore the gritty details of parsing and lexing from now on. Although an interesting subject parsing is effectively a solved problem and the details are not terribly important for our purposes.

The *Parsec* library defines a set of common combinators much like the operators we defined in our toy library.

Combinator	Description
`char`	Match the given character.
`string`	Match the given string.
`<|>`	The choice operator tries to parse the first argument before proceeding to the second. Can be chained sequen
`many`	Consumes an arbitrary number of patterns matching the given pattern and returns them as a list.
`many1`	Like many but requires at least one match.
`sepBy`	Match a arbitrary length sequence of patterns, delimited by a given pattern.
`optional`	Optionally parses a given pattern returning its value as a Maybe.
`try`	Backtracking operator will let us parse ambiguous matching expressions and restart with a different pattern.
`parens`	Parses the given pattern surrounded by parentheses.

Tokens

To create a Parsec lexer we must first specify several parameters about how individual characters are handled and converted into tokens. For example some tokens will be handled as comments and simply omitted from the parse stream. Other parameters include indicating what characters are to be handled as keyword identifiers or operators.

```
langDef :: Tok.LanguageDef ()
langDef = Tok.LanguageDef
  { Tok.commentStart    = "{-"
  , Tok.commentEnd      = "-}"
  , Tok.commentLine     = "--"
  , Tok.nestedComments  = True
  , Tok.identStart      = letter
  , Tok.identLetter     = alphaNum <|> oneOf "_'"
  , Tok.opStart         = oneOf ":!#$%&*+./<=>?@\\^|-~"
  , Tok.opLetter        = oneOf ":!#$%&*+./<=>?@\\^|-~"
  , Tok.reservedNames   = reservedNames
  , Tok.reservedOpNames = reservedOps
  , Tok.caseSensitive   = True
  }
```

Lexer

Given the token definition we can create the lexer functions.

```
lexer :: Tok.TokenParser ()
lexer = Tok.makeTokenParser langDef

parens :: Parser a -> Parser a
parens = Tok.parens lexer

reserved :: String -> Parser ()
reserved = Tok.reserved lexer

semiSep :: Parser a -> Parser [a]
semiSep = Tok.semiSep lexer

reservedOp :: String -> Parser ()
reservedOp = Tok.reservedOp lexer

prefixOp :: String -> (a -> a) -> Ex.Operator String () Identity a
prefixOp s f = Ex.Prefix (reservedOp s >> return f)
```

Abstract Syntax Tree

In a separate module we'll now define the abstract syntax for our language as a datatype.

```
module Syntax where

data Expr
  = Tr
  | Fl
  | Zero
  | IsZero Expr
  | Succ Expr
  | Pred Expr
  | If Expr Expr Expr
  deriving (Eq, Show)
```

Parser

Much like before our parser is simply written in monadic blocks, each mapping a set of patterns to a construct in our `Expr` type. The toplevel entry point to our parser is the `expr` function which we can parse with by using the Parsec function `parse`.

```
prefixOp s f = Ex.Prefix (reservedOp s >> return f)


-- Prefix operators
table :: Ex.OperatorTable String () Identity Expr
table = [
    [
      prefixOp "succ" Succ
    , prefixOp "pred" Pred
    , prefixOp "iszero" IsZero
    ]
  ]


-- if/then/else
ifthen :: Parser Expr
ifthen = do
  reserved "if"
  cond <- expr
  reservedOp "then"
  tr <- expr
  reserved "else"
  fl <- expr
  return (If cond tr fl)

-- Constants
true, false, zero :: Parser Expr
true  = reserved "true"  >> return Tr
```

```
false = reserved "false" >> return Fl
zero  = reservedOp "0"   >> return Zero

expr :: Parser Expr
expr = Ex.buildExpressionParser table factor

factor :: Parser Expr
factor =
      true
  <|> false
  <|> zero
  <|> ifthen
  <|> parens expr

contents :: Parser a -> Parser a
contents p = do
  Tok.whiteSpace lexer
  r <- p
  eof
  return r
```

The toplevel function we'll expose from our Parse module is `parseExpr` which will be called as the entry point in our REPL.

```
parseExpr s = parse (contents expr) "<stdin>" s
```

Evaluation

Our small language gives rise to two syntactic classes, values and expressions. Values are in *normal form* and cannot be reduced further. They consist of `True` and `False` values and literal numbers.

```
isNum Zero     = True
isNum (Succ t) = isNum t
isNum _        = False

isVal :: Expr -> Bool
isVal Tr = True
isVal Fl = True
isVal t | isNum t = True
isVal _ = False
```

The evaluation of our languages uses the `Maybe` applicative to accommodate the fact that our reduction may halt at any level with a Nothing if the expression being reduced has reached a normal form or

cannot proceed because the reduction simply isn't well-defined. The rules for evaluation are a single step by which an expression takes a single small step from one form to another by a given rule.

```
eval' x = case x of
  IsZero Zero               -> Just Tr
  IsZero (Succ t) | isNum t -> Just Fl
  IsZero t                  -> IsZero <$> (eval' t)
  Succ t                    -> Succ <$> (eval' t)
  Pred Zero                 -> Just Zero
  Pred (Succ t) | isNum t   -> Just t
  Pred t                    -> Pred <$> (eval' t)
  If Tr  c _                -> Just c
  If Fl _ a                 -> Just a
  If t c a                  -> (\t' -> If t' c a) <$> eval' t
  _                         -> Nothing
```

At the toplevel we simply apply `eval'` repeatedly until either a value is reached or we're left with an expression that has no well-defined way to proceed. The term is "stuck" and the program is in an undefined state.

```
nf x = fromMaybe x (nf <$> eval' x)

eval :: Expr -> Maybe Expr
eval t = case nf t of
  nft | isVal nft -> Just nft
      | otherwise -> Nothing -- term is "stuck"
```

REPL

The driver for our simple language simply invokes all of the parser and evaluation logic in a loop feeding the resulting state to the next iteration. We will use the haskeline library to give us readline interactions for the small REPL. Behind the scenes haskeline is using readline or another platform-specific system library to manage the terminal input. To start out we just create the simplest loop, which only parses and evaluates expressions and prints them to the screen. We'll build on this pattern in each chapter, eventually ending up with a more full-featured REPL.

The two functions of note are the operations for the `InputT` monad transformer.

```
runInputT :: Settings IO -> InputT IO a -> IO a
getInputLine :: String -> InputT IO (Maybe String)
```

When the user enters an `EOF` or sends a `SIGQUIT` to input, `getInputLine` will yield `Nothing` and can handle the exit logic.

```haskell
process :: String -> IO ()
process line = do
  let res = parseExpr line
  case res of
    Left err -> print err
    Right ex -> print $ runEval ex

main :: IO ()
main = runInputT defaultSettings loop
  where
  loop = do
    minput <- getInputLine "Repl> "
    case minput of
      Nothing -> outputStrLn "Goodbye."
      Just input -> (liftIO $ process input) >> loop
```

Soundness

Great, now let's test our little interpreter and indeed we see that it behaves as expected.

```
Arith> succ 0
succ 0

Arith> succ (succ 0)
succ (succ 0)

Arith> iszero 0
true

Arith> if false then true else false
false

Arith> iszero (pred (succ (succ 0)))
false

Arith> pred (succ 0)
0

Arith> iszero false
Cannot evaluate

Arith> if 0 then true else false
Cannot evaluate
```

Oh no, our calculator language allows us to evaluate terms which are syntactically valid but semantically meaningless. We'd like to restrict the existence of such terms since when we start compiling our

languages later into native CPU instructions these kind errors will correspond to all sorts of nastiness (segfaults, out of bounds errors, etc). How can we make these illegal states unrepresentable to begin with?

Full Source

- NanoParsec
- Calculator

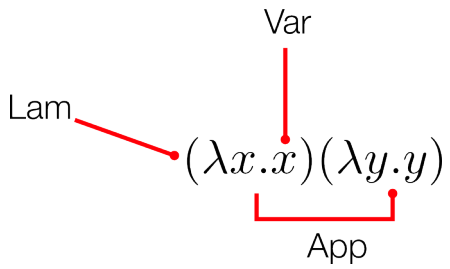
Lambda Calculus

That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted.

— George Boole

Lambda Calculus

Fundamental to all functional languages is the most atomic notion of composition, function abstraction of a single variable. The **lambda calculus** consists very simply of three terms and all valid recursive combinations thereof:



This compact notation looks slightly different from what you're used to in Haskell but it's actually not: $\lambda x.xa$ is equivalent to `\x -> x a`. This means what you see in the picture above would translate to `(\x -> x) (\y -> y)`, which would be equivalent to writing `id id` (which of course evaluates to `id`).

The three terms are typically referred to in code by several contractions of their names:

- **Var** - A variable
- **Lam** - A lambda abstraction
- **App** - An application

A lambda term is said to bind its variable. For example the lambda here binds x. In mathematics we would typically write:

$$f(x) = e$$

Using the lambda calculus notation we write:

$$f = \lambda x.e$$

In other words, $\lambda x.e$ is a function that takes a variable x and returns e.

$$\begin{aligned} e := x \quad & (\textbf{Var}) \\ \lambda x.e \quad & (\textbf{Lam}) \\ e\ e \quad & (\textbf{App}) \end{aligned}$$

The lambda calculus is often called the "assembly language" of functional programming, and variations and extensions on it form the basis of many functional compiler intermediate forms for languages like Haskell, OCaml, Standard ML, etc. The variation we will discuss first is known as **untyped lambda calculus**, by contrast later we will discuss the **typed lambda calculus** which is an extension thereof.

There are several syntactical conventions that we will adopt when writing lambda expressions. Application of multiple expressions associates to the left.

$$x_1\ x_2\ x_3\ ...x_n = (...((x_1 x_2)x_3)...x_n)$$

By convention application extends as far to the right as is syntactically meaningful. Parentheses are used to disambiguate.

In the lambda calculus, each lambda abstraction binds a single variable, and the lambda abstraction's body may be another lambda abstraction. Out of convenience we often write multiple lambda abstractions with their variables on one lambda symbol. This is merely a syntactical convention and does not change the underlying meaning.

$$\lambda xy.z = \lambda x.\lambda y.z$$

The actual implementation of the lambda calculus admits several degrees of freedom in how lambda abstractions are represented. The most notable is the choice of identifiers for the binding variables. A variable is said to be *bound* if it is contained in a lambda expression of the same variable binding. Conversely a variable is *free* if it is not bound.

A term with free variables is said to be an *open term* while one without free variables is said to be *closed* or a *combinator*.

$$\begin{aligned} e_0 &= \lambda x.x \\ e_1 &= \lambda x.(x(\lambda y.ya)x)y \end{aligned}$$

e_0 is a combinator while e_1 is not. In e_1 both occurrences of x are bound. The first y is bound, while the second is free. a is also free.

Multiple lambda abstractions may bind the same variable name. Each occurrence of a variable is then bound by the nearest enclosing binder. For example, the x variable in the following expression is bound

on the inner lambda, while y is bound on the outer lambda. This phenomenon is referred to as *name shadowing*.

$$\lambda xy.(\lambda xz.x + y)$$

SKI Combinators

There are three fundamental closed expressions called the SKI combinators.

$$\begin{aligned}\mathbf{S} &= \lambda f.(\lambda g.(\lambda x.fx(gx))) \\ \mathbf{K} &= \lambda x.\lambda y.x \\ \mathbf{I} &= \lambda x.x\end{aligned}$$

In Haskell these are written simply:

```
s f g x = f x (g x)
k x y = x
i x = x
```

Rather remarkably Moses Schönfinkel showed that all closed lambda expressions can be expressed in terms of only the **S** and **K** combinators - even the **I** combinator. For example one can easily show that **SKK** reduces to **I**.

$$\begin{aligned}&\mathbf{SKK} \\ &= ((\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)) \\ &= ((\lambda yz.(\lambda xy.x)z(yz))(\lambda xy.x)) \\ &= \lambda z.(\lambda xy.x)z((\lambda xy.x)z) \\ &= \lambda z.(\lambda y.z)((\lambda xy.x)z) \\ &= \lambda z.z \\ &= \mathbf{I}\end{aligned}$$

This fact is a useful sanity check when testing an implementation of the lambda calculus.

Implementation

The simplest implementation of the lambda calculus syntax with named binders is the following definition.

```
type Name = String

data Expr
```

```
  = Var Name
  | App Expr Expr
  | Lam Name Expr
```

There are several lexical syntax choices for lambda expressions, we will simply choose the Haskell convention which denotes lambda by the backslash (`\`) to the body with (`->`), and application by spaces. Named variables are simply alphanumeric sequences of characters.

- **Logical notation**: `const` $= \lambda xy.x$
- **Haskell notation**: `const = \x y -> x`

In addition other terms like literal numbers or booleans can be added, and these make writing expository examples a little easier. For these we will add a `Lit` constructor.

```
data Expr
  = ...
  | Lit Lit

data Lit
  = LInt Int
  | LBool Bool
```

Substitution

Evaluation of a lambda term $((\lambda x.e)a)$ proceeds by substitution of all free occurrences of the variable x in e with the argument a. A single substitution step is called a *reduction*. We write the substitution application in brackets before the expression it is to be applied over, $[x/a]e$ maps the variable x to the new replacement a over the expression e.

$$(\lambda x.e)a \to [x/a]e$$

A substitution metavariable will be written as $[s]$.

In detail, substitution is defined like this:

$$\begin{array}{lll} [x/a]x & = a & \\ [x/a]y & = y & \text{if } x \neq y \\ [x/a]ee' & = ([x/a]e)([x/a]e') & \\ [x/a]\lambda x.e & = \lambda x.e & \\ [x/a]\lambda y.e & = \lambda y.[x/a]e & \text{if } x \neq y \text{ and } y \notin \texttt{fv}(a) \end{array}$$

where $\texttt{fv}(e)$ is the set of free variables in e.

The fundamental issue with using locally named binders is the problem of *name capture*, or how to handle the case where a substitution conflicts with the names of free variables. We need the condition in the last case to avoid the naive substitution that would fundamentally alter the meaning of the following expression when y is rewritten to x.

$$[y/x](\lambda x.xy) \to \lambda x.xx$$

By convention we will always use a *capture-avoiding* substitution. Substitution will only proceed if the variable is not in the set of free variables of the expression, and if it does then a fresh variable will be created in its place.

$$(\lambda x.e)a \to [x/a]e \quad \text{if } x \notin \texttt{fv}(a)$$

There are several binding libraries and alternative implementations of the lambda calculus syntax that avoid these problems. It is a very common problem and it is very easy to implement incorrectly even for experts.

Conversion and Equivalences

Alpha equivalence

$$(\lambda x.e) \stackrel{\alpha}{=} (\lambda y.[x/y]e)$$

Alpha equivalence is the property (when using named binders) that changing the variable on the binder and throughout the body of the expression should not change the fundamental meaning of the whole expression. So for example the following are alpha-equivalent.

$$\lambda xy.xy \quad \stackrel{\alpha}{=} \quad \lambda ab.ab$$

Beta-reduction

Beta reduction is simply a single substitution step, replacing a variable bound by a lambda expression with the argument to the lambda throughout the body of the expression.

$$(\lambda x.a)y \stackrel{\beta}{\to} [x/y]a$$

Eta-reduction

$$\lambda x.ex \stackrel{\eta}{\to} e \quad \text{if} \quad x \notin \texttt{fv}(e)$$

This is justified by the fact that if we apply both sides to a term, one step of beta reduction turns the left side to the right side:

$$(\lambda x.ex)e' \xrightarrow{\beta} ee' \quad \text{if} \quad x \notin \texttt{fv}(e)$$

Eta-expansion

The opposite of eta reduction is eta-expansion, which takes a function that is not saturated and makes all variables explicitly bound in a lambda. Eta-expansion will be important when we discuss translation into STG.

Reduction

Evaluation of lambda calculus expressions proceeds by beta reduction. The variables bound in a lambda are substituted across the body of the lambda. There are several degrees of freedom in the design space about how to do this, and in which order an expression should be evaluated. For instance we could evaluate under the lambda and then substitute variables into it, or instead evaluate the arguments and then substitute and then reduce the lambda expressions. More on this will be discussed in the section on Evaluation models.

```
Untyped> (\x.x) 1
1

Untyped> (\x y . y) 1 2
2

Untyped> (\x y z. x z (y z)) (\x y . x) (\x y . x)
  => \x y z . (x z (y z))
  => \y z . ((\x y . x) z (y z))
    => \x y . x
    => \y . z
   => z
 => \z . z
\z . z
```

Note that the last evaluation was **SKK** which we encountered earlier.

In the untyped lambda calculus we can freely represent infinitely diverging expressions:

```
Untyped> \f . (f (\x . (f x x)) (\x . (f x x)))
\f . (f (\x . (f x x)) (\x . (f x x)))

Untyped> (\f . (\x. (f x x)) (\x. (f x x))) (\f x . f f)
...

Untyped> (\x. x x) (\x. x x)
...
```

Let

In addition to application, a construct known as a **let binding** is often added to the lambda calculus syntax. In the untyped lambda calculus, let bindings are semantically equivalent to applied lambda expressions.

$$\texttt{let}\ a = e\ \texttt{in}\ b \quad := \quad (\lambda a.b)e$$

In our languages we will write let statements like they appear in Haskell.

```
let a = e in b
```

Toplevel expressions will be written as `let` statements without a body to indicate that they are added to the global scope. The Haskell language does not use this convention but OCaml, StandardML use this convention. In Haskell the preceding let is simply omitted for toplevel declarations.

```
let S f g x = f x (g x);
let K x y = x;
let I x = x;

let skk = S K K;
```

For now the evaluation rule for let is identical to that of an applied lambda.

$$\begin{array}{rcll} (\lambda x.e)v & \rightarrow & [x/v]e & (\textbf{E-Lam}) \\ \texttt{let}\ x = v\ \texttt{in}\ e & \rightarrow & [x/v]e & (\textbf{E-Let}) \end{array}$$

In later variations of the lambda calculus let expressions will have different semantics and will differ from applied lambda expressions. More on this will be discussed in the section on Hindley-Milner inference.

Everything Can Be a λ term

- 0
- 1
- 2
- succ
- pred
- not
- and
- or
- add
- mul

Recursion

Probably the most famous combinator is Curry's Y combinator. Within an untyped lambda calculus, Y can be used to allow an expression to contain a reference to itself and reduce on itself permitting recursion and looping logic.

The **Y** combinator is one of many so called fixed point combinators.

$$\mathbf{Y} = \lambda R.(\lambda x.(R(xx))\lambda x.(R(xx)))$$

Y is quite special in that given **R** It returns the fixed point of **R**.

$$\begin{aligned}\mathbf{YR} &= \lambda f.(\lambda x.(f(xx))\lambda x.(f(xx)))R \\ &= (\lambda x.(\mathbf{R}(xx))\lambda x.(\mathbf{R}(xx))) \\ &= \mathbf{R}(\lambda x.(\mathbf{R}(xx))\lambda x.(\mathbf{R}(xx))) \\ &= \mathbf{R}\ \mathbf{Y}\ \mathbf{R}\end{aligned}$$

For example the factorial function can be defined recursively in terms of repeated applications of itself to fixpoint until the base case of $0!$.

$$n! = n(n-1)!$$

$$\begin{aligned}&\mathbf{fac}\ 0 = 1 \\ &\mathbf{fac}\ n = \mathbf{R}(\mathbf{fac}) = \mathbf{R}(\mathbf{R}(\mathbf{fac}))...\end{aligned}$$

For fun one can prove that the Y-combinator can be expressed in terms of the S and K combinators.

$$\mathbf{Y} = \mathbf{SSK(S(K(SS(S(SSK))))K)}$$

In an untyped lambda calculus language without explicit fixpoint or recursive let bindings, the Y combinator can be used to create both of these constructs out of nothing but lambda expressions. However it is more common to just add either an atomic fixpoint operator or a recursive let as a fundamental construct in the term syntax.

$$\begin{aligned}e := &\ x \\ &\ e_1\ e_2 \\ &\ \lambda x.e \\ &\ \texttt{fix}\ e\end{aligned}$$

Where `fix` has the evaluation rule:

$$\texttt{fix}\ v \quad \rightarrow \quad v\ (\texttt{fix}\ v)$$

Together with the fixpoint (or the Y combinator) we can create let bindings which contain a reference to itself within the body of the bound expression. We'll call these *recursive let bindings*, they are written as `let rec` in ML dialects. For now we will implement recursive lets as simply syntactic sugar for wrapping a fixpoint around a lambda binding by the following equivalence.

```
let rec x = e1 in e2    =    let x = fix (\x. e1) in e2
```

So for example we can now write down every functional programmer's favorite two functions: `factorial` and `fibonacci`. To show both styles, one is written with `let rec` and the other with explicit `fix`.

```
let fact = fix (\fact -> \n ->
  if (n == 0)
    then 1
    else (n * (fact (n-1))));

let rec fib n =
  if (n == 0)
  then 0
  else if (n==1)
  then 1
  else ((fib (n-1)) + (fib (n-2)));
```

Omega Combinator

An important degenerate case that we'll test is the omega combinator which applies a single argument to itself.

$$\omega = \lambda x.xx$$

When we apply the ω combinator to itself we find that this results in an infinitely long repeating chain of reductions. A sequence of reductions that has no normal form (i.e. it reduces indefinitely) is said to *diverge*.

$$(\lambda x.xx)(\lambda x.xx) \rightarrow \quad (\lambda x.xx)(\lambda x.xx) \rightarrow \quad (\lambda x.xx)(\lambda x.xx) \ldots$$

We'll call this expression the Ω combinator. It is the canonical looping term in the lambda calculus. Quite a few of our type systems which are statically typed will reject this term from being well-formed, so it is quite a useful tool for testing.

$$\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx)$$

Pretty Printing

Hackage provides quite a few pretty printer libraries that ease the process of dumping out textual forms for our data types. Although there are some differences between the libraries most of them use the same set of combinators. We will use the `Text.PrettyPrint` module from the pretty package on Hackage. Most of our pretty printing will be unavoidable boilerplate but will make debugging internal state much easier.

	Combinators
`<>`	Concatenation
`<+>`	Spaced concatenation
`char`	Renders a character as a `Doc`
`text`	Renders a string as a `Doc`
`hsep`	Horizontally concatenates a list of `Doc`
`vcat`	Vertically joins a list of `Doc` with newlines

The core type of the pretty printer is the `Doc` type which is the abstract type of documents. Combinators over this type will manipulate the internal structure of this document which is then finally reified to a physical string using the `render` function. Since we intend to pretty print across multiple types we will create a `Pretty` typeclass.

```
module Pretty where

import Text.PrettyPrint

class Pretty p where
  ppr :: Int -> p -> Doc

  pp :: p -> Doc
  pp = ppr 0
```

First, we create two helper functions that collapse our lambda bindings so we can print them out as single lambda expressions.

```
viewVars :: Expr -> [Name]
viewVars (Lam n a) = n : viewVars a
viewVars _ = []

viewBody :: Expr -> Expr
viewBody (Lam _ a) = viewBody a
viewBody x = x
```

Then we create a helper function for parenthesizing subexpressions.

```
parensIf ::  Bool -> Doc -> Doc
parensIf True = parens
parensIf False = id
```

Finally, we define `ppr`. The `p` variable will indicate our depth within the current structure we're printing and allow us to print out differently to disambiguate it from its surroundings if necessary.

```
instance Pretty Expr where
  ppr p e = case e of
    Lit (LInt a)  -> text (show a)
    Lit (LBool b) -> text (show b)
    Var x   -> text x
    App a b -> parensIf (p>0) $ (ppr (p+1) a) <+> (ppr p b)
    Lam x a -> parensIf (p>0) $
         char '\\'
      <> hsep (fmap pp (viewVars e))
      <+> "->"
      <+> ppr (p+1) (viewBody e)

ppexpr :: Expr -> String
ppexpr = render . ppr 0
```

Full Source

- Untyped Lambda Calculus

Type Systems

[A type system is a] tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

— Benjamin Pierce

Type Systems

Type systems are a formal language in which we can describe and restrict the semantics of a programming language. The study of the subject is a rich and open area of research with many degrees of freedom in the design space.

As stated in the introduction, this is a very large topic and we are only going to cover enough of it to get through writing the type checker for our language, not the subject in its full generality. The classic text that everyone reads is *Types and Programming Languages* or (TAPL) and discusses the topic more in depth. In fact we will follow TAPL very closely with a bit of a Haskell flavor.

Rules

In the study of programming language semantics, logical statements are written in a specific logical notation. A property, for our purposes, will be a fact about the type of a term. It is written with the following notation:

$$1 : \texttt{Nat}$$

These facts exist within a preset universe of discourse called a *type system* with definitions, properties, conventions, and rules of logical deduction about types and terms. Within a given system, we will have several properties about these terms. For example:

- (**A1**) 0 is a natural number.
- (**A2**) For a natural number n, $\texttt{succ}(n)$ is a natural number.

Given several properties about natural numbers, we'll use a notation that will allow us to chain them together to form proofs about arbitrary terms in our system.

$$\frac{}{0 : \texttt{Nat}} \quad (\textbf{A1})$$

$$\frac{n : \texttt{Nat}}{\texttt{succ}(n) : \texttt{Nat}} \quad (\textbf{A2})$$

In this notation, the expression above the line is called the *antecedent*, and expression below the line is called the *conclusion*. A rule with no antecedent is an *axiom*.

The variable n is *metavariable* standing for any natural number, an instance of a rule is a substitution of values for these metavariables. A *derivation* is a tree of rules of finite depth. We write $\vdash C$ to indicate that there exists a derivation whose conclusion is C, that C is provable.

For example $\vdash 2 : \texttt{Nat}$ by the derivation:

$$\dfrac{\dfrac{\dfrac{}{0 : \texttt{Nat}}(\textbf{A1})}{\texttt{succ}(0) : \texttt{Nat}}(\textbf{A2})}{\texttt{succ}(\texttt{succ}(0)) : \texttt{Nat}}(\textbf{A2})$$

Also present in these derivations may be a *typing context* or *typing environment* written as Γ. The context is a sequence of named variables mapped to properties about the named variable. The comma operator for the context extends Γ by adding a new property on the right of the existing set. The empty context is denoted $\varnothing$ and is the terminal element in this chain of properties that carries no information. So contexts are defined by:

$$\begin{aligned}\Gamma ::= {} & \varnothing \\ & \Gamma,\ x : \tau\end{aligned}$$

Here is an example for a typing rule for addition using contexts:

$$\dfrac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 + e_2 : \texttt{Nat}}$$

In the case where the property is always implied regardless of the context we will shorten the expression. This is just a lexical convention.

$$\varnothing \vdash P \qquad := \qquad \vdash P$$

Type Safety

In the context of modeling the semantics of programming languages using this logical notation, we often refer to two fundamental categories of rules of the semantics.

- **Statics** : Semantic descriptions which are derived from the syntax of the language.
- **Dynamics** : Semantics descriptions which describe the value evolution resulting from a program.

Type safety is defined to be the equivalence between the statics and the dynamics of the language. This equivalence is modeled by two properties that relate the types and evaluation semantics:

- **Progress** : If an expression is well typed then either it is a value, or it can be further evaluated by an available evaluation rule.
- **Preservation** : If an expression e has type τ, and is evaluated to e', then e' has type τ.

Types

The word "type" is quite often overload in the common programming lexicon. Other languages often refer to runtime tags present in the dynamics of the languages as "types". Some examples:

```
# Python
>>> type(1)
<type 'int'>

# Javascript
> typeof(1)
'number'

# Ruby
irb(main):001:0> 1.class
=> Fixnum

# Julia
julia> typeof(1)
Int64

# Clojure
user=> (type 1)
java.lang.Long
```

While this is a perfectly acceptable alternative definition, we are not going to go that route and instead restrict ourselves purely to the discussion of *static types*, in other words types which are known before runtime. Under this set of definitions many so-called dynamically typed languages often only have a single static type. For instance in Python all static types are subsumed by the `PyObject` and it is only at runtime that the tag `PyTypeObject *ob_type` is discriminated on to give rise to the Python notion of "types". Again, this is not the kind of type we will discuss. The trade-offs that these languages make is that they often have trivial static semantics while the dynamics for the language are often exceedingly complicated. Languages like Haskell and OCaml are the opposite point in this design space.

Types will usually be written as τ and can consist of many different constructions to the point where the type language may become as rich as the value level language. For now let's only consider three simple types, two *ground types* (`Nat` and `Bool`) and an *arrow type*.

$$\begin{aligned} \tau ::= & \texttt{Bool} \\ & \texttt{Nat} \\ & \tau \to \tau \end{aligned}$$

The arrow type will be the type of function expressions, the left argument being the input type and the output type on the right. The arrow type will by convention associate to the right.

$$\tau_1 \to \tau_2 \to \tau_3 \to \tau_4 \quad = \quad \tau_1 \to (\tau_2 \to (\tau_3 \to \tau_4))$$

In all the languages which we will implement the types present during compilation are *erased*. Although types are possibly present in the evaluation semantics, the runtime cannot dispatch on types of values at runtime. Types by definition only exist at compile-time in the static semantics of the language.

Small-Step Semantics

The real quantity we're interested in formally describing is expressions in programming languages. A programming language semantics is described by the *operational semantics* of the language. The operational semantics can be thought of as a description of an abstract machine which operates over the abstract terms of the programming language in the same way that a virtual machine might operate over instructions.

We use a framework called *small-step semantics* where a derivation shows how individual rewrites compose to produce a term, which we can evaluate to a value through a sequence of state changes. This is a framework for modeling aspects of the runtime behavior of the program before running it by describing the space of possible transitions type and terms may take. Ultimately we'd like the term to transition and terminate to a *value* in our language instead of becoming "stuck" as we encountered before.

Recall our little calculator language from before when we constructed our first parser:

```
data Expr
  = Tr
  | Fl
  | IsZero Expr
  | Succ Expr
  | Pred Expr
  | If Expr Expr Expr
  | Zero
```

The expression syntax is as follows:

$$
\begin{aligned}
e ::= \ & \texttt{True} \\
& \texttt{False} \\
& \texttt{iszero}\ e \\
& \texttt{succ}\ e \\
& \texttt{pred}\ e \\
& \text{if } e \text{ then } e \text{ else } e \\
& 0
\end{aligned}
$$

The small step evaluation semantics for this little language is uniquely defined by the following 9 rules. They describe each step that an expression may take during evaluation which may or may not terminate and converge on a value.

$$\frac{e_1 \rightarrow e_2}{\texttt{succ}\ e_1 \rightarrow \texttt{succ}\ e_2} \quad \textbf{(E-Succ)}$$

$$\frac{e_1 \rightarrow e_2}{\texttt{pred}\ e_1 \rightarrow \texttt{pred}\ e_2} \quad \textbf{(E-Pred)}$$

$$\texttt{pred}\ 0 \rightarrow 0 \quad \textbf{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ n) \rightarrow n \quad \textbf{(E-PredSucc)}$$

$$\frac{e_1 \rightarrow e_2}{\texttt{iszero}\ e_1 \rightarrow \texttt{iszero}\ e_2} \quad \textbf{(E-IsZero)}$$

$$\texttt{iszero}\ 0 \rightarrow \texttt{true} \quad \textbf{(E-IsZeroZero)}$$

$$\texttt{iszero}\ (\texttt{succ}\ n) \rightarrow \texttt{false} \quad \textbf{(E-IsZeroSucc)}$$

$$\text{if}\ \texttt{True}\ \text{then}\ e_2\ \text{else}\ e_3 \rightarrow e_2 \quad \textbf{(E-IfTrue)}$$

$$\text{if}\ \texttt{False}\ \text{then}\ e_2\ \text{else}\ e_3 \rightarrow e_3 \quad \textbf{(E-IfFalse)}$$

The evaluation logic for our interpreter simply reduced an expression by the predefined evaluation rules until either it reached a normal form (a value) or got stuck.

```
nf :: Expr -> Expr
nf t = fromMaybe t (nf <$> eval1 t)

eval :: Expr -> Maybe Expr
eval t = case isVal (nf t) of
  True  -> Just (nf t)
  False -> Nothing -- term is "stuck"
```

Values in our language are defined to be literal numbers or booleans.

```
isVal :: Expr -> Bool
isVal Tr = True
isVal Fl = True
isVal t | isNum t = True
isVal _ = False
```

Written in applicative form there is a noticeable correspondence between each of the evaluation rules and our evaluation logic.

```
-- Evaluate a single step.
eval1 :: Expr -> Maybe Expr
```

```
eval1 expr = case expr of
  Succ t                    -> Succ <$> (eval1 t)
  Pred Zero                 -> Just Zero
  Pred (Succ t) | isNum t   -> Just t
  Pred t                    -> Pred <$> (eval1 t)
  IsZero Zero               -> Just Tr
  IsZero (Succ t) | isNum t -> Just Fl
  IsZero t                  -> IsZero <$> (eval1 t)
  If Tr  c _                -> Just c
  If Fl _ a                 -> Just a
  If t c a                  -> (\t' -> If t' c a) <$> eval1 t
  _                         -> Nothing
```

As we noticed before we could construct all sorts of pathological expressions that would become stuck. Looking at the evaluation rules, each of the guarded pattern matches gives us a hint of where things might "go wrong" whenever a boolean is used in the place of a number and vice versa. We'd like to statically enforce this invariant at compile-time instead, and so we'll introduce a small type system to handle the two syntactic categories of terms that exist. In addition to the arrow type, we add the abstract type of natural numbers and the type of booleans:

$$\begin{aligned} \tau ::= & \texttt{Bool} \\ & \texttt{Nat} \\ & \tau \to \tau \end{aligned}$$

Which is implemented in Haskell as the following datatype:

```
data Type
  = TBool
  | TNat
  | TArr Type Type
```

Now for the typing rules:

$$\frac{e_1 : \texttt{Nat}}{\texttt{succ}\ e_1 : \texttt{Nat}} \quad \textbf{(T-Succ)}$$

$$\frac{e_1 : \texttt{Nat}}{\texttt{pred}\ e_1 : \texttt{Nat}} \quad \textbf{(T-Pred)}$$

$$\frac{e_1 : \texttt{Nat}}{\texttt{iszero}\ e_1 : \texttt{Bool}} \quad \textbf{(T-IsZero)}$$

$$0 : \texttt{Nat} \quad \textbf{(T-Zero)}$$

$$\texttt{True} : \texttt{Bool} \quad \textbf{(T-True)}$$

$$\texttt{False} : \texttt{Bool} \quad \textbf{(T-False)}$$

$$\frac{e_1 : \texttt{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad \textbf{(T-If)}$$

These rules restrict the space of all possible programs. It is more involved to show, but this system has both progress and preservation as well. If a term is now well-typed it will always evaluate to a value and cannot "go wrong" at evaluation.

To check the well-formedness of an expression we implement a piece of logic known as *type checker* which determines whether the term has a well-defined type in terms of typing rules, and if so returns it or fails with an exception in the case where it does not.

```
type Check a = Except TypeError a

data TypeError
  = TypeMismatch Type Type

check :: Expr -> Either TypeError Type
check = runExcept . typeof

typeof :: Expr -> Check Type
typeof expr = case expr of
  Succ a -> do
    ta <- typeof a
    case ta of
      TNat -> return TNat
      _    -> throwError $ TypeMismatch ta TNat

  Pred a -> do
    ta <- typeof a
    case ta of
```

```
      TNat -> return TNat
      _    -> throwError $ TypeMismatch ta TNat

  IsZero a -> do
    ta <- typeof a
    case ta of
      TNat -> return TBool
      _    -> throwError $ TypeMismatch ta TNat

  If a b c -> do
    ta <- typeof a
    tb <- typeof b
    tc <- typeof c
    if ta /= TBool
    then throwError $ TypeMismatch ta TBool
    else
      if tb /= tc
      then throwError $ TypeMismatch ta tb
      else return tc

  Tr   -> return TBool
  Fl   -> return TBool
  Zero -> return TNat
```

Observations

The pathological stuck terms that we encountered previously in our untyped language are now completely inexpressive and are rejected at compile-time.

```
Arith> succ 0
succ 0 : Nat

Arith> succ (succ 0)
succ (succ 0) : Nat

Arith> if false then true else false
false : Bool

Arith> iszero (pred (succ (succ 0)))
false : Bool

Arith> pred (succ 0)
0 : Nat

Arith> iszero false
```

```
Type Mismatch: Bool is not Nat

Arith> if 0 then true else false
Type Mismatch: Nat is not Bool
```

This is good, we've made a whole class of illegal programs unrepresentable. Lets do more of this!

Simply Typed Lambda Calculus

The *simply typed lambda calculus* (STLC) of Church and Curry is an extension of the lambda calculus that annotates each lambda binder with a type term. The STLC is *explictly typed*, all types are present directly on the binders and to determine the type of any variable in scope we only need to traverse to its enclosing scope.

$$
\begin{aligned}
e := & \ x \\
& \ e_1\ e_2 \\
& \ \lambda x : \tau . e
\end{aligned}
$$

The simplest STLC language is these three terms, however we will add numeric and boolean literal terms so that we can write meaningful examples.

$$
\begin{aligned}
e := & \ x \\
& \ e_1\ e_2 \\
& \ \lambda x : \tau . e \\
& \ n \\
& \ \texttt{true} \\
& \ \texttt{false} \\
& \ \text{if } e \text{ then } e \text{ else } e
\end{aligned}
$$

We can consider a very simple type system for our language that will consist of `Int` and `Bool` types and function types.

$$
\begin{aligned}
\tau := & \ \texttt{Int} \\
& \ \texttt{Bool} \\
& \ \tau \to \tau
\end{aligned}
$$

Type Checker

The typing rules are quite simple, and again we get the nice property that there is a one-to-one mapping between each syntax term and a typing rule.

- **T-Var** Variables are simply pulled from the context.
- **T-Lam** lambdas introduce a typed variable into the environment when inferring the body.
- **T-App** Applications of a lambda with type `t1 -> t2` to a value of type `t1` yields a value of type `t2`.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \textbf{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \quad \textbf{(T-Lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad \textbf{(T-App)}$$

$$\frac{\Gamma \vdash c : \texttt{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } c \text{ then } e_1 \text{ else } e_2 : \tau} \quad \textbf{(T-If)}$$

$$\Gamma \vdash n : \texttt{Int} \quad \textbf{(T-Int)}$$

$$\Gamma \vdash \texttt{True} : \texttt{Bool} \quad \textbf{(T-True)}$$

$$\Gamma \vdash \texttt{False} : \texttt{Bool} \quad \textbf{(T-False)}$$

The evaluation rules describe the nature by which values transition between other values and determine the runtime behavior of the program.

$$\frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2} \quad \textbf{(E-App1)}$$

$$\frac{e_2 \to e_2'}{v_1 e_2 \to v_1 e_2'} \quad \textbf{(E-App2)}$$

$$(\lambda x : \tau . e_1) v_2 \to [x / v_2] e_1 \quad \textbf{(E-AppLam)}$$

$$\text{if } \texttt{True} \text{ then } e_2 \text{ else } e_3 \to e_2 \quad \textbf{(E-IfTrue)}$$

$$\text{if } \texttt{False} \text{ then } e_2 \text{ else } e_3 \to e_3 \quad \textbf{(E-IfFalse)}$$

$$\frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3} \quad \textbf{(E-If)}$$

Since we now have the notion of scoped variables for lambda, we will implement a typing environment `Env` as manifest as Γ in our typing rules.

```
type Env = [(Name, Type)]

extend :: (Name, Type) -> Env -> Env
extend xt env = xt : env

inEnv :: (Name, Type) -> Check a -> Check a
inEnv (x,t) = local (extend (x,t))

lookupVar :: Name -> Check Type
lookupVar x = do
  env <- ask
  case lookup x env of
    Just e  -> return e
    Nothing -> throwError $ NotInScope x
```

The typechecker will be a `ExceptT` + `Reader` monad transformer stack, with the reader holding the typing environment. There are three possible failure modes for our simply typed lambda calculus typechecker:

- The case when we try to unify two unlike types.
- The case when we try to apply a non-function to an argument.
- The case when a variable is referred to that is not in scope.

```
data TypeError
  = Mismatch Type Type
  | NotFunction Type
  | NotInScope Name

type Check = ExceptT TypeError (Reader Env)
```

There is a direct equivalence between syntax patterns here and the equivalent typing judgement for it. This will not always be the case in general though. The implementation of the type checker is as follows:

```
check :: Expr -> Check Type
check expr = case expr of

  Lit (LInt{}) -> return TInt

  Lit (LBool{}) -> return TBool

  Lam x t e -> do
    rhs <- inEnv (x,t) (check e)
    return (TArr t rhs)

  App e1 e2 -> do
```

```
    t1 <- check e1
    t2 <- check e2
    case t1 of
       (TArr a b) | a == t2 -> return b
                  | otherwise -> throwError $ Mismatch t2 a
       ty -> throwError $ NotFunction ty

  Var x -> lookupVar x
```

Evaluation

Fundamentally the evaluation of the typed lambda calculus is no different than the untyped lambda calculus, nor could it be since the syntactic addition of types is purely a static construction and cannot have any manifestation at runtime by definition. The only difference is that the simply typed lambda calculus admits strictly less programs than the untyped lambda calculus.

The foundational idea in compilation of static typed languages is that a typed program can be transformed into an untyped program by *erasing* type information but preserving the evaluation semantics of the typed program. If our program has *type safety* then it can never "go wrong" at runtime.

Of course the converse is not true, programs that do not "go wrong" are not necessarily well-typed, although whether we can prove whether a non well-typed program cannot go wrong is an orthogonal issue. The game that we as statically typed language implementors play is fundamentally one of restriction: we take the space of all programs and draw a large line around the universe of discourse of programs that we are willing to consider, since these are the only programs that we can prove properties for.

> *Well-typed programs don't go wrong, but not every program that never goes wrong is well-typed. It's easy to exhibit programs that don't go wrong but are ill-typed in … any … decidable type system. Many such programs are useful, which is why dynamically-typed languages like Erlang and Lisp are justly popular.*
>
> — Simon Peyton Jones

Power always comes at a price. Using one system you can do more things. In another you can say more about the things a program can do. The fundamental art in the discipline of language design is balancing the two to find the right power-to-weight ratio.

Observations

Some examples to try:

```
Stlc> (\x : Int . \y : Int . y) 1 2
2
```

```
Stlc> (\x : (Int -> Int). x) (\x : Int . 1) 2
1

Stlc> (\x : Int . x) False
Couldn't match expected type 'Int' with actual type: 'Bool'

Stlc> 1 2
Tried to apply to non-function type: Int

Stlc> (\x : Int . (\y : Int . x))
<<closure>>
```

Notation Reference

The notation introduced here will be used throughout the construction of the Haskell compiler. For reference here is a list of each of the notational conventions we will use. Some of these terms are not yet introduced.

Notation	Convention
$\{a, b, c\}$	Set
$\overline{\alpha}$	Vector
$e : \tau$	Type judgement
$P(x)$	Predicate
$P(x) : Q(x)$	Conditional
$P \vdash Q$	Implication
α, β	Type variables
Γ	Type context
x, y, z	Expression variables
e	Expression metavariable
τ	Type metavariable
κ	Kind metavariable
σ	Type scheme metavariable
C	Type constraint
$\tau_1 \sim \tau_2$	Unification constraint
$[\tau/\alpha]$	Substitution
s	Substitution metavariable
$[s]\tau$	Substitution application
$\tau_1 \to \tau_2$	Function type
$C \Rightarrow \tau$	Qualified type
$\tau_1 \times \tau_2$	Product type
$\tau_1 + \tau_2$	Sum type
$\bot$	Bottom type
$\forall \alpha.\tau$	Universal quantifier
$\exists \alpha.\tau$	Existential quantifier
`Nat`, `Bool`	Ground type

Full Source

- Typed Arithmetic
- Simply Typed Lambda Calculus

Evaluation

Well-typed programs cannot "go wrong".
— Robin Milner

Evaluation

While the lambda calculus is exceedingly simple, there is a great deal of variety in ways to evaluate and implement the reduction of lambda expressions. The different models for evaluation are *evaluation stratgies*.

There is a bifurcation between two points in the design space: *strict* and *non-strict* evaluation. An evaluation strategy is strict if the arguments to a lambda expression are necessarily evaluated before a lambda is reduced. A language in which the arguments are not necessarily evaluated before a lambda is reduced is non-strict.

Alternatively expressed, diverging terms are represented up to equivalence by the *bottom* value, written as $\perp$. A function f is non-strict if:

$$f\perp \neq \perp$$

Evaluation Models

There are many different models, and various hybrids thereof. We will consider three dominant models:

- Call-by-value: arguments are evaluated before a function is entered
- Call-by-name: arguments are passed unevaluated
- Call-by-need: arguments are passed unevaluated but an expression is only evaluated once and shared upon subsequent references

Given an expression fx the reduction in different evaluation models proceeds differently:

Call-by-value:

1. Evaluate x to v
2. Evaluate f to $\lambda y.e$
3. Evaluate $[y/v]e$

Call-by-name:

1. Evaluate f to $\lambda y.e$
2. Evaluate $[y/x]e$

Call-by-need:

1. Allocate a thunk v for x
2. Evaluate f to $\lambda y.e$
3. Evaluate $[y/v]e$

Terms that have a normal form in one model, may or may not have a normal form in another. In call-by-need and call-by-name evaluation diverging terms are not necessarily evaluated before entry, so some terms that have a normal form in these models may diverge under call-by-value.

Call-by-value

Call by value is an extremely common evaluation model. Many programming languages both imperative and functional use this evaluation strategy. The essence of call-by-value is that there are two categories of expressions: *terms* and *values*. Values are lambda expressions and other terms which are in normal form and cannot be reduced further. All arguments to a function will be reduced to normal form *before* they are bound inside the lambda and reduction only proceeds once the arguments are reduced.

For a simple arithmetic expression, the reduction proceeds as follows. Notice how the subexpression `(2 + 2)` is evaluated to normal form before being bound.

```
(\x. \y. y x) (2 + 2) (\x. x + 1)
=> (\x. \y. y x) 4 (\x. x + 1)
=> (\y. y 4) (\x. x + 1)
=> (\x. x + 1) 4
=> 4 + 1
=> 5
```

Naturally there are two evaluation rules for applications.

$$\frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2} \quad \textbf{(E-App1)}$$

$$\frac{e_2 \to e_2'}{v_1 e_2 \to v_1 e_2'} \quad \textbf{(E-App2)}$$

$$(\lambda x.e)v \to [x/v]e \quad \textbf{(E-AppLam)}$$

For a simple little lambda calculus the call-by-value interpreter is quite simple. Part of the runtime evaluation of lambda calculus involves the creation of *closures*, environments which hold the local variables in scope. In our little language there are two possible values which reduction may converge on, **VInt** and **VClosure**.

```haskell
data Expr
  = Var Int
  | Lam Expr
  | App Expr Expr
  | Lit Int
  | Prim PrimOp Expr Expr
  deriving Show

data PrimOp = Add | Mul
  deriving Show

data Value
  = VInt Int
  | VClosure Expr Env
  deriving Show

type Env = [Value]

emptyEnv :: Env
emptyEnv = []
```

The evaluator function simply maps the local scope and a term to the final value. Whenever a variable is referred to it is looked up in the environment. Whenever a lambda is entered it extends the environment with the local scope of the closure.

```haskell
eval :: Env -> Expr -> Value
eval env term = case term of
  Var n -> env !! n
  Lam a -> VClosure a env
  App a b ->
    let VClosure c env' = eval env a in
    let v = eval env b in
    eval (v : env') c

  Lit n -> VInt n
  Prim p a b -> (evalPrim p) (eval env a) (eval env b)

evalPrim :: PrimOp -> Value -> Value -> Value
evalPrim Add (VInt a) (VInt b) = VInt (a + b)
evalPrim Mul (VInt a) (VInt b) = VInt (a * b)
```

Call-by-name

In call-by-name evaluation, the arguments to lambda expressions are substituted as is, evaluation simply proceeds from left to right substituting the outermost lambda or reducing a value. If a substituted expression is not used it is never evaluated.

$$\frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2} \qquad \textbf{(E-App)}$$

$$(\lambda x.e_1)e_2 \to [x/e_2]e_1 \quad \textbf{(E-AppLam)}$$

For example, the same expression we looked at for call-by-value has the same normal form but arrives at it by a different sequence of reductions:

```
(\x. \y. y x) (2 + 2) (\x. x + 1)
=> (\y. y (2 + 2)) (\x. x + 1)
=> (\x. x + 1) (2 + 2)
=> (2 + 2) + 1
=> 4 + 1
=> 5
```

Call-by-name is non-strict, although very few languages use this model.

Call-by-need

Call-by-need is a special type of non-strict evaluation in which unevaluated expressions are represented by suspensions or *thunks* which are passed into a function unevaluated and only evaluated when needed or *forced*. When the thunk is forced the representation of the thunk is *updated* with the computed value and is not recomputed upon further reference.

The thunks for unevaluated lambda expressions are allocated when evaluated, and the resulting computed value is placed in the same reference so that subsequent computations share the result. If the argument is never needed it is never computed, which results in a trade-off between space and time.

Since the evaluation of subexpression does not follow any pre-defined order, any impure functions with side-effects will be evaluated in an unspecified order. As a result call-by-need can only effectively be implemented in a purely functional setting.

```
type Thunk = () -> IO Value

data Value
  = VBool Bool
  | VInt Integer
  | VClosure (Thunk -> IO Value)

update :: IORef Thunk -> Value -> IO ()
update ref v = do
  writeIORef ref (\() -> return v)
  return ()
```

```
force :: IORef Thunk -> IO Value
force ref = do
  th <- readIORef ref
  v <- th ()
  update ref v
  return v

mkThunk :: Env -> String -> Expr -> (Thunk -> IO Value)
mkThunk env x body = \a -> do
  a' <- newIORef a
  eval ((x, a') : env) body

eval :: Env -> Expr -> IO Value
eval env ex = case ex of
  EVar n -> do
    th <- lookupEnv env n
    v <- force th
    return v

  ELam x e -> return $ VClosure (mkThunk env x e)

  EApp a b -> do
    VClosure c <- eval env a
    c (\() -> eval env b)

  EBool b -> return $ VBool b
  EInt n  -> return $ VInt n
  EFix e  -> eval env (EApp e (EFix e))
```

For example, in this model the following program will not diverge since the omega combinator passed into the constant function is not used and therefore the argument is not evaluated.

```
omega = (\x -> x x) (\x -> x x)
test1 = (\y -> 42) omega

omega :: Expr
omega = EApp (ELam "x" (EApp (EVar "x") (EVar "x")))
             (ELam "x" (EApp (EVar "x") (EVar "x")))

test1 :: IO Value
test1 = eval [] $ EApp (ELam "y" (EInt 42)) omega
```

Higher Order Abstract Syntax (HOAS)

GHC Haskell being a rich language has a variety of extensions that, among other things, allow us to map lambda expressions in our defined language directly onto lambda expressions in Haskell. In this

case we will use a GADT to embed a Haskell expression inside our expression type.

```
{-# LANGUAGE GADTs #-}

data Expr a where
  Lift :: a                       -> Expr a
  Tup  :: Expr a -> Expr b        -> Expr (a, b)
  Lam  :: (Expr a -> Expr b)      -> Expr (a -> b)
  App  :: Expr (a -> b) -> Expr a -> Expr b
  Fix  :: Expr (a -> a)           -> Expr a
```

The most notable feature of this encoding is that there is no distinct constructor for variables. Instead they are simply values in the host language. Some example expressions:

```
id :: Expr (a -> a)
id = Lam (\x -> x)

tr :: Expr (a -> b -> a)
tr = Lam (\x -> (Lam (\y -> x)))

fl :: Expr (a -> b -> b)
fl = Lam (\x -> (Lam (\y -> y)))
```

Our evaluator then simply uses Haskell for evaluation.

```
eval :: Expr a -> a
eval (Lift v)    = v
eval (Tup e1 e2) = (eval e1, eval e2)
eval (Lam f)     = \x -> eval (f (Lift x))
eval (App e1 e2) = (eval e1) (eval e2)
eval (Fix f)     = (eval f) (eval (Fix f))
```

Some examples of use:

```
fact :: Expr (Integer -> Integer)
fact =
  Fix (
    Lam (\f ->
      Lam (\y ->
        Lift (
          if eval y == 0
          then 1
          else eval y * (eval f) (eval y - 1)))))
```

```
test :: Integer
test = eval fact 10

main :: IO ()
main = print test
```

Several caveats must be taken when working with HOAS. First of all, it takes more work to transform expressions in this form since in order to work with the expression we would need to reach under the lambda binder of a Haskell function itself. Since all the machinery is wrapped up inside of Haskell's implementation even simple operations like pretty printing and writing transformation passes can be more difficult. This form is a good form for evaluation, but not for transformation.

Parametric Higher Order Abstract Syntax (PHOAS)

A slightly different form of HOAS called PHOAS uses a lambda representation parameterized over the binder type under an existential type.

```
{-# LANGUAGE RankNTypes #-}

data ExprP a
  = VarP a
  | AppP (ExprP a) (ExprP a)
  | LamP (a -> ExprP a)
  | LitP Integer

newtype Expr = Expr { unExpr :: forall a . ExprP a }
```

The lambda in our language is simply a lambda within Haskell. As an example, the usual SK combinators would be written as follows:

```
-- i x = x
i :: ExprP a
i = LamP (\a -> VarP a)

-- k x y = x
k :: ExprP a
k = LamP (\x -> LamP (\y -> VarP x))

-- s f g x = f x (g x)
s :: ExprP a
s =
  LamP (\f ->
    LamP (\g ->
      LamP (\x ->
```

```
    AppP
      (AppP (VarP f) (VarP x))
      (AppP (VarP g) (VarP x))
    )))
```

Evaluation will result in a runtime `Value` type, just as before with our outer interpreters. We will use several "extractor" functions which use incomplete patterns under the hood. The model itself does not prevent malformed programs from blowing up here, and so it is necessary to guarantee that the program is sound before evaluation. Normally this would be guaranteed at a higher level by a typechecker before even reaching this point.

```
data Value
  = VLit Integer
  | VFun (Value -> Value)

fromVFun :: Value -> (Value -> Value)
fromVFun val = case val of
  VFun f -> f
  _      -> error "not a function"

fromVLit :: Value -> Integer
fromVLit val = case val of
  VLit n -> n
  _      -> error "not an integer"
```

Evaluation simply exploits the fact that nestled up under our existential type is just a Haskell function and so we get all the name capture, closures and binding machinery for free. The evaluation logic for PHOAS model is extremely short.

```
eval :: Expr -> Value
eval e = ev (unExpr e) where
  ev (LamP f)      = VFun(ev . f)
  ev (VarP v)      = v
  ev (AppP e1 e2)  = fromVFun (ev e1) (ev e2)
  ev (LitP n)      = VLit n
```

Consider the `S K K = I` example again and check the result:

```
skk :: ExprP a
skk = AppP (AppP s k) k

example :: Integer
example = fromVLit $ eval $ Expr (AppP skk (LitP 3))
```

We will use this evaluation technique extensively in writing interpreters for our larger languages. It is an extremely convenient and useful method for writing interpreters in Haskell.

Embedding IO

As mentioned before, effects are first class values in Haskell.

In Haskell we don't read from a file directly, but create a value that represents reading from a file. This allows us to very cleanly model an interpreter for our language inside of Haskell by establishing a mapping between the base operations of our language and existing function implementations of the standard operations in Haskell, and using monadic operations to build up a pure effectful computation as a result of interpretation. After evaluation, we finally lift the resulting IO value into Haskell and execute the results. This fits in nicely with the PHOAS model and allows us to efficiently implement a fully-fledged interpreter for our language with remarkably little code, simply by exploiting Haskell's implementation.

To embed IO actions inside of our interpreter we create a distinct `VEffect` value that will build up a sequenced IO computation during evaluation. This value will be passed off to Haskell and reified into real world effects.

```
data ExprP a
  = VarP a
  | GlobalP Name
  | AppP (ExprP a) (ExprP a)
  | LamP (a -> ExprP a)
  | LitP Char
  | EffectP a

data Value
  = VChar Char
  | VFun (Value -> Value)
  | VEffect (IO Value)
  | VUnit

fromVEff :: Value -> (IO Value)
fromVEff val = case val of
  VEffect f -> f
  _         -> error "not an effect"

eval :: Expr -> Value
eval e = ev (unExpr e) where
  ev (LamP f)      = VFun(ev . f)
  ev (AppP e1 e2)  = fromVFun (ev e1) (ev e2)
  ev (LitP n)      = VChar n
  ev (EffectP v)   = v
  ev (VarP v)      = v
  ev (GlobalP op)  = prim op

-- Lift an effect from our language into Haskell IO.
run :: Expr -> IO ()
run f = void (fromVEff (eval f))
```

The `prim` function will simply perform a lookup on the set of builtin operations, which we'll define with a bit of syntactic sugar for wrapping up Haskell functions.

```
unary :: (Value -> Value) -> Value
unary f = lam $ \a -> f a

binary :: (Value -> Value -> Value) -> Value
binary f = lam $ \a ->
           lam $ \b -> f a b

prim :: Name -> Value
prim op = case op of
 "putChar#" -> unary $ \x ->
    VEffect $ do
      putChar (fromVChar x)
      return VUnit

 "getChar#" -> VEffect $ do
      val <- getChar
      return (VChar val)

 "bindIO#"   -> binary $ \x y -> bindIO x y
 "returnIO#" -> unary  $ \x   -> returnIO x
 "thenIO#"   -> binary $ \x y -> thenIO x y
```

For example `thenIO#` sequences effects in our language will simply squash two `VEffect` objects into one composite effect building up a new `VEffect` value that is using Haskell's monadic sequencing on the internal `IO` value.

```
bindIO :: Value -> Value -> Value
bindIO (VEffect f) (VFun g) = VEffect (f >>= fromVEff . g)

thenIO :: Value -> Value -> Value
thenIO (VEffect f) (VEffect g) = VEffect (f >> g)

returnIO :: Value -> Value
returnIO a = VEffect $ return a
```

Effectively we're just recreating the same conceptual relationship that Haskell IO has with its runtime, but instead our host language uses Haskell as the runtime!

Full Source

Evaluation

- Call-by-value
- Call-by-need

Higher Order Interpreters

- HOAS
- PHOAS
- Embedding IO

Hindley-Milner Inference

There is nothing more practical than a good theory.
— James C. Maxwell

Hindley-Milner Inference

The Hindley-Milner type system (also referred to as Damas-Hindley-Milner or HM) is a family of type systems that admit the serendipitous property of having a tractable algorithm for determining types from untyped syntax. This is achieved by a process known as *unification*, whereby the types for a well-structured program give rise to a set of constraints that when solved always have a unique *principal type*.

The simplest Hindley Milner type system is defined by a very short set of rules. The first four rules describe the judgements by which we can map each syntactic construct (`Lam`, `App`, `Var`, `Let`) to their expected types. We'll elaborate on these rules shortly.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \textbf{(T-Var)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad \textbf{(T-App)}$$

$$\frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x\ .\ e : \tau_1 \to \tau_2} \quad \textbf{(T-Lam)}$$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \quad \textbf{(T-Let)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \overline{\alpha} \notin \texttt{ftv}(\Gamma)}{\Gamma \vdash e : \forall\, \overline{\alpha}\ .\ \sigma} \quad \textbf{(T-Gen)}$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \quad \textbf{(T-Inst)}$$

Milner's observation was that since the typing rules map uniquely onto syntax, we can in effect run the typing rules "backwards" and whenever we don't have a known type for a subexpression, we "guess" by putting a fresh variable in its place, collecting constraints about its usage induced by subsequent typing judgements. This is the essence of *type inference* in the ML family of languages, that by the generation and solving of a class of unification problems we can reconstruct the types uniquely from the syntax. The algorithm itself is largely just the structured use of a unification solver.

However full type inference leaves us in a bit a bind, since while the problem of inference is tractable within this simple language and trivial extensions thereof, but nearly any major addition to the language destroys the ability to infer types unaided by annotation or severely complicates the inference algorithm. Nevertheless the Hindley-Milner family represents a very useful, productive "sweet spot" in the design space.

Syntax

The syntax of our first type inferred language will effectively be an extension of our untyped lambda calculus, with fixpoint operator, booleans, integers, let, and a few basic arithmetic operations.

```
type Name = String

data Expr
  = Var Name
  | App Expr Expr
  | Lam Name Expr
  | Let Name Expr Expr
  | Lit Lit
  | If Expr Expr Expr
  | Fix Expr
  | Op Binop Expr Expr
  deriving (Show, Eq, Ord)

data Lit
  = LInt Integer
  | LBool Bool
  deriving (Show, Eq, Ord)

data Binop = Add | Sub | Mul | Eql
  deriving (Eq, Ord, Show)

data Program = Program [Decl] Expr deriving Eq

type Decl = (String, Expr)
```

The parser is trivial, the only addition will be the toplevel let declarations (`Decl`) which are joined into the global `Program`. All toplevel declarations must be terminated with a semicolon, although they can span multiple lines and whitespace is ignored. So for instance:

```
-- SKI combinators
let I x = x;
let K x y = x;
let S f g x = f x (g x);
```

As before `let rec` expressions will expand out in terms of the fixpoint operator and are just syntactic sugar.

Polymorphism

We will add an additional constructs to our language that will admit a new form of *polymorphism* for our language. Polymorphism is the property of a term to simultaneously admit several distinct types for the same function implementation.

For instance the polymorphic signature for the identity function maps an input of type α

$$\begin{aligned} \texttt{id} &:: \forall\alpha.\alpha \to \alpha \\ \texttt{id} &= \lambda x : \alpha.\ x \end{aligned}$$

Now instead of having to duplicate the functionality for every possible type (i.e. implementing idInt, idBool, ...) we our type system admits any instantiation that is subsumed by the polymorphic type signature.

$$\begin{aligned} \texttt{id}_{\texttt{Int}} &= \texttt{Int} \to \texttt{Int} \\ \texttt{id}_{\texttt{Bool}} &= \texttt{Bool} \to \texttt{Bool} \end{aligned}$$

A rather remarkably fact of universal quantification is that many properties about inhabitants of a type are guaranteed by construction, these are the so-called *free theorems*. For instance any (nonpathological) inhabitant of the type `(a, b) -> a` must be equivalent to `fst`.

A slightly less trivial example is that of the `fmap` function of type `Functor f => (a -> b) -> f a -> f b`. The second functor law demands that:

```
forall f g. fmap f . fmap g = fmap (f . g)
```

However it is impossible to write down a (nonpathological) function for `fmap` that has the required type and doesn't have this property. We get the theorem for free!

Types

The type language we'll use starts with the simple type system we used for our typed lambda calculus.

```
newtype TVar = TV String
  deriving (Show, Eq, Ord)

data Type
  = TVar TVar
  | TCon String
  | TArr Type Type
```

```
  deriving (Show, Eq, Ord)

typeInt, typeBool :: Type
typeInt  = TCon "Int"
typeBool = TCon "Bool"
```

Type schemes model polymorphic types, they indicate that the type variables bound in quantifier are polymorphic across the enclosed type and can be instantiated with any type consistent with the signature. Intuitively the indicate that the implementation of the function

```
data Scheme = Forall [TVar] Type
```

Type schemes will be written as σ in our typing rules.

$$\begin{aligned}\sigma ::= & \tau \\ & \forall \overline{\alpha}.\tau\end{aligned}$$

For example the `id` and the `const` functions would have the following types:

$$\begin{aligned}\text{id} &: \forall a.a \to a \\ \text{const} &: \forall ab.a \to b \to a\end{aligned}$$

We've now divided our types into two syntactic categories, the *monotypes* and the *polytypes*. In our simple initial languages type schemes will always be the representation of top level signature, even if there are no polymorphic type variables. In implementation terms this means when a monotype is yielded from our Infer monad after inference, we will immediately generalize it at the toplevel *closing over* all free type variables in a type scheme.

Context

The typing context or environment is the central container around which all information during the inference process is stored and queried. In Haskell our implementation will simply be a newtype wrapper around a Map of `Var` to `Scheme` types.

```
newtype TypeEnv = TypeEnv (Map.Map Var Scheme)
```

The two primary operations are *extension* and *restriction* which introduce or remove named quantities from the context.

$$\Gamma \backslash x = \{y : \sigma | y : \sigma \in \Gamma, x \neq y\}$$

$$\Gamma, x : \tau = (\Gamma \backslash x) \cup \{x : \tau\}$$

Operations over the context are simply the usual Set operations on the underlying map.

```
extend :: TypeEnv -> (Var, Scheme) -> TypeEnv
extend (TypeEnv env) (x, s) = TypeEnv $ Map.insert x s env
```

Inference Monad

All our logic for type inference will live inside of the `Infer` monad. It is a monad transformer stack of `ExcpetT` + `State`, allowing various error reporting and statefully holding the fresh name supply.

```
type Infer a = ExceptT TypeError (State Unique) a
```

Running the logic in the monad results in either a type error or a resulting type scheme.

```
runInfer :: Infer (Subst, Type) -> Either TypeError Scheme
runInfer m = case evalState (runExceptT m) initUnique of
  Left err  -> Left err
  Right res -> Right $ closeOver res
```

Substitution

Two operations that will perform quite a bit are querying the free variables of an expression and applying substitutions over expressions.

$$
\begin{aligned}
\texttt{fv}(x) &= x \\
\texttt{fv}(\lambda x.e) &= \texttt{fv}(e) - \{x\} \\
\texttt{fv}(e_1 e_2) &= \texttt{fv}(e_1) \cup \texttt{fv}(e_2)
\end{aligned}
$$

The same pattern applies to type variables at the type level.

$$
\begin{aligned}
\texttt{ftv}(\alpha) &= \{\alpha\} \\
\texttt{ftv}(\tau_1 \to \tau_2) &= \texttt{ftv}(\tau_1) \cup \texttt{ftv}(\tau_2) \\
\texttt{ftv}(\texttt{Int}) &= \varnothing \\
\texttt{ftv}(\texttt{Bool}) &= \varnothing \\
\texttt{ftv}(\forall x.t) &= \texttt{ftv}(t) - \{x\}
\end{aligned}
$$

Substitutions over expressions apply the substitution to local variables, replacing the named subexpression if matched. In the case of name capture a fresh variable is introduced.

$$
\begin{aligned}
[x/e']x &= e' \\
[x/e']y &= y \quad (y \neq x) \\
[x/e'](e_1 e_2) &= ([x/e']\ e_1)([x/e']e_2) \\
[x/e'](\lambda y.e_1) &= \lambda y.[x/e']e \quad y \neq x, y \notin \texttt{fv}(e')
\end{aligned}
$$

And likewise, substitutions can be applied element wise over the typing environment.

$$[t/s]\Gamma = \{y : [t/s]\sigma \mid y : \sigma \in \Gamma\}$$

Our implementation of a substitution in Haskell is simply a Map from type variables to types.

```
type Subst = Map.Map TVar Type
```

Composition of substitutions ($s_1 \circ s_2$, `s1 ‘compose‘ s2`) can be encoded simply as operations over the underlying map. Importantly note that in our implementation we have chosen the substitution to be left-biased, it is up to the implementation of the inference algorithm to ensure that clashes do not occur between substitutions.

```
nullSubst :: Subst
nullSubst = Map.empty

compose :: Subst -> Subst -> Subst
s1 ‘compose‘ s2 = Map.map (apply s1) s2 ‘Map.union‘ s1
```

The implementation in Haskell is via a series of implementations of a `Substitutable` typeclass which exposes an `apply` function which applies the substitution given over the structure of the type replacing type variables as specified.

```
class Substitutable a where
  apply :: Subst -> a -> a
  ftv   :: a -> Set.Set TVar

instance Substitutable Type where
  apply _ (TCon a)       = TCon a
  apply s t@(TVar a)     = Map.findWithDefault t a s
  apply s (t1 ‘TArr‘ t2) = apply s t1 ‘TArr‘ apply s t2

  ftv TCon{}         = Set.empty
  ftv (TVar a)       = Set.singleton a
  ftv (t1 ‘TArr‘ t2) = ftv t1 ‘Set.union‘ ftv t2

instance Substitutable Scheme where
  apply s (Forall as t)   = Forall as $ apply s' t
                            where s' = foldr Map.delete s as
  ftv (Forall as t) = ftv t ‘Set.difference‘ Set.fromList as

instance Substitutable a => Substitutable [a] where
  apply = fmap . apply
  ftv   = foldr (Set.union . ftv) Set.empty
```

```haskell
instance Substitutable TypeEnv where
  apply s (TypeEnv env) =  TypeEnv $ Map.map (apply s) env
  ftv (TypeEnv env) = ftv $ Map.elems env
```

Throughout both the typing rules and substitutions we will require a fresh supply of names. In this naive version we will simply use an infinite list of strings and slice into n'th element of list per an index that we hold in a State monad. This is a simplest implementation possible, and later we will adapt this name generation technique to be more robust.

```haskell
letters :: [String]
letters = [1..] >>= flip replicateM ['a'..'z']

fresh :: Infer Type
fresh = do
  s <- get
  put s{count = count s + 1}
  return $ TVar $ TV (letters !! count s)
```

The creation of fresh variables will be essential for implementing the inference rules. Whenever we encounter the first use of a variable within some expression we will create a fresh type variable.

Unification

Central to the idea of inference is the notion of *unification*. A unifier for two expressions e_1 and e_2 is a substitution s such that:

$$s := [n_0/m_0, n_1/m_1, ..., n_k/m_k][s]e_1 = [s]e_2$$

Two terms are said to be *unifiable* if there exists a unifying substitution set between them. A substitution set is said to be *confluent* if the application of substitutions is independent of the order applied, i.e. if we always arrive at the same normal form regardless of the order of substitution chosen.

We'll adopt the notation

$$\tau \sim \tau' : s$$

for the fact that two types τ, τ' are unifiable by a substitution s, such that:

$$[s]\tau = [s]\tau'$$

Two identical terms are trivially unifiable by the empty unifier.

$$c \sim c : [\,]$$

The unification rules for our little HM language are as follows:

$$c \sim c : [] \qquad \textbf{(Uni-Const)}$$

$$\alpha \sim \alpha : [] \qquad \textbf{(Uni-Var)}$$

$$\frac{\alpha \notin \texttt{ftv}(\tau)}{\alpha \sim \tau : [\alpha/\tau]} \qquad \textbf{(Uni-VarLeft)}$$

$$\frac{\alpha \notin \texttt{ftv}(\tau)}{\tau \sim \alpha : [\alpha/\tau]} \qquad \textbf{(Uni-VarRight)}$$

$$\frac{\tau_1 \sim \tau_1' : \theta_1 \qquad [\theta_1]\tau_2 \sim [\theta_1]\tau_2' : \theta_2}{\tau_1\tau_2 \sim \tau_1'\tau_2' : \theta_2 \circ \theta_1} \qquad \textbf{(Uni-Con)}$$

$$\frac{\tau_1 \sim \tau_1' : \theta_1 \qquad [\theta_1]\tau_2 \sim [\theta_1]\tau_2' : \theta_2}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' : \theta_2 \circ \theta_1} \qquad \textbf{(Uni-Arrow)}$$

If we want to unify a type variable α with a type τ, we usually can just substitute the variable with the type: $[\alpha/\tau]$. However, our rules state a precondition known as the *occurs check* for that unification: the type variable α must not occur free in τ. If it did, the substitution would not be a unifier.

Take for example the problem of unifying α and $\alpha \to \beta$. The substitution $s = [\alpha/\alpha \to \beta]$ doesn't unify: we get

$$[s]\alpha = \alpha \to \beta$$

and

$$[s]\alpha \to \beta = (\alpha \to \beta) \to \beta.$$

Indeed, whatever substitution s we try, $[s]\alpha \to \beta$ will always be longer than $[s]\alpha$, so no unifier exists. The only chance would be to substitute with an infinite type: $[\alpha/(\ldots((\alpha \to \beta) \to \beta) \to \cdots \to \beta) \to \beta]$ would be a unifier, but our language has no such types.

If the unification fails because of the occurs check, we say that unification would give an infinite type.

Note that unifying $\alpha \to \beta$ and α is exactly what we would have to do if we tried to type check the omega combinator $\lambda x.xx$, so it is ruled out by the occurs check, as are other pathological terms we discussed when covering the untyped lambda calculus.

```
occursCheck ::  Substitutable a => TVar -> a -> Bool
occursCheck a t = a `Set.member` ftv t
```

The unify function lives in the Infer monad and yields a subsitution:

```
unify ::  Type -> Type -> Infer Subst
unify (l `TArr` r) (l' `TArr` r')  = do
```

```
    s1 <- unify l l'
    s2 <- unify (apply s1 r) (apply s1 r')
    return (s2 `compose` s1)

unify (TVar a) t = bind a t
unify t (TVar a) = bind a t
unify (TCon a) (TCon b) | a == b = return nullSubst
unify t1 t2 = throwError $ UnificationFail t1 t2

bind ::  TVar -> Type -> Infer Subst
bind a t | t == TVar a     = return nullSubst
         | occursCheck a t = throwError $ InfiniteType a t
         | otherwise       = return $ Map.singleton a t
```

Generalization and Instantiation

At the heart of Hindley-Milner is two fundamental operations:

- **Generalization**: Converting a τ type into a σ type by closing over all free type variables in a type scheme.
- **Instantiation**: Converting a σ type into a τ type by creating fresh names for each type variable that does not appear in the current typing environment.

$$\frac{\Gamma \vdash e : \sigma \quad \overline{\alpha} \notin \texttt{ftv}(\Gamma)}{\Gamma \vdash e : \forall\, \overline{\alpha}\, .\, \sigma} \quad (\textbf{T-Gen})$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \quad (\textbf{T-Inst})$$

The $\sqsubseteq$ operator in the (**T-Inst**) rule indicates that a type is an *instantiation* of a type scheme.

$$\forall \overline{\alpha}.\tau_2 \sqsubseteq \tau_1$$

A type τ_1 is a instantiation of a type scheme $\sigma = \forall \overline{\alpha}.\tau_2$ if there exists a substitution $[s]\beta = \beta$ for all $\beta \in \texttt{ftv}(\sigma)$ so that $\tau_1 = [s]\tau_2$. Some examples:

$$\forall a.a \to a \sqsubseteq \texttt{Int} \to \texttt{Int}$$
$$\forall a.a \to a \sqsubseteq b \to b$$
$$\forall ab.a \to b \to a \sqsubseteq \texttt{Int} \to \texttt{Bool} \to \texttt{Int}$$

These map very intuitively into code that simply manipulates the Haskell `Set` objects of variables and the fresh name supply:

```
instantiate ::  Scheme -> Infer Type
instantiate (Forall as t) = do
  as' <- mapM (const fresh) as
  let s = Map.fromList $ zip as as'
  return $ apply s t

generalize :: TypeEnv -> Type -> Scheme
generalize env t  = Forall as t
    where as = Set.toList $ ftv t `Set.difference` ftv env
```

By convention let-bindings are generalized as much as possible. So for instance in the following definition `f` is generalized across the body of the binding so that at each invocation of `f` it is instantiated with fresh type variables.

```
Poly> let f = (\x -> x) in let g = (f True) in f 3
3 : Int
```

In this expression, the type of `f` is generated at the let definition and will be instantiated with two different signatures. At call site of `f` it will unify with `Int` and the other unify with `Bool`.

By contrast, binding `f` in a lambda will result in a type error.

```
Poly> (\f -> let g = (f True) in (f 3)) (\x -> x)
Cannot unify types:
    Bool
with
    Int
```

This is the essence of *let generalization*.

Typing Rules

And finally with all the typing machinery in place, we can write down the typing rules for our simple little polymorphic lambda calculus.

```
infer :: TypeEnv -> Expr -> Infer (Subst, Type)
```

The `infer` maps the local typing environment and the active expression to a 2-tuple of the partial unifier solution and the intermediate type. The AST is traversed bottom-up and constraints are solved at each level of recursion by applying partial substitutions from unification across each partially inferred subexpression and the local environment. If an error is encountered the `throwError` is called in the `Infer` monad and an error is reported.

```haskell
infer :: TypeEnv -> Expr -> Infer (Subst, Type)
infer env ex = case ex of

  Var x -> lookupEnv env x

  Lam x e -> do
    tv <- fresh
    let env' = env `extend` (x, Forall [] tv)
    (s1, t1) <- infer env' e
    return (s1, apply s1 tv `TArr` t1)

  App e1 e2 -> do
    tv <- fresh
    (s1, t1) <- infer env e1
    (s2, t2) <- infer (apply s1 env) e2
    s3       <- unify (apply s2 t1) (TArr t2 tv)
    return (s3 `compose` s2 `compose` s1, apply s3 tv)

  Let x e1 e2 -> do
    (s1, t1) <- infer env e1
    let env' = apply s1 env
        t'   = generalize env' t1
    (s2, t2) <- infer (env' `extend` (x, t')) e2
    return (s1 `compose` s2, t2)

  If cond tr fl -> do
    (s1, t1) <- infer env cond
    (s2, t2) <- infer env tr
    (s3, t3) <- infer env fl
    s4 <- unify t1 typeBool
    s5 <- unify t2 t3
    return (s5 `compose` s4 `compose` s3 `compose` s2 `compose` s1, apply s5 t2)

  Fix e1 -> do
    (s1, t) <- infer env e1
    tv <- fresh
    s2 <- unify (TArr tv tv) t
    return (s2, apply s1 tv)

  Op op e1 e2 -> do
    (s1, t1) <- infer env e1
    (s2, t2) <- infer env e2
    tv <- fresh
    s3 <- unify (TArr t1 (TArr t2 tv)) (ops Map.! op)
    return (s1 `compose` s2 `compose` s3, apply s3 tv)
```

```
  Lit (LInt _)  -> return (nullSubst, typeInt)
  Lit (LBool _) -> return (nullSubst, typeBool)
```

Let's walk through each of the rule derivations and look how it translates into code:

T-Var

The `T-Var` rule, simply pull the type of the variable out of the typing context.

```
  Var x -> lookupEnv env x
```

The function `lookupVar` looks up the local variable reference in typing environment and if found it instantiates a fresh copy.

```
lookupEnv :: TypeEnv -> Var -> Infer (Subst, Type)
lookupEnv (TypeEnv env) x = do
  case Map.lookup x env of
    Nothing -> throwError $ UnboundVariable (show x)
    Just s  -> do t <- instantiate s
                  return (nullSubst, t)
```

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad (\textbf{T-Var})$$

T-Lam

For lambdas the variable bound by the lambda is locally scoped to the typing environment and then the body of the expression is inferred with this scope. The output type is a fresh type variable and is unified with the resulting inferred type.

```
  Lam x e -> do
    tv <- fresh
    let env' = env `extend` (x, Forall [] tv)
    (s1, t1) <- infer env' e
    return (s1, apply s1 tv `TArr` t1)
```

$$\frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x\, .\, e : \tau_1 \to \tau_2} \quad (\textbf{T-Lam})$$

T-App

For applications, the first argument must be a lambda expression or return a lambda expression, so know it must be of form `t1 -> t2` but the output type is not determined except by the confluence of the two values. We infer both types, apply the constraints from the first argument over the result second inferred type and then unify the two types with the excepted form of the entire application expression.

```
  App e1 e2 -> do
    tv <- fresh
    (s1, t1) <- infer env e1
    (s2, t2) <- infer (apply s1 env) e2
    s3       <- unify (apply s2 t1) (TArr t2 tv)
    return (s3 `compose` s2 `compose` s1, apply s3 tv)
```

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad (\textbf{T-App})$$

T-Let

As mentioned previously, let will be generalized so we will create a local typing environment for the body of the let expression and add the generalized inferred type let bound value to the typing environment of the body.

```
  Let x e1 e2 -> do
    (s1, t1) <- infer env e1
    let env' = apply s1 env
        t'   = generalize env' t1
    (s2, t2) <- infer (env' `extend` (x, t')) e2
    return (s1 `compose` s2, t2)
```

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \quad (\textbf{T-Let})$$

T-BinOp

There are several builtin operations, we haven't mentioned up to now because the typing rules are trivial. We simply unify with the preset type signature of the operation.

```
 Op op e1 e2 -> do
    (s1, t1) <- infer env e1
    (s2, t2) <- infer env e2
    tv <- fresh
    s3 <- unify (TArr t1 (TArr t2 tv)) (ops Map.! op)
    return (s1 `compose` s2 `compose` s3, apply s3 tv)

ops :: Map.Map Binop Type
ops = Map.fromList [
      (Add, (typeInt `TArr` (typeInt `TArr` typeInt)))
    , (Mul, (typeInt `TArr` (typeInt `TArr` typeInt)))
    , (Sub, (typeInt `TArr` (typeInt `TArr` typeInt)))
    , (Eql, (typeInt `TArr` (typeInt `TArr` typeBool)))
  ]
```

$$
\begin{aligned}
(+) &: \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
(\times) &: \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
(-) &: \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
(=) &: \texttt{Int} \to \texttt{Int} \to \texttt{Bool}
\end{aligned}
$$

Literals

The type of literal integer and boolean types is trivially their respective types.

$$\frac{}{\Gamma \vdash n : \texttt{Int}} \quad \textbf{(T-Int)}$$

$$\frac{}{\Gamma \vdash \texttt{True} : \texttt{Bool}} \quad \textbf{(T-True)}$$

$$\frac{}{\Gamma \vdash \texttt{False} : \texttt{Bool}} \quad \textbf{(T-False)}$$

Constraint Generation

The previous implementation of Hindley Milner is simple, but has this odd property of intermingling two separate processes: constraint solving and traversal. Let's discuss another implementation of the inference algorithm that does not do this.

In the *constraint generation* approach, constraints are generated by bottom-up traversal, added to a ordered container, canonicalized, solved, and then possibly back-substituted over a typed AST. This will be the approach we will use from here out, and while there is an equivalence between the "on-line solver", using the separate constraint solver becomes easier to manage as our type system gets more complex and we start building out the language.

Our inference monad now becomes a `RWST` (Reader-Writer-State Transformer) + `Except` for typing errors. The inference state remains the same, just the fresh name supply.

```
-- | Inference monad
type Infer a = (RWST
                  Env             -- Typing environment
                  [Constraint]    -- Generated constraints
                  InferState      -- Inference state
                  (Except         -- Inference errors
                    TypeError)
                  a)              -- Result

-- | Inference state
data InferState = InferState { count :: Int }
```

Instead of unifying type variables at each level of traversal, we will instead just collect the unifiers inside the Writer and emit them with the `uni` function.

```
-- | Unify two types
uni :: Type -> Type -> Infer ()
uni t1 t2 = tell [(t1, t2)]
```

Since the typing environment is stored in the Reader monad, we can use the `local` to create a locally scoped additions to the typing environment. This is convenient for typing binders.

```
-- | Extend type environment
inEnv :: (Name, Scheme) -> Infer a -> Infer a
inEnv (x, sc) m = do
  let scope e = (remove e x) `extend` (x, sc)
  local scope m
```

Typing

The typing rules are identical, except they now can be written down in a much less noisy way that isn't threading so much state. All of the details are taken care of under the hood and encoded in specific combinators manipulating the state of our Infer monad in a way that lets focus on the domain logic.

```
infer :: Expr -> Infer Type
infer expr = case expr of
  Lit (LInt _)  -> return $ typeInt
  Lit (LBool _) -> return $ typeBool

  Var x -> lookupEnv x

  Lam x e -> do
    tv <- fresh
    t <- inEnv (x, Forall [] tv) (infer e)
    return (tv `TArr` t)

  App e1 e2 -> do
    t1 <- infer e1
    t2 <- infer e2
    tv <- fresh
    uni t1 (t2 `TArr` tv)
    return tv

  Let x e1 e2 -> do
    env <- ask
    t1 <- infer e1
    let sc = generalize env t1
    t2 <- inEnv (x, sc) (infer e2)
    return t2
```

```
  Fix e1 -> do
    t1 <- infer e1
    tv <- fresh
    uni (tv `TArr` tv) t1
    return tv

  Op op e1 e2 -> do
    t1 <- infer e1
    t2 <- infer e2
    tv <- fresh
    let u1 = t1 `TArr` (t2 `TArr` tv)
        u2 = ops Map.! op
    uni u1 u2
    return tv

  If cond tr fl -> do
    t1 <- infer cond
    t2 <- infer tr
    t3 <- infer fl
    uni t1 typeBool
    uni t2 t3
    return t2
```

Constraint Solver

The Writer layer for the Infer monad contains the generated set of constraints emitted from inference pass. Once inference has completed we are left with a resulting type signature full of meaningless unique fresh variables and a set of constraints that we must solve to refine the type down to its principal type.

The constraints are pulled out solved by a separate `Solve` monad which holds the Unifier (most general unifier) solution that when applied to generated signature will yield the solution.

```
type Constraint = (Type, Type)

type Unifier = (Subst, [Constraint])

-- | Constraint solver monad
type Solve a = StateT Unifier (ExceptT TypeError Identity) a
```

The unification logic is also identical to before, except it is now written independent of inference and stores its partial state inside of the Solve monad's state layer.

```
unifies :: Type -> Type -> Solve Unifier
unifies t1 t2 | t1 == t2 = return emptyUnifer
unifies (TVar v) t = v `bind` t
```

```
unifies t (TVar v) = v `bind` t
unifies (TArr t1 t2) (TArr t3 t4) = unifyMany [t1, t2] [t3, t4]
unifies t1 t2 = throwError $ UnificationFail t1 t2

unifyMany :: [Type] -> [Type] -> Solve Unifier
unifyMany [] [] = return emptyUnifer
unifyMany (t1 : ts1) (t2 : ts2) =
  do (su1,cs1) <- unifies t1 t2
     (su2,cs2) <- unifyMany (apply su1 ts1) (apply su1 ts2)
     return (su2 `compose` su1, cs1 ++ cs2)
unifyMany t1 t2 = throwError $ UnificationMismatch t1 t2
```

The solver function simply iterates over the set of constraints, composing them and applying the resulting constraint solution over the intermediate solution eventually converting on the *most general unifier* which yields the final subsitution which when applied over the inferred type signature, yields the principal type solution for the expression.

```
-- Unification solver
solver :: Solve Subst
solver = do
  (su, cs) <- get
  case cs of
    [] -> return su
    ((t1, t2): cs0) -> do
      (su1, cs1)  <- unifies t1 t2
      put (su1 `compose` su, cs1 ++ (apply su1 cs0))
      solver
```

This a much more elegant solution than having to intermingle inference and solving in the same pass, and adapts itself well to the generation of a typed Core form which we will discuss in later chapters.

Worked Examples

Let's walk through two examples of how inference works for simple functions.

Example 1

Consider:

```
\x y z -> x + y + z
```

The generated type from the `infer` function consists simply of a fresh variable for each of the arguments and the return type.

```
a -> b -> c -> e
```

The constraints induced from **T-BinOp** are emitted as we traverse both of the addition operations.

1. `a -> b -> d  ~  Int -> Int -> Int`
2. `d -> c -> e  ~  Int -> Int -> Int`

Here `d` is the type of the intermediate term `x + y`. By applying **Uni-Arrow** we can then deduce the following set of substitutions.

1. `a ~ Int`
2. `b ~ Int`
3. `c ~ Int`
4. `d ~ Int`
5. `e ~ Int`

Substituting this solution back over the type yields the inferred type:

```
Int -> Int -> Int -> Int
```

Example 2

```
compose f g x = f (g x)
```

The generated type from the `infer` function consists again simply of unique fresh variables.

```
a -> b -> c -> e
```

Induced by two cases of the **T-App** rule we get the following constraints:

1. `b ~ c -> d`
2. `a ~ d -> e`

Here `d` is the type of `(g x)`. The constraints are already in a canonical form, by applying **Uni-VarLeft** twice we get the following set of substitutions:

1. `b ~ c -> d`
2. `a ~ d -> e`

So we get this type:

```
compose :: forall c d e. (d -> e) -> (c -> d) -> c -> e
```

If desired, we can rename the variables in alphabetical order to get:

```
compose :: forall a b c. (a -> b) -> (c -> a) -> c -> b
```

Interpreter

Our evaluator will operate directly on the syntax and evaluate the results in into a `Value` type.

```
data Value
  = VInt Integer
  | VBool Bool
  | VClosure String Expr TermEnv
```

The interpreter is set up an Identity monad. Later it will become a more complicated monad, but for now its quite simple. The value environment will explicitly threaded around, and whenever a closure is created we simply store a copy of the local environment in the closure.

```
type TermEnv = Map.Map String Value
type Interpreter t = Identity t
```

Our logic for evaluation is an extension of the lambda calculus evaluator implemented in previous chapter. However you might notice quite a few incomplete patterns used throughout evaluation. Fear not though, the evaluation of our program cannot "go wrong". Each of these patterns represents a state that our type system guarantees will never happen. For example, if our program did have not every variable referenced in scope then it would never reach evaluation to begin with and would be rejected in our type checker. We are morally correct in using incomplete patterns here!

```
eval :: TermEnv -> Expr -> Interpreter Value
eval env expr = case expr of
  Lit (LInt k)  -> return $ VInt k
  Lit (LBool k) -> return $ VBool k

  Var x -> do
    let Just v = Map.lookup x env
    return v

  Op op a b -> do
    VInt a' <- eval env a
    VInt b' <- eval env b
    return $ (binop op) a' b'

  Lam x body ->
    return (VClosure x body env)

  App fun arg -> do
    VClosure x body clo <- eval env fun
    argv <- eval env arg
    let nenv = Map.insert x argv clo
    eval nenv body
```

```
  Let x e body -> do
    e' <- eval env e
    let nenv = Map.insert x e' env
    eval nenv body

  If cond tr fl -> do
    VBool br <- eval env cond
    if br == True
    then eval env tr
    else eval env fl

  Fix e -> do
    eval env (App e (Fix e))

binop :: Binop -> Integer -> Integer -> Value
binop Add a b = VInt $ a + b
binop Mul a b = VInt $ a * b
binop Sub a b = VInt $ a - b
binop Eql a b = VBool $ a == b
```

Interactive Shell

Our language has now grown out the small little shells we were using before, and now we need something much more robust to hold the logic for our interactive interpreter.

We will structure our REPL as a monad wrapped around IState (the interpreter state) datatype. We will start to use the repline library from here out which gives us platform independent readline, history, and tab completion support.

```
data IState = IState
  { tyctx :: TypeEnv  -- Type environment
  , tmctx :: TermEnv  -- Value environment
  }

initState :: IState
initState = IState emptyTyenv emptyTmenv

type Repl a = HaskelineT (StateT IState IO) a

hoistErr :: Show e => Either e a -> Repl a
hoistErr (Right val) = return val
hoistErr (Left err) = do
  liftIO $ print err
  abort
```

Our language can be compiled into a standalone binary by GHC:

```
$ ghc --make Main.hs -o poly
$ ./poly
Poly>
```

At the top of our program we will look at the command options and allow three variations of commands.

```
$ poly                # launch shell
$ poly input.ml       # launch shell with 'input.ml' loaded
$ poly test input.ml  # dump test for 'input.ml' to stdout
```

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    []      -> shell (return ())
    [fname] -> shell (load [fname])
    ["test", fname] -> shell (load [fname] >> browse [] >> quit ())
    _ -> putStrLn "invalid arguments"
```

The shell command takes a `pre` action which is run before the shell starts up. The logic simply evaluates our Repl monad into an IO and runs that from the main function.

```
shell :: Repl a -> IO ()
shell pre
  = flip evalStateT initState
  $ evalRepl "Poly> " cmd options completer pre
```

The `cmd` driver is the main entry point for our program, it is executed every time the user enters a line of input. The first argument is the line of user input.

```
cmd :: String -> Repl ()
cmd source = exec True (L.pack source)
```

The heart of our language is then the `exec` function which imports all the compiler passes, runs them sequentially threading the inputs and outputs and eventually yielding a resulting typing environment and the evaluated result of the program. These are monoidally joined into the state of the interpreter and then the loop yields to the next set of inputs.

```
exec :: Bool -> L.Text -> Repl ()
exec update source = do
  -- Get the current interpreter state
  st <- get
```

```
  -- Parser ( returns AST )
  mod <- hoistErr $ parseModule "<stdin>" source

  -- Type Inference ( returns Typing Environment )
  tyctx' <- hoistErr $ inferTop (tyctx st) mod

  -- Create the new environment
  let st' = st { tmctx = foldl' evalDef (tmctx st) mod
               , tyctx = tyctx' <> (tyctx st)
               }

  -- Update the interpreter state
  when update (put st')
```

Repline also supports adding special casing certain sets of inputs so that they map to builtin commands in the compiler. We will implement three of these.

Command	Action
`:browse`	Browse the type signatures for a program
`:load <file>`	Load a program from file
`:type`	Show the type of an expression
`:quit`	Exit interpreter

Their implementations are mostly straightforward.

```
options :: [(String, [String] -> Repl ())]
options = [
    ("load"   , load)
  , ("browse" , browse)
  , ("quit"   , quit)
  , ("type"   , Main.typeof)
  ]

-- :browse command
browse :: [String] -> Repl ()
browse _ = do
  st <- get
  liftIO $ mapM_ putStrLn $ ppenv (tyctx st)

-- :load command
load :: [String] -> Repl ()
load args = do
  contents <- liftIO $ L.readFile (unwords args)
```

```
  exec True contents

-- :type command
typeof :: [String] -> Repl ()
typeof args = do
  st <- get
  let arg = unwords args
  case Infer.typeof (tyctx st) arg of
    Just val -> liftIO $ putStrLn $ ppsignature (arg, val)
    Nothing -> exec False (L.pack arg)

-- :quit command
quit :: a -> Repl ()
quit _ = liftIO $ exitSuccess
```

Finally tab completion for our shell will use the interpreter's typing environment keys to complete on the set of locally defined variables. Repline supports prefix based tab completion where the prefix of the current command will be used to determine what to tab complete. In the case where we start with the command `:load` we will instead tab complete on filenames in the current working directly instead.

```
completer :: CompleterStyle (StateT IState IO)
completer = Prefix (wordCompleter comp) defaultMatcher

-- Prefix tab completer
defaultMatcher :: MonadIO m => [(String, CompletionFunc m)]
defaultMatcher = [
    (":load"  , fileCompleter)
  ]

-- Default tab completer
comp :: (Monad m, MonadState IState m) => WordCompleter m
comp n = do
  let cmds = [":load", ":browse", ":quit", ":type"]
  TypeEnv ctx <- gets tyctx
  let defs = Map.keys ctx
  return $ filter (isPrefixOf n) (cmds ++ defs)
```

Observations

There we have it, our first little type inferred language! Load the `poly` interpreter by running `ghci Main.hs` and the call the `main` function.

```
$ ghci Main.hs
λ: main
Poly> :load test.ml
Poly> :browse
```

Try out some simple examples by declaring some functions at the toplevel of the program. We can query the types of expressions interactively using the `:type` command which effectively just runs the expression halfway through the pipeline and halts after typechecking.

```
Poly> let id x = x
Poly> let const x y = x
Poly> let twice x = x + x

Poly> :type id
id : forall a. a -> a

Poly> :type const
const : forall a b. a -> b -> a

Poly> :type twice
twice : Int -> Int
```

Notice several important facts. Our type checker will now flat our reject programs with scoping errors before interpretation.

```
Poly> \x -> y
Not in scope: "y"
```

Also programs that are also not well-typed are now rejected outright as well.

```
Poly> 1 + True
Cannot unify types:
    Bool
with
    Int
```

The omega combinator will not pass the occurs check.

```
Poly> \x -> x x
Cannot construct the the infinite type: a = a -> b
```

The file `test.ml` provides a variety of tests of the little interpreter. For instance both `fact` and `fib` functions uses the fixpoint to compute Fibonacci numbers or factorials.

```
let fact = fix (\fact -> \n ->
  if (n == 0)
    then 1
    else (n * (fact (n-1))));
```

```
let rec fib n =
  if (n == 0)
    then 0
    else if (n==1)
      then 1
      else ((fib (n-1)) + (fib (n-2)));

Poly> :type fact
fact : Int -> Int

Poly> fact 5
120

Poly> fib 16
610
```

Full Source

- Poly
- Poly - Constraint Generation

ProtoHaskell

Design of ProtoHaskell

Now that we've completed our simple little ML language, let's discuss the road ahead toward building a more complex language we'll call *ProtoHaskell* that will eventually become the full *Fun* language.

Language	Chapters	Description
Poly	1 - 8	Minimal type inferred ML-like language.
ProtoHaskell	8 - 18	Interpreted minimal Haskell subset.
Fun	18 - 27	ProtoHaskell with native code generator.

The defining feature of ProtoHaskell is that it is independent of an evaluation model, so hypothetically one could write either a lazy or a strict backend and use the same frontend.

Before we launch into writing compiler passes let's look at the overview of where we're going, the scope of what we're going to do, and what needs to be done to get there. *We will refer to concepts that are not yet introduced, so keep in mind this is meant to be a high-level overview of the ProtoHaskell compiler pipeline.*

Haskell: A Rich Language

Haskell itself is a beautifully simple language at its core, although the implementation of GHC is arguably anything but simple! The more one digs into the implementation the more it becomes apparent that a lot of care and forethought was given to making the frontend language as expressive as it is. Many of these details require a great detail of engineering work to make them work as seamlessly as they do.

Consider this simple Haskell example but note how much of an extension this is from our simple little ML interpreter.

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred [] = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      =     filter pred xs
```

Consider all the things that are going on just in this simple example.

- Lazy evaluation
- Custom datatypes

- Higher order functions
- Parametric polymorphism
- Function definition by pattern matching
- Pattern matching desugaring
- Guards to distinguish sub-cases
- Type signature must subsume inferred type
- List syntactic sugar (value/pattern syntax)

Clearly we're going to need a much more sophisticated design, and we'll likely be doing quite a bit more bookkeeping about our program during compilation.

Scope

Considering our project is intended to be a simple toy language, we are not going to implement all of Haskell 2010. Doing so in its entirety would actually be a fairly involved effort. However we will implement a sizable chunk of the functionality, certainly enough to write non-trivial programs and implement most of the standard Prelude.

Things we will implement:

- Indentation sensitive grammar
- Pattern matching
- Algebraic datatypes
- Where statements
- Recursive functions/datatypes
- Operator sections
- Implicit let-rec
- List and tuple sugar
- Records
- Custom operators
- Do-notation
- Type annotations
- Monadic IO
- Typeclasses
- Arithmetic primops
- Type synonyms
- List comprehensions

Things we will not implement are:

- Overloaded literals
- GADTs
- Polymorphic recursion

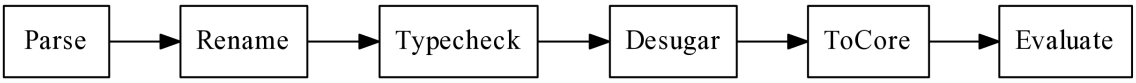
- Any GHC-specific language extensions.
- Newtypes
- Module namespaces
- Operator parameters
- Defaulting rules
- Exceptions
- Parallelism
- Software Transactional Memory
- Foreign Function Interface

Now if one feels so inclined one could of course implement these features on top of our final language, but they are left as an exercise to the reader!

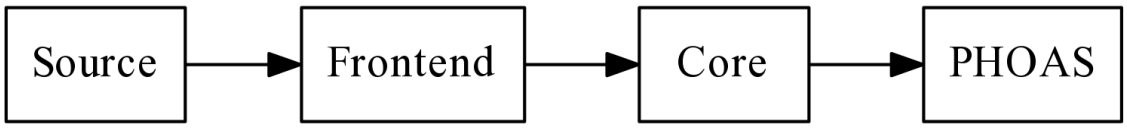
This of course begs the question of whether or not our language is "a Haskell". In the strictest sense, it will not be since it doesn't fully conform to either the Haskell 98 or Haskell 2010 language specifications. However in terms of the colloquial usage of the term Haskell, there does seem to be some growing feeling that the "Haskell language family" does exist as a definable subset of the functional programming design space, although many people disagree what its defining features are. In this sense we will most certainly be writing a language in the Haskell family.

Intermediate Forms

The passes between each of the phases make up the main *compilation pipeline* .



For *ProtoHaskell* our pipeline consists of the transitions between four intermediate forms of the program.



- The **Source**, the textual representation of the program from a file or user input. This is stored in a `Text` type.
- The **Frontend** source, the untyped AST generated from the parser.
- The **Core**, the explicitly typed, desugared form of the program generated after type inference.

- The **PHOAS**, the type-erased Core is transformed into Haskell expressions by mapping lambda expressions in our language directly into Haskell lambda expressions and then evaluated using the Haskell runtime.

Pass	Rep	Haskell Type
Parsing	Source	`Text.Text`
Desugaring	Frontend	`Frontend.Expr`
Typechecking	Core	`Core.Expr`
Evaluation	PHOAS	`CoreEval.ExprP`

For our later *Fun* language our pipeline builds on top of the *ProtoHaskell* but instead of going to an interpreter it will be compiled into native code through the native code generator (on top of LLVM) and compiled into a binary executable or evaluated by a just-in-time (JIT) compiler.

Pass	Rep	Haskell Type
Parsing	Source	`Text.Text`
Desugaring	Frontend	`Frontend.Expr`
Typechecking	Core	`Core.Expr`
Transformation	STG	`STG.Expr`
Transformation	Imp	`Imp.Expr`
Code Generation	LLVM	`LLVM.General.Module`

Compiler Monad

The main driver of the compiler will be a `ExceptT` + `State` + `IO` transformer stack . All other passes and transformations in the compiler will hang off of this monad, which encapsulates the main compilation pipeline.

```
type CompilerMonad =
  ExceptT Msg
    (StateT CompilerState IO)

data CompilerState = CompilerState
  { _fname    :: Maybe FilePath           -- ^ File path
  , _imports  :: [FilePath]               -- ^ Loaded modules
  , _src      :: Maybe L.Text             -- ^ File source
  , _ast      :: Maybe Syn.Module         -- ^ Frontend AST
  , _tenv     :: Env.Env                  -- ^ Typing environment
  , _kenv     :: Map.Map Name Kind        -- ^ Kind environment
  , _cenv     :: ClassEnv.ClassEnv        -- ^ Typeclass environment
  , _cast     :: Maybe Core.Module        -- ^ Core AST
```

```
  , _flags    :: Flags.Flags               -- ^ Compiler flags
  , _venv     :: CoreEval.ValEnv Core.Expr -- ^ Core interpreter environment
  , _denv     :: DataEnv.DataEnv           -- ^ Entity dictionary
  , _clenv    :: ClassEnv.ClassHier        -- ^ Typeclass hierarchy
  } deriving (Eq, Show)
```

The compiler itself will have several entry points, `expr` for interactive evaluation that expects an expression object and joins it into accumulated interactive environment. And `modl` path that compile whole modules.

Throughout the next 10 chapters we will incrementally create a series of transformations with the following type signatures.

```
parseP   :: FilePath -> L.Text -> CompilerM Syn.Module
dataP    :: Syn.Module -> CompilerM Syn.Module
groupP   :: Syn.Module -> CompilerM Syn.Module
renameP  :: Syn.Module -> CompilerM Syn.Module
desugarP :: Syn.Module -> CompilerM Syn.Module
inferP   :: Syn.Module -> CompilerM Core.Module
evalP    :: Core.Module -> CompilerM ()
```

The code path for `modl` is then simply the passes composed with the Kleisli composition operator to form the composite pipeline for compiling modules.

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

And that's basically the entire structure of the compiler. It's just a pipeline of monadic actions for each pass rolled up inside of `CompilerM`.

```
modl :: FilePath -> L.Text -> CompilerM ()
modl fname
    = parseP fname
  >=> dataP
  >=> groupP
  >=> renameP
  >=> desugarP
  >=> inferP
  >=> evalP
```

Engineering Overview

REPL

It is important to have an interactive shell to be able to interactively explore the compilation steps and intermediate forms for arbitrary expressions. GHCi does this very well, and nearly every intermediate form is inspectable. We will endeavor to recreate this experience with our toy language.

If the ProtoHaskell compiler is invoked either in GHCi or as standalone executable, you will see a similar interactive shell.

```
 _   _           _        | ProtoHaskell Compiler 0.1.0
| | | | __ _ ___| | __    | Copyright (c) 2013-2015 Stephen Diehl
| |_| |/ _` / __| |/ /    | Released under the MIT License
|  _  | (_| \__ \   <     |
|_| |_|\__,_|___/_|\_\    | Type :help for help

Compiling module: prelude.fun
> id (1+2)
3
> :type (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
> :set -ddump-rn
> :load test.fun
```

Command line conventions will follow GHCi's naming conventions. There will be a strong emphasis on building debugging systems on top of our architecture so that when subtle bugs creep up you will have the tools to diagnose the internal state of the type system and detect flaws in the implementation.

Command	Action
`:browse`	Browse the type signatures for a program
`:load <file>`	Load a program from file
`:reload`	Run the active file
`:edit`	Edit the active file in system editor
`:core`	Show the core of an expression or program
`:module`	Show active modules imports
`:source`	Show the source code of an expression or program
`:type`	Show the type of an expression
`:kind`	Show the kind of an expression
`:set <flag>`	Set a flag
`:unset <flag>`	Unset a flag
`:constraints`	Dump the typing constraints for an expression
`:quit`	Exit interpreter

The most notable difference is the very important `:core` command which will dump out the core representation of any expression given in the interactive shell. Another one is the `:constraints` command which will interactively walk you through the type checker's reasoning about how it derived the type it did for a given expression.

```
> :type plus
plus :: forall a. Num a => a -> a -> a
```

```
> :core id
id :: forall a. a -> a
id = \(ds1 : a) -> a

> :core compose
compose :: forall c d e. (d -> e) -> (c -> d) -> c -> e
compose = \(ds1 : d -> e)
           (ds2 : c -> d)
           (ds3 : c) ->
            (ds1 (ds2 ds3))
```

The flags we use also resemble GHC's and allow dumping out the pretty printed form of each of the intermediate transformation passes.

- `-ddump-parsed`
- `-ddump-desugar`
- `-ddump-rn`
- `-ddump-infer`
- `-ddump-core`
- `-ddump-types`
- `-ddump-stg`
- `-ddump-imp`
- `-ddump-c`
- `-ddump-llvm`
- `-ddump-asm`
- `-ddump-to-file`

The implementation of the interactive shell will use a custom library called repline , which is a higher-level wrapper on top of `haskeline` made to be more pleasant when writing interactive shells.

Parser

We will use the normal Parsec parser with a few extensions. We will add indentation sensitive parsing so that block syntax (where statements, let statements, do-notation) can be parsed.

```
main :: IO ()
main = do
  putStrLn msg
  where
    msg = "Hello World"
```

We will also need to allow the addition of infix operators from user-defined declarations, and allow this information to be used during parsing.

```
infixl 6 +
infixl 7 *

f = 1 + 2 * 3
```

Renamer

After parsing we will traverse the entire AST and rename all user-named variables to machine generated names and eliminate any name-shadowing. For example in the following ambiguous binder will replace the duplicate occurrence of `x` with a fresh name.

```
f x y = \g x -> x + y    -- x in definition of g shadows x in f
f x y = \g a0 -> a0 + y
```

We will also devise a general method of generating fresh names for each pass such that the names generated are uniquely relatable to that pass and cannot conflict with later passes.

Ensuring that all names are unique in the syntax tree will allow us more safety later on during program transformation, to know that names cannot implicitly capture and the program can be transformed without changing its meaning.

Datatypes

User defined data declarations need to be handled and added to the typing context so that their use throughout the program logic can be typechecked. This will also lead us into the construction of a simple kind inference system, and the support of higher-kinded types.

```
data Bool = False | True
data Maybe a = Nothing | Just a
data T1 f a = T1 (f a)
```

Each constructor definition will also introduce several constructor functions into the Core representation of the module. Record types will also be supported and will expand out into selectors for each of the various fields.

Desugaring

Pattern matching is an extremely important part of a modern functional programming language, but the implementation of the pattern desugaring is remarkably subtle. The frontend syntax allows the expression of nested pattern matches and incomplete patterns, both can generate very complex *splitting trees* of case expressions that need to be expanded out recursively.

Multiple Equations

For instance the following toplevel pattern for the `xor` function is transformed into the following nested set of case statements:

```
-- Frontend
xor False False = False;
xor False True = True;
xor True False = True;
xor True True = False;

-- Desugared
xor :: Bool -> Bool -> Bool
xor = \_a _b -> case _a of {
                  False -> case _b of {
                             False -> False;
                             True -> True
                           };
                  True -> case _b of {
                            False -> True;
                            True -> False
                          }
                }
```

Constructor Patterns

Toplevel declarations in the frontend language can consist of patterns for on the right-hand-side of the declaration, while in the Core language these are transformed into case statements in the body of the function.

```
-- Frontend
f (Left  l) = a
f (Right r) = b

-- Desugared
f x = case x of
    Left  l -> a
    Right r -> b
```

Nested Patterns

The frontend language also allows nested constructors in a single pattern, while in the Core language these are expanded out into two case statements which scrutinize only one level of pattern.

```
-- Frontend
f x = case x of
  Just (Just y) -> y

-- Desugared
f x = case x of
  Just _a -> case _a of
    Just _b -> _b
```

There are many edge cases of pattern matching that we will have to consider. The confluence of all them gives rise to a rather complex set of AST rewrites:

- Multiple arguments
- Overlapping patterns
- Literal patterns
- Nested patterns
- Non-exhaustive equations
- Conditional equations
- Non-linear patterns

On top of pattern matching we will implement the following more trivial syntactic sugar translations:

- Expand `if/then` statements into case expressions.
- Expand pattern guards into case expressions.
- Expand out do-notation for monads.
- Expand list syntactic sugar.
- Expand tuple syntactic sugar.
- Expand out operator sections.
- Expand out string literals.
- Expand out numeric literals.

We will however punt on an important part of the Haskell specification, namely *overloaded literals*. In Haskell numeric literals are replaced by specific functions from the `Num` or `Fractional` typeclasses.

```
-- Frontend
42 :: Num a => a
3.14 :: Fractional a => a

-- Desugared
fromInteger (42 :: Integer)
fromRational (3.14 :: Rational)
```

We will not implement this, as it drastically expands the desugarer scope.

We will however follow GHC's example in manifesting unboxed types as first class values in the language so literals that appear in the AST are rewritten in terms of the wired-in constructors (`Int#`, `Char#`, `Addr#`, etc).

```
I# : Int# -> Int
C# : Char# -> Char

> :core 1
I# 1#
```

```
> :core 1 + 2
plus (I# 1#) (I# 2#)
> :core "snazzleberry"
unpackCString# "snazzleberry"#
```

Core

The Core language is the result of translation of the frontend language into an explicitly typed form. Just like GHC we will use a System-F variant, although unlike GHC we will effectively just be using vanilla System-F without all of the extensions (coercions, equalities, roles, etc) that GHC uses to implement more complicated features like GADTs and type families.

This is one of the most defining feature of GHC Haskell, its compilation into a statically typed intermediate Core language. It is a well-engineers detail of GHC's design that has informed much of how Haskell the language has evolved as a language with a exceedingly large frontend language that all melts away into a very tiny concise set of abstractions. Just like GHC we will extract all our language into a small core, with just a few constructors.

```
data Expr
  = App Expr Expr
  | Var Var
  | Lam Name Type Expr
  | Case Expr [Alt]
  | Let Bind Expr
  | Lit Literal
  | Placeholder Name Type

data Var
  = Id Name Type
  | TyVar Name Kind
```

The types and kind types are also equally small.

```
data Type
  = TVar TVar
  | TCon TyCon
  | TApp Type Type
  | TArr Type Type
  | TForall [Pred] [TVar] Type

data Kind
  = KStar
  | KArr Kind Kind
  | KPrim
  | KVar Name
```

Since the Core language is explicitly typed, it is trivial to implement an internal type checker for it. Running the typechecker on the generated core is a good way to catch optimization and desugaring bugs, and determine if the compiler has produced invalid intermediate code.

Type Classes

Typeclasses are also remarkably subtle to implement. We will implement just single parameter typeclasses and use the usual *dictionary passing translation* when compiling the frontend to Core. Although the translation and instance search logic is not terribly complicated, it is however very verbose and involves a lot of bookkeeping about the global typeclass hierarchy.

For example the following simplified `Num` typeclass generates quite a bit of elaborated definitions in the Core language to generate the dictionary and selector functions for the overloaded `plus` function.

```
class Num a where
  plus :: a -> a -> a
  mult :: a -> a -> a
  sub :: a -> a -> a

instance Num Int where
  plus = plusInt
  mult = multInt
  sub  = subInt

plusInt :: Int -> Int -> Int
plusInt (I# a) (I# b) = I# (plusInt# a b)
```

This expands into the following set of Core definitions.

```
plusInt :: Int -> Int -> Int
plusInt = \(ds1 : Int)
           (ds2 : Int) ->
            case ds1 of {
              I# ds8 ->
                case ds2 of {
                  I# ds9 ->
                    case (plusInt# ds8 ds9) of {
                      __DEFAULT {ds5} -> (I# ds5)
                    }
                }
            }

dplus :: forall a. DNum a -> a -> a -> a
dplus = \(tpl : DNum a) ->
          case tpl of {
```

```
            DNum a b c -> a
          }

plus :: forall e. Num e => e -> e -> e
plus = \($dNum_a : DNum e)
     (ds1 : e)
     (ds2 : e) ->
      (dplus $dNum_a ds1 ds2)
```

Our typeclass infrastructure will be able to support the standard typeclass hierarchy from the Prelude. Our instance search mechanism will be subject to the same restriction rules that GHC enforces.

- Paterson condition
- Coverage condition
- Bounded context stack

Type Checker

The type checker is the largest module and probably the most nontrivial part of our compiler. The module consists of roughly 1200 lines of code. Although the logic is not drastically different from the simple little HM typechecker we wrote previously, it simply has to do more bookkeeping and handle more cases.

The implementation of the typechecker will be split across four modules:

- `Infer.hs` - Main inference driver
- `Unify.hs` - Unification logic
- `ClassEnv.hs` - Typeclass instance resolution
- `Elaboration.hs` - Typeclass elaboration

The monad itself is a `RWST` + `Except` stack, with State holding the internal state of the inference engine and Writer gathering the generated constraint set that is passed off to the solver.

```
-- | Inference monad
type InferM = RWST
   Env             -- Typing environment
   [Constraint]    -- Generated constraints
   InferMState     -- Inference state
     (Except       -- Inference errors
       TypeError)

-- | Inference state
data InferMState = InferMState
  { count     :: Int         -- ^ Name supply
```

```
  , preds     :: [Pred]      -- ^ Typeclass predicates
  , skolems   :: Skolems     -- ^ Skolem scope
  , overloads :: ClassEnv    -- ^ Overloaded identifiers
  , active    :: Name        -- ^ Active function name
  , classh    :: ClassHier   -- ^ Typeclass hierarchy
  }
```

In *Fun* we will extend the simple type checker with arbitrary rank polymorphism (i.e. `RankNTypes` in GHC). This is actually required to implement typeclasses in their full generality, although in *ProtoHaskell* we will cheat a little bit.

Interpreter

For ProtoHaskell we will actually directly evaluate the resulting Core expressions in an interpreter. By virtue of us translating the expressions into Haskell expressions we will get lazy evaluation almost for free and will let us run our programs from inside of the interactive shell.

```
sieve :: [Int] -> [Int]
sieve [] = []
sieve (p:ps) = p : sieve (filter (nonMultiple p) ps)

nonMultiple :: Int -> Int -> Bool
nonMultiple p n = ((n/p)*p) /= n

primes :: [Int]
primes = sieve [2..]
```

```
ProtoHaskell> take 5 (cycle [1,2])
[1,2,1,2,1]

ProtoHaskell> take 5 primes
[2,3,5,7,11]
```

Error Reporting

We will do quite a bit of error reporting for the common failure modes of the type checker, desugar, and rename phases including position information tracking in Fun. However doing this in full is surprisingly involved and would add a significant amount of code to the reference implementation. As such we will not be as thorough as GHC in handling every failure mode by virtue of the scope of our project being a toy language with the primary goal being conciseness and simplicity.

Frontend

The Frontend language for ProtoHaskell is a fairly large language, consisting of many different types. Let's walk through the different constructions. The frontend syntax is split across several datatypes.

- `Decls` - Declarations syntax
- `Expr` - Expressions syntax
- `Lit` - Literal syntax
- `Pat` - Pattern syntax
- `Types` - Type syntax
- `Binds` - Binders

At the top is the named *Module* and all toplevel declarations contained therein. The first revision of the compiler has a very simple module structure, which we will extend later in fun with imports and public interfaces.

```
data Module = Module Name [Decl]          -- ^ module T where { .. }
  deriving (Eq,Show)
```

Declarations or `Decl` objects are any construct that can appear at toplevel of a module. These are namely function, datatype, typeclass, and operator definitions.

```
data Decl
  = FunDecl BindGroup                     -- ^ f x = x + 1
  | TypeDecl Type                         -- ^ f :: Int -> Int
  | DataDecl Constr [Name] [ConDecl]      -- ^ data T where { ... }
  | ClassDecl [Pred] Name [Name] [Decl]   -- ^ class (P) => T where { ... }
  | InstDecl [Pred] Name Type [Decl]      -- ^ instance (P) => T where { ... }
  | FixityDecl FixitySpec                 -- ^ infixl 1  {..}
  deriving (Eq, Show)
```

A binding group is a single line of definition for a function declaration. For instance the following function has two binding groups.

```
factorial :: Int -> Int

-- Group #1
factorial 0 = 1

-- Group #2
factorial n = n * factorial (n - 1)
```

One of the primary roles of the desugarer is to merge these disjoint binding groups into a single splitting tree of case statements under a single binding group.

```
data BindGroup = BindGroup
  { _matchName  :: Name
  , _matchPats  :: [Match]
  , _matchType  :: Maybe Type
  , _matchWhere :: [[Decl]]
  } deriving (Eq, Show)
```

The expression or `Expr` type is the core AST type that we will deal with and transform most frequently. This is effectively a simple untyped lambda calculus with let statements, pattern matching, literals, type annotations, if/these/else statements and do-notation.

```
type Constr = Name

data Expr
  = EApp  Expr Expr         -- ^ a b
  | EVar  Name              -- ^ x
  | ELam  Name Expr         -- ^ \\x . y
  | ELit  Literal           -- ^ 1, 'a'
  | ELet  Name Expr Expr    -- ^ let x = y in x
  | EIf   Expr Expr Expr    -- ^ if x then tr else fl
  | ECase Expr [Match]      -- ^ case x of { p -> e; ... }
  | EAnn  Expr Type         -- ^ ( x : Int )
  | EDo  [Stmt]             -- ^ do { ... }
  | EFail                   -- ^ pattern match fail
  deriving (Eq, Show)
```

Inside of case statements will be a distinct pattern matching syntax, this is used both at the toplevel function declarations and inside of case statements.

```
data Match = Match
  { _matchPat :: [Pattern]
  , _matchBody :: Expr
  , _matchGuard :: [Guard]
  } deriving (Eq, Show)

data Pattern
  = PVar Name               -- ^ x
  | PCon Constr [Pattern]   -- ^ C x y
  | PLit Literal            -- ^ 3
  | PWild                   -- ^ _
  deriving (Eq, Show)
```

The do-notation syntax is written in terms of two constructions, one for monadic binds and the other for monadic statements.

```
data Stmt
  = Generator Pattern Expr -- ^ pat <- exp
  | Qualifier Expr         -- ^ exp
  deriving (Eq, Show)
```

Literals are the atomic wired-in types that the compiler has knowledge of and will desugar into the appropriate builtin datatypes.

```
data Literal
  = LitInt Int           -- ^ 1
  | LitChar Char         -- ^ 'a'
  | LitString [Word8]    -- ^ "foo"#
  deriving (Eq, Ord, Show)
```

For data declarations we have two categories of constructor declarations that can appear in the body, regular constructors and record declarations. We will adopt the Haskell `-XGADTSyntax` for all data declarations.

```
-- Regular Syntax
data Person = Person String Int

-- GADTSyntax
data Person where
  Person :: String -> Int -> Person

-- Record Syntax
data Person where
  Person :: Person { name :: String, age :: Int }
```

```
data ConDecl
  = ConDecl Constr Type                -- ^ T :: a -> T a
  | RecDecl Constr [(Name, Type)] Type -- ^ T :: { label :: a } -> T a
  deriving (Eq, Show, Ord)
```

Fixity declarations are simply a binding between the operator symbol and the fixity information.

```
data FixitySpec = FixitySpec
  { fixityFix :: Fixity
  , fixityName :: String
  } deriving (Eq, Show)

data Assoc = L | R | N
  deriving (Eq,Ord,Show)
```

```
data Fixity
  = Infix Assoc Int
  | Prefix Int
  | Postfix Int
  deriving (Eq,Ord,Show)
```

Data Declarations

Data declarations are named blocks of various *ConDecl* constructors for each of the fields or constructors of a user-defined datatype.

```
data qname [var] where
  [tydecl]
```

```
data Unit where
  Unit :: Unit
```

```
DataDecl
      (Name "Unit")
      []
      [ ConDecl
          (Name "Unit") (Forall [] [] (TCon AlgTyCon { tyId = Name "Unit" }))
      ]
```

Function Declarations

Function declarations create *FunDecl* with *BindGroup* for the pattern on the left hand side of the toplevel declaration. The `matchPat` is simply the sequence of patterns (`PVar`, `PCon`, `PLit`) on the left hand side. If a type annotation is specified it is stored in the `matchType` field. Likewise, if there is a sequence of where statements these are also attached directly to the declaration, and will later be desugared away into local let statements across the body of the function.

```
qname [pat] = rhs [where decls]
```

```
const x y = x
```

```
FunDecl
  BindGroup
    { _matchName = Name "const"
    , _matchPats =
        [ Match
            { _matchPat = [ PVar (Name "x") , PVar (Name "y") ]
            , _matchBody = EVar (Name "x")
```

```
            }
        ]
    , _matchType = Nothing
    , _matchWhere = [ [] ]
    }
```

Pattern matches across multiple lines are treated as two separate declarations, which will later be grouped on the `matchName` in the desugaring phase. So for instance the `map` function has the following representation:

```
map f [] = []
map f (x:xs) = Cons (f x) (map f xs)
```

```
FunDecl
  BindGroup
    { _matchName = Name "map"
    , _matchPats =
        [ Match
            { _matchPat = [ PVar (Name "f") , PCon (Name "Nil") [] ]
            , _matchBody = EVar (Name "Nil")
            }
        ]
    , _matchType = Nothing
    , _matchWhere = [ [] ]
    }

FunDecl
  BindGroup
    { _matchName = Name "map"
    , _matchPats =
        [ Match
            { _matchPat =
                [ PVar (Name "f")
                , PCon (Name "Cons") [ PVar (Name "x") , PVar (Name "xs") ]
                ]
            , _matchBody =
                EApp
                  (EApp
                     (EVar (Name "Cons")) (EApp (EVar (Name "f")) (EVar (Name "x"))))
                  (EApp
                     (EApp (EVar (Name "map")) (EVar (Name "f"))) (EVar (Name "xs")))
            }
        ]
    , _matchType = Nothing
    , _matchWhere = [ [] ]
    }
```

Fixity Declarations

Fixity declarations are exceedingly simple, they store the binding precedence of the declaration along with its associativity (Left, Right, Non-Associative) and the infix symbol.

```
[infixl|infixr|infix] [integer] ops;
```

```
infixl 4 +;
```

```
FixityDecl
  FixitySpec { fixityFix = Infix L 4 , fixityName = "+" }
```

Typeclass Declarations

Typeclass declarations consist simply of the list of typeclass constraints, the name of the class, and the type variable (single parameter only). The body of the class is simply a sequence of scoped `FunDecl` declarations with only the `matchType` field.

```
class [context] => classname [var] where
  [body]
```

Consider a very simplified `Num` class.

```
class Num a where
  plus :: a -> a -> a
```

```
ClassDecl
  []
  (Name "Num")
  [ Name "a" ]
  [ FunDecl
      BindGroup
        { _matchName = Name "plus"
        , _matchPats = []
        , _matchType =
            Just
              (Forall
                 []
                 []
                 (TArr
                    (TVar TV { tvName = Name "a" })
                    (TArr
                       (TVar TV { tvName = Name "a" })
```

```
                        (TVar TV { tvName = Name "a" }))))
        , _matchWhere = []
        }
  ]
```

Typeclass instances follow the same pattern, they are simply the collection of instance constraints, the name of the typeclass, and the *head* of the type class instance type. The declarations are a sequence of `FunDecl` objects with the bodies of the functions for each of the overloaded function implementations.

```
instance [context] => head where
  [body]
```

For example:

```
instance Num Int where
  plus = plusInt
```

```
InstDecl
  []
  (Name "Num")
  (TCon AlgTyCon { tyId = Name "Int" })
  [ FunDecl
      BindGroup
        { _matchName = Name "plus"
        , _matchPats =
            [ Match { _matchPat = [] , _matchBody = EVar (Name "plusInt") } ]
        , _matchType = Nothing
        , _matchWhere = [ [] ]
        }
  ]
```

Wired-in Types

While the base Haskell is quite small, several portions of the desugaring process require the compiler to be aware about certain types before they are otherwise defined in the Prelude. For instance the type of every guarded pattern in the typechecker is `Bool`. These are desugared into a case statement that includes the `True` and `False` constructors. The Bool type is therefore necessarily baked into the syntax of the language and is inseparable from the implementation.

```
sign x
  |  x >  0  =  1
  |  x == 0  =  0
  |  x <  0  = -1
```

These are called the *wired-in types*, and while they are still defined in our Prelude they will have somewhat special status. The primitive types (of kind `#`) will be very special and are not user extensible, they map directly to implementation details of the code generation backend or Haskell functions hard-wired into the interpreter.

Syntax	Name	Kind	Description
`Int#`	Int#	`#`	Machine integer
`Char#`	Char#	`#`	Machine char
`Double#`	Double#	`#`	Machine double
`Addr#`	Addr#	`#`	Heap address
`Int`	Int	`*`	Boxed integer
`Char`	Char	`*`	Boxed char
`Double`	Double	`*`	Boxed double
`[]`	List	`* -> *`	List
`(,)`	Pair	`* -> * -> *`	2-Tuple
`()`	Unit	`*`	Unit type
`(->)`	Arrow	`* -> * -> *`	Arrow type
`Bool`	Bool	`*`	Boolean
`IO`	IO	`* -> *`	IO Monad

Traversals

Bottom-up traversals and rewrites in a monadic context are so common that we'd like to automate this process so that we don't have to duplicate the same logic across all our code. So we'll write several generic traversal functions.

```
descendM :: (Monad m, Applicative m) => (Expr -> m Expr) -> Expr -> m Expr
descendM f e = case e of
  EApp  a b   -> EApp <$> descendM f a <*> descendM f b
  EVar  a     -> EVar <$> pure a
  ELam  a b   -> ELam <$> pure a <*> descendM f b
  ELit  n     -> ELit <$> pure n
  ELet  n a b -> ELet <$> pure n <*> descendM f a <*> descendM f b
  EIf a b c   -> EIf <$> descendM f a <*> descendM f b <*> descendM f c
  ECase a xs  -> ECase <$> f a <*> traverse (descendCaseM f) xs
  EAnn a t    -> EAnn <$> descendM f a <*> pure t
  EFail       -> pure EFail

descendCaseM :: (Monad m, Applicative m) => (Expr -> m Expr) -> Match -> m Match
descendCaseM f e = case e of
  Match ps a -> Match <$> pure ps <*> descendM f a
```

The case where the monad is Identity simply corresponds to a pure expression rewrite.

```
descend :: (Expr -> Expr) -> Expr -> Expr
descend f ex = runIdentity (descendM (return . f) ex)
```

For example a transformation for use in this framework of traversals might just use pattern matching to match specific syntactic structure and rewrite it or simply yield the input and traverse to the next element in the bottom-up traversal.

A pure transformation that rewrites all variables named “a” to “b” might be written concisely as the following higher order function.

```
transform :: Expr -> Expr
transform = descend f
  where
    f (Syn.EVar ”a”) = (Syn.EVar ”b”)
    f x = x
```

This is good for pure logic, but most often our transformations will have to have access to some sort of state or context during traversal and thus `descendM` will let us write the same rewrite but in a custom monadic context.

```
transform :: Expr -> RewriteM Expr
transform = descendM f
  where
    f (Syn.EVar x) = do
      env <- gets _env
      return $ Syn.EVar (lookupVar env x)
    f x = x
```

These traversals admit very nice composition semantics, and AST transformations can be composed and chained as easily as functions.

```
compose
  :: (Expr -> Expr)
  -> (Expr -> Expr)
  -> (Expr -> Expr)
compose f g = descend (f . g)
```

Recall that monadic actions can be composed like functions using the Kleisli composition operator.

```
Functions          : a -> b
Monadic operations : a -> m b

-- Function composition
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
f . g = \x -> g (f x)

-- Monad composition
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
f <=< g = \x -> g x >>= f
```

We can now write composition `descendM` functions in terms of Kleisli composition to give us a very general notion of AST rewrite composition.

```
composeM
  :: (Applicative m, Monad m)
  => (Expr -> m Expr)
  -> (Expr -> m Expr)
  -> (Expr -> m Expr)
composeM f g = descendM (f <=< g)
```

So for instance if we have three AST monadic transformations (`a`, `b`, `c`) that we wish to compose into a single pass `t` we can use `composeM` to generate the composite transformation.

```
a :: Expr -> RewriteM Expr
b :: Expr -> RewriteM Expr
c :: Expr -> RewriteM Expr

t :: Expr -> RewriteM Expr
t =  a `composeM` b `composeM` c
```

Later we utilize both `GHC.Generics` and `Uniplate` to generalize this technique even more.

Misc Infrastructure

Repline

Command Line Arguments

GraphSCC

Optparse Applicative

Full Source

The partial source for the Frontend of ProtoHaskell is given. This is a stub of all the data structures and scaffolding we will use to construct the compiler pipeline.

- ProtoHaskell Frontend

The modules given are:

- `Monad.hs` - Compiler monad
- `Flags.hs` - Compiler flags
- `Frontend.hs` - Frontend syntax
- `Name.hs` - Syntax names
- `Compiler.hs` - Initial compiler stub
- `Pretty.hs` - Pretty printer
- `Type.hs` - Type syntax

Resources

See:

- GHC Commentary
- The Architecture of Open Source Applications: GHC

Extended Parser

Extended Parser

Up until now we've been using parser combinators to build our parsers. Parser combinators build top-down parsers that formally belong to the `LL`(k) family of parsers. The parser proceeds top-down, with a sequence of k characters used to dispatch on the leftmost production rule. Combined with backtracking (i.e. the `try` combinator) this is simultaneously both an extremely powerful and simple model to implement as we saw before with our simple 100 line parser library.

However there is a family of grammars that include left-recursion that `LL`(k) can be inefficient and often incapable of parsing. Left-recursive rules are such where the left-most symbol of the rule recurses on itself. For example:

$$e ::= e \texttt{ op atom}$$

Now we demonstrated before that we could handle these cases using the parser combinator `chainl1`, and while this is possible sometimes it can in many cases be an inefficient use of the parser stack and lead to ambiguous cases.

The other major family of parsers, `LR`, are not plagued with the same concerns over left recursion. On the other hand `LR` parser are exceedingly more complicated to implement, relying on a rather sophisticated method known as Tomita's algorithm to do the heavy lifting. The tooling around the construction of the *production rules* in a form that can be handled by the algorithm is often handled by a DSL that generates the code for the parser. While the tooling is fairly robust, there is a level of indirection between us and the code that can often be a bit of brittle to extend with custom logic.

The most common form of this toolchain is the Lex/Yacc lexer and parser generator which compile into efficient C parsers for `LR` grammars. Haskell's **Happy** and **Alex** are roughly the Haskell equivalent of these tools.

Toolchain

Our parser and lexer logic will be spread across two different modules.

- Lexer.x
- Parser.y

The code in each of these modules is a hybrid of the specific Alex/Happy grammar syntax and arbitrary Haskell logic that is spliced in. Code delineated by braces (`{}`) is regular Haskell, while code outside is parsera and lexer logic.

```
-- **Begin Haskell Syntax**
{
{-# OPTIONS_GHC -w #-}

module Lexer (
  Token(..),
  scanTokens
) where

import Syntax
}
-- **End Haskell Syntax**

-- **Begin Alex Syntax**
%wrapper "basic"

$digit = 0-9
$alpha = [a-zA-Z]
$eol   = [\n]
-- **End Alex Syntax**
```

The files will be used during the code generation of the two modules `Lexer` and `Parser`. The toolchain is accessible in several ways, first via the command-line tools `alex` and `happy` which will generate the resulting modules by passing the appropriate input file to the tool.

```
$ alex Lexer.x    # Generates Lexer.hs
$ happy Parser.y  # Generates Parser.hs
```

Or inside of the cabal file using the `build-tools` command.

```
  Build-depends:       base, array
  build-tools:         alex, happy
  other-modules:
    Parser,
    Lexer
```

So the resulting structure of our interpreter will have the following set of files.

- **Lexer.hs**
- **Parser.hs**
- Eval.hs
- Main.hs
- Syntax.hs

Alex

Our lexer module will export our Token definition and a function for converting an arbitrary String into a *token stream* or a list of Tokens.

```
{
module Lexer (
  Token(..),
  scanTokens
) where

import Syntax
}
```

The tokens are simply an enumeration of the unique possible tokens in our grammar.

```
data Token
  = TokenLet
  | TokenTrue
  | TokenFalse
  | TokenIn
  | TokenLambda
  | TokenNum Int
  | TokenSym String
  | TokenArrow
  | TokenEq
  | TokenAdd
  | TokenSub
  | TokenMul
  | TokenLParen
  | TokenRParen
  | TokenEOF
  deriving (Eq,Show)

scanTokens :: String -> [Token]
scanTokens = alexScanTokens
```

The token definition is a list of function definitions mapping atomic characters and alphabetical sequences to constructors for our `Token` datatype.

```
%wrapper "basic"

$digit = 0-9
$alpha = [a-zA-Z]
$eol   = [\n]
```

```
tokens :-

  -- Whitespace insensitive
  $eol                          ;
  $white+                       ;

  -- Comments
  "#".*                         ;

  -- Syntax
  let                           { \s -> TokenLet }
  True                          { \s -> TokenTrue }
  False                         { \s -> TokenFalse }
  in                            { \s -> TokenIn }
  $digit+                       { \s -> TokenNum (read s) }
  "->"                          { \s -> TokenArrow }
  \=                            { \s -> TokenEq }
  \\                            { \s -> TokenLambda }
  [\+]                          { \s -> TokenAdd }
  [\-]                          { \s -> TokenSub }
  [\*]                          { \s -> TokenMul }
  \(                            { \s -> TokenLParen }
  \)                            { \s -> TokenRParen }
  $alpha [$alpha $digit \_ \']* { \s -> TokenSym s }
```

Happy

Using Happy and our previosly defind lexer we'll write down the production rules for our simple untyped lambda calculus.

We start by defining a `Syntax` module where we define the AST we'll generate from running over the token stream to produce the program graph structure.

```
module Syntax where

type Name = String

data Expr
  = Lam Name Expr
  | App Expr Expr
  | Var Name
  | Lit Lit
  | Op Binop Expr Expr
  deriving (Eq,Show)
```

```
data Lit
  = LInt Int
  | LBool Bool
  deriving (Show, Eq, Ord)

data Binop = Add | Sub | Mul | Eql
  deriving (Eq, Ord, Show)
```

The token constructors are each assigned to a name that will be used in our production rules.

```
-- Lexer structure
%tokentype { Token }

-- Token Names
%token
    let   { TokenLet }
    true  { TokenTrue }
    false { TokenFalse }
    in    { TokenIn }
    NUM   { TokenNum $$ }
    VAR   { TokenSym $$ }
    '\\'  { TokenLambda }
    '->'  { TokenArrow }
    '='   { TokenEq }
    '+'   { TokenAdd }
    '-'   { TokenSub }
    '*'   { TokenMul }
    '('   { TokenLParen }
    ')'   { TokenRParen }
```

The parser itself will live inside of a custom monad of our choosing. In this case we'll add error handling with the `Except` monad that will break out of the parsing process if an invalid production or token is found and return a `Left` value which we'll handle inside of our toplevel logic.

```
-- Parser monad
%monad { Except String } { (>>=) } { return }
%error { parseError }
```

And finally our production rules, the toplevel entry point for our parser will be the `expr` rule. The left hand side of the production is a Happy production rule which can be mutually recursive, while the right hand side is a Haskell expression with several metavariable indicated by the dollar sign variables that map to the nth expression on the left hand side.

```
$0  $1  $2  $3   $4 $5
let VAR '=' Expr in Expr    { App (Lam $2 $6) $4 }
```

```
-- Entry point
%name expr

-- Operators
%left '+' '-'
%left '*'
%%

Expr : let VAR '=' Expr in Expr    { App (Lam $2 $6) $4 }
     | '\\' VAR '->' Expr          { Lam $2 $4 }
     | Form                        { $1 }

Form : Form '+' Form               { Op Add $1 $3 }
     | Form '-' Form               { Op Sub $1 $3 }
     | Form '*' Form               { Op Mul $1 $3 }
     | Fact                        { $1 }

Fact : Fact Atom                   { App $1 $2 }
     | Atom                        { $1 }

Atom : '(' Expr ')'                { $2 }
     | NUM                         { Lit (LInt $1) }
     | VAR                         { Var $1 }
     | true                        { Lit (LBool True) }
     | false                       { Lit (LBool False) }
```

Notice how naturally we can write a left recursive grammar for our binary infix operators.

Syntax Errors

Parsec's default error reporting leaves a bit to be desired, but does in fact contain most of the information needed to deliver better messages packed inside the ParseError structure.

```
showSyntaxError :: L.Text -> ParseError -> String
showSyntaxError s err = L.unpack $ L.unlines [
      "  ",
      "  " <> lineContents,
      "  " <> ((L.replicate col " ") <> "^"),
      (L.pack $ show err)
    ]
  where
    lineContents = (L.lines s) !! line
    pos  = errorPos err
    line = sourceLine pos - 1
    col  = fromIntegral $ sourceColumn pos - 1
```

Now when we enter an invalid expression the error reporting will point us directly to the adjacent lexeme that caused the problem as is common in many languages.

```
λ> \x -> x +

  \x -> x +
            ^
"<interactive>" (line 1, column 11):
unexpected end of input
expecting "(", character, literal string, "[", integer, "if" or identifier
```

Type Error Provenance

Before our type inference engine would generate somewhat typical type inference error messages. If two terms couldn't be unified it simply told us this and some information about the top-level declaration where it occurred, leaving us with a bit of a riddle about how exactly this error came to be.

```
Cannot unify types:
        Int
with
        Bool
in the definition of 'foo'
```

Effective error reporting in the presence of type inference is a difficult task, effectively our typechecker takes our frontend AST and transforms it into a large constraint problem, destroying position information in the process. Even if the position information were tracked, the nature of unification is that a cascade of several unifications can lead to unsolvability and the immediate two syntactic constructs that gave rise to a unification failure are not necessarily the two that map back to human intuition about how the type error arose. Very little research has done on this topic and it remains an open topic with very immediate and applicable results to programming.

To do simple provenance tracking we will use a technique of tracking the "flow" of type information through our typechecker and associate position information with the inferred types.

```
type Name = String

data Expr
  = Var Loc Name
  | App Loc Expr Expr
  | Lam Loc Name Expr
  | Lit Loc Int

data Loc = NoLoc | Located Int
  deriving (Show, Eq, Ord)
```

So now inside of our parser we simply attach Parsec information on to each AST node. For example for the variable term.

```
variable :: Parser Expr
variable = do
  x <- identifier
  l <- sourceLine <$> getPosition
  return (Var (Located l) x)
```

Our type system will also include information, although by default it will use the `NoLoc` value until explicit information is provided during inference. The two functions `getLoc` and `setLoc` will be used to update and query the position information from type terms.

```
data Type
  = TVar Loc TVar
  | TCon Loc Name
  | TArr Loc Type Type
  deriving (Show, Eq, Ord)

newtype TVar = TV String
  deriving (Show, Eq, Ord)

typeInt :: Type
typeInt = TCon NoLoc "Int"

setLoc :: Loc -> Type -> Type
setLoc l (TVar _ a)   = TVar l a
setLoc l (TCon _ a)   = TCon l a
setLoc l (TArr _ a b) = TArr l a b

getLoc :: Type -> Loc
getLoc (TVar l _) = l
getLoc (TCon l _) = l
getLoc (TArr l _ _) = l
```

Our fresh variable supply now also takes a location field which is attached to the resulting type variable.

```
fresh :: Loc -> Check Type
fresh l = do
  s <- get
  put s{count = count s + 1}
  return $ TVar l (TV (letters !! count s))

infer :: Expr -> Check Type
infer expr = case expr of
```

```
  Var l n -> do
    t <- lookupVar n
    return $ setLoc l t

  App l a b -> do
    ta <- infer a
    tb <- infer b
    tr <- fresh l
    unify ta (TArr l tb tr)
    return tr

  Lam l n a -> do
    tv <- fresh l
    ty <- inEnv (n, tv) (infer a)
    return (TArr l (setLoc l ty) tv)

  Lit l _ -> return (setLoc l typeInt)
```

Now specifically at the call site of our unification solver, if we encounter a unification fail we simply pluck the location information off the two type terms and plug it into the type error fields.

```
unifies t1 t2 | t1 == t2 = return emptyUnifer
unifies (TVar _ v) t = v `bind` t
unifies t (TVar _ v) = v `bind` t
unifies (TArr _ t1 t2) (TArr _ t3 t4) = unifyMany [t1, t2] [t3, t4]
unifies (TCon _ a) (TCon _ b) | a == b = return emptyUnifer
unifies t1 t2 = throwError $ UnificationFail t1 (getLoc t1) t2 (getLoc t2)

bind ::  TVar -> Type -> Solve Unifier
bind a t
  | eqLoc t a        = return (emptySubst, [])
  | occursCheck a t  = throwError $ InfiniteType a (getLoc t) t
  | otherwise        = return $ (Subst $ Map.singleton a t, [])
```

So now we can explicitly trace the provenance of the specific constraints that gave rise to a given type error all the way back to the source that generated them.

```
Cannot unify types:
        Int
        Introduced at line 27 column 5

          f 2 3

with
        Int -> c
```

```
        Introduced at line 5 column 9

          let f x y = x y
```

This is of course the simplest implementation of the tracking method and could be further extended by giving a weighted ordering to the constraints based on their likelihood of importance and proximity and then choosing which location to report based on this information. This remains an open area of work.

Indentation

Haskell's syntax uses indentation blocks to delineated sections of code. This use of indentation sensitive layout to convey the structure of logic is sometimes called the *offside rule* in parsing literature. At the beginning of a "laidout" block the first declaration or definition can start in any column, and the parser marks that indentation level. Every subsequent declaration at the same logical level must have the same indentation.

```
-- Start of layout ( Column: 0 )
fib :: Int -> Int
fib x = truncate $ ( 1 / sqrt 5 ) * ( phi ^ x - psi ^ x ) -- (Column: > 0)
  -- Start of new layout ( Column: 2 )
  where
      -- Indented block ( Column: > 2 )
      phi = ( 1 + sqrt 5 ) / 2
      psi = ( 1 - sqrt 5 ) / 2
```

The Parsec monad is parameterized over a type which stands for the State layer baked into the monad allowing us to embed custom parser state inside of our rules. To adopt our parser to handle sensitive whitespace we will use:

```
-- Indentation sensitive Parsec monad.
type IParsec a = Parsec Text ParseState a

data ParseState = ParseState
  { indents :: Column
  } deriving (Show)

initParseState :: ParseState
initParseState = ParseState 0
```

The parser stores the internal position state (SourcePos) during its traversal, and makes it accessible inside of rule logic via the `getPosition` function.

```
data SourcePos = SourcePos SourceName !Line !Column
getPosition :: Monad m => ParsecT s u m SourcePos
```

In terms of this function we can write down a set of logic that will allow us to query the current column count and then either succeed or fail to match on a pattern based on the current indentation level. The `laidout` combinator will capture the current indentation state and push it into the `indents` field in the State monad.

```
laidout :: Parsec s ParseState a -> Parsec s ParseState a
laidout m = do
  cur <- indents <$> getState
  pos <- sourceColumn <$> getPosition
  modifyState $ \st -> st { indents = pos }
  res <- m
  modifyState $ \st -> st { indents = cur }
  return res
```

And then have specific logic which guard the parser match based on comparing the current indentation level to the stored indentation level.

```
indentCmp
  :: (Column -> Column -> Bool)
  -> Parsec s ParseState ()
indentCmp cmp = do
  col <- sourceColumn <$> getPosition
  current <- indents <$> getState
  guard (col `cmp` current)
```

We can then write two combinators in terms of this function which match on either further or identical indentation.

```
indented :: IParsec ()
indented = indentCmp (>) <?> "Block (indented)"

align :: IParsec ()
align = indentCmp (==) <?> "Block (same indentation)"
```

On top of these we write our two combinators for handling block syntax, which match a sequence of vertically aligned patterns as a list.

```
block, block1 :: Parser a -> Parser [a]
block  p = laidout (many (align >> p))
block1 p = laidout (many1 (align >> p))
```

Haskell uses an optional layout rule for several constructs, allowing us to equivalently manually delimit indentation sensitive syntax with braces. The most common use is for do-notation. So for example:

```haskell
example = do { a <- m; b }

example = do
  a <- m
  b
```

To support this in Parsec style we implement a `maybeBraces` function.

```haskell
maybeBraces :: Parser a -> Parser [a]
maybeBraces p = braces (endBy p semi) <|> block p

maybeBraces1 :: Parser a -> Parser [a]
maybeBraces1 p = braces (endBy1 p semi) <|> block p
```

Extensible Operators

Haskell famously allows the definition of custom infix operators, an extremely useful language feature although this poses a bit of a challenge to parse! There are two ways to do this and both depend on two properties of the operators.

- Precedence
- Associativity

1. The first, the way that GHC does it, is to parse all operators as left associative and of the same precedence, and then before desugaring go back and "fix" the parse tree given all the information we collected after finishing parsing.
2. The second method is a bit of a hack, and involves simply storing the collected operators inside of the Parsec state monad and then simply calling `buildExpressionParser` on the current state each time we want to parse an infix operator expression.

To do the later method we set up the AST objects for our fixity definitions, which associate precedence and associativity annotations with a custom symbol.

```haskell
data FixitySpec = FixitySpec
  { fixityFix :: Fixity
  , fixityName :: String
  } deriving (Eq, Show)

data Assoc
  = L
  | R
  | N
  deriving (Eq,Ord,Show)
```

```haskell
data Fixity
  = Infix Assoc Int
  | Prefix Int
  | Postfix Int
  deriving (Eq,Ord,Show)
```

Our parser state monad will hold a list of the active fixity specifications and whenever a definition is encountered we will append to this list.

```haskell
data ParseState = ParseState
  { indents :: Column
  , fixities :: [FixitySpec]
  } deriving (Show)

initParseState :: ParseState
initParseState = ParseState 0 defaultOps

addOperator ::  FixitySpec -> Parsec s ParseState ()
addOperator fixdecl = do
  modifyState $ \st -> st { fixities = fixdecl : (fixities st) }
```

The initial state will consist of the default arithmetic and list operators defined with the same specification as the Haskell specification.

```haskell
defaultOps :: [FixitySpec]
defaultOps = [
    FixitySpec (Infix L 4) ">"
  , FixitySpec (Infix L 4) "<"
  , FixitySpec (Infix L 4) "/="
  , FixitySpec (Infix L 4) "=="

  , FixitySpec (Infix R 5) ":"

  , FixitySpec (Infix L 6) "+"
  , FixitySpec (Infix L 6) "-"

  , FixitySpec (Infix L 5) "*"
  , FixitySpec (Infix L 5) "/"
  ]
```

Now in our parser we need to be able to transform the fixity specifications into Parsec operator definitions. This is a pretty straightforward sort and group operation on the list.

```haskell
fixityPrec :: FixitySpec -> Int
fixityPrec (FixitySpec (Infix _ n) _) = n
```

```
fixityPrec (FixitySpec _ _) = 0

toParser (FixitySpec ass tok) = case ass of
    Infix L _ -> infixOp tok (op (Name tok)) Ex.AssocLeft
    Infix R _ -> infixOp tok (op (Name tok)) Ex.AssocRight
    Infix N _ -> infixOp tok (op (Name tok)) Ex.AssocNone

mkTable ops =
  map (map toParser) $
    groupBy ((==) `on` fixityPrec) $
      reverse $ sortBy (compare `on` fixityPrec) $ ops
```

Now when parsing an infix operator declaration we simply do a state operation and add the operator to the parser state so that all subsequent definitions can use it. This differs from Haskell slightly in that operators must be defined before their usage in a module.

```
fixityspec :: Parser FixitySpec
fixityspec = do
  fix  <- fixity
  prec <- precedence
  op   <- operator
  semi
  let spec = FixitySpec (fix prec) op
  addOperator spec
  return spec
  where
    fixity = Infix L <$ reserved "infixl"
         <|> Infix R <$ reserved "infixr"
         <|> Infix N <$ reserved "infix"

precedence :: Parser Int
precedence = do
  n <- natural
  if n <= 10
  then return (fromInteger n)
  else empty
  <?> "Invalid operator precedence"

fixitydecl :: Parser Decl
fixitydecl = do
  spec <- fixityspec
  return $ FixityDecl spec
 <?> "operator fixity definition"
```

And now when we need to parse an infix expression term we simply pull our state out and build the custom operator table, and feed this to the build Expression Parser just as before.

```haskell
term :: Parser Expr -> Parser Expr
term a = do
  st <- getState
  let customOps = mkTable (fixities st)
  Ex.buildExpressionParser customOps a
```

Full Source

- Happy Parser
- Imperative Language (Happy)
- Layout Combinators
- Type Provenance Tracking

Resources

The tooling and documentation for Alex and Happy is well-developed as it is used extensively inside of GHC:

- Alex User Guide
- Happy User Guide
- A Tool for Generalized LR Parsing In Haskell
- Haskell Syntax Definition

GHC itself uses Alex and Happy for its parser infastructure. The resulting parser is rather sophisicated.

- Lexer.x
- Parser.y

One of the few papers ever written in Type Error reporting gives some techniques for presentation and tracing provenance:

- Top Quality Type Error Messages

Datatypes

Algebraic data types

Algebraic datatypes are a family of constructions arising out of two operations, *products* (`a * b`) and *sums* (`a + b`) (sometimes also called *coproducts*). A product encodes multiple arguments to constructors and sums encode choice between constructors.

```
{-# LANGUAGE TypeOperators #-}

data Unit = Unit                  -- 1
data Empty                        -- 0
data (a * b) = Product a b        -- a * b
data (a + b) = Inl a | Inr b      -- a + b
data Exp a b = Exp (a -> b)       -- a^b
data Rec f   = Rec (f (Rec f))    -- \mu
```

The two constructors `Inl` and `Inr` are the left and right *injections* for the sum. These allows us to construct sums.

```
Inl :: a -> a + b
Inr :: b -> a + b
```

Likewise for the product there are two function `fst` and `snd` which are *projections* which de construct products.

```
fst :: a * b -> a
snd :: a * b -> b
```

Once a language is endowed with the capacity to write a single product or a single sum, all higher order products can written in terms of sums of products. For example a 3-tuple can be written in terms of the composite of two 2-tuples. And indeed any n-tuple or record type can be written in terms of compositions of products.

```
type Prod3 a b c = a*(b*c)

data Prod3' a b c
  = Prod3 a b c

prod3 :: Prod3 Int Int Int
prod3 = Product 1 (Product 2 3)
```

Or a sum type of three options can be written in terms of two sums:

```haskell
type Sum3 a b c = (a+b)+c

data Sum3' a b c
  = Opt1 a
  | Opt2 b
  | Opt3 c

sum3 :: Sum3 Int Int Int
sum3 = Inl (Inl 2)

data Option a = None | Some a

type Option' a = Unit + a

some :: Unit + a
some = Inl Unit

none :: a -> Unit + a
none a = Inr a
```

In Haskell the convention for the sum and product notation is as follows:

Notation	Haskell Type
`a * b`	`(a,b)`
`a + b`	`Either a b`
`Inl`	`Left`
`Inr`	`Right`
`Empty`	`Void`
`Unit`	`()`

Isorecursive Types

$$\texttt{Nat} = \mu\alpha.1 + \alpha$$

```haskell
roll :: Rec f -> f (Rec f)
roll (Rec f) = f

unroll :: f (Rec f) -> Rec f
unroll f = Rec f
```

Peano numbers:

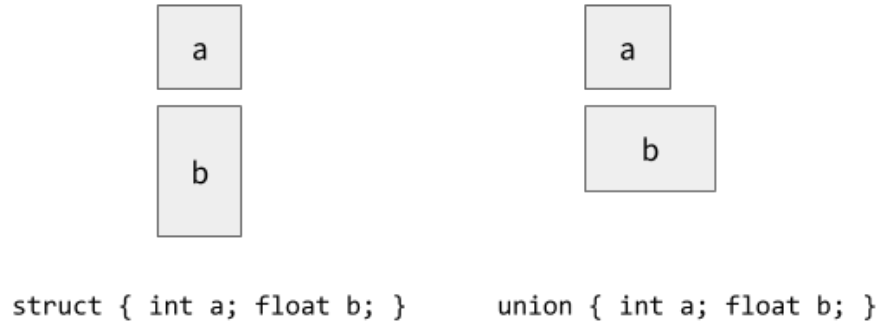
```haskell
type Nat = Rec NatF
data NatF s = Zero | Succ s

zero :: Nat
zero = Rec Zero

succ :: Nat -> Nat
succ x = Rec (Succ x)
```

Lists:

```haskell
type List a = Rec (ListF a)
data ListF a b = Nil | Cons a b

nil :: List a
nil = Rec Nil

cons :: a -> List a -> List a
cons x y = Rec (Cons x y)
```

Memory Layout

Just as the type-level representation




```haskell
typedef union {
    int a;
```

```
    float b;
} Sum;

typedef struct {
    int a;
    float b;
} Prod;

int main()
{
    Prod x = { .a = 1, .b = 2.0 };
    Sum sum1 = { .a = 1 };
    Sum sum2 = { .b = 2.0 };
}
```

```
#include <stddef.h>

typedef struct T
{
  enum { NONE, SOME } tag;
  union
  {
     void *none;
     int some;
  } value;
} Option;

int main()
{
    Option a = { .tag = NONE, .value = { .none = NULL } };
    Option b = { .tag = SOME, .value = { .some = 3 } };
}
```

- Algebraic Datatypes in C (Part 1)
- Algebraic Datatypes in C (Part 2)

Pattern Scrutiny

In Haskell:

```
data T
  = Add T T
  | Mul T T
  | Div T T
```

```
  | Sub T T
  | Num Int

eval :: T -> Int
eval x = case x of
  Add a b -> eval a + eval b
  Mul a b -> eval a + eval b
  Div a b -> eval a + eval b
  Sub a b -> eval a + eval b
  Num a   -> a
```

In C:

```
typedef struct T {
    enum { ADD, MUL, DIV, SUB, NUM } tag;
    union {
        struct {
            struct T *left, *right;
        } node;
        int value;
    };
} Expr;

int eval(Expr t)
{
    switch (t.tag) {
        case ADD:
            return eval(*t.node.left) + eval(*t.node.right);
            break;
        case MUL:
            return eval(*t.node.left) * eval(*t.node.right);
            break;
        case DIV:
            return eval(*t.node.left) / eval(*t.node.right);
            break;
        case SUB:
            return eval(*t.node.left) - eval(*t.node.right);
            break;
        case NUM:
            return t.value;
            break;
    }
}
```

- Pattern Matching

Syntax

GHC.Generics

```
class Generic a where
  type family Rep a :: * -> *
  to   :: a -> Rep a x
  from :: Rep a x -> a
```

Constructor	Models
`V1`	Void: used for datatypes without constructors
`U1`	Unit: used for constructors without arguments
`K1`	Constants, additional parameters.
`:*:`	Products: encode multiple arguments to constructors
`:+:`	Sums: encode choice between constructors
`L1`	Left hand side of a sum.
`R1`	Right hand side of a sum.
`M1`	Meta-information (constructor names, etc.)

```
newtype M1 i c f p = M1 (f p)
newtype K1 i c   p = K1 c
data U           p = U

data (:*:) a b p = a p :*: b p
data (:+:) a b p = L1 (a p) | R1 (b p)
```

Implementation:

```
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE DefaultSignatures #-}

import GHC.Generics

-- Auxiliary class
class GEq' f where
  geq' :: f a -> f a -> Bool

instance GEq' U1 where
  geq' _ _ = True

instance (GEq c) => GEq' (K1 i c) where
  geq' (K1 a) (K1 b) = geq a b
```

```haskell
instance (GEq' a) => GEq' (M1 i c a) where
  geq' (M1 a) (M1 b) = geq' a b

instance (GEq' a, GEq' b) => GEq' (a :+: b) where
  geq' (L1 a) (L1 b) = geq' a b
  geq' (R1 a) (R1 b) = geq' a b
  geq' _      _      = False

instance (GEq' a, GEq' b) => GEq' (a :*: b) where
  geq' (a1 :*: b1) (a2 :*: b2) = geq' a1 a2 && geq' b1 b2

--
class GEq a where
  geq :: a -> a -> Bool
  default geq :: (Generic a, GEq' (Rep a)) => a -> a -> Bool
  geq x y = geq' (from x) (from y)

-- Base equalities
instance GEq Char where geq = (==)
instance GEq Int where geq = (==)
instance GEq Float where geq = (==)

-- Equalities derived from structure of (:+:) and (:*:) for free
instance GEq a => GEq (Maybe a)
instance (GEq a, GEq b) => GEq (a,b)

main :: IO ()
main = do
  print $ geq 2 (3 :: Int)           -- False
  print $ geq 'a' 'b'                -- False
  print $ geq (Just 'a') (Just 'a')  -- True
  print $ geq ('a','b') ('a', 'b')   -- True
```

- Generics

Wadler's Algorithm

Consider the task of expanding

```haskell
f :: Just (Either a b) -> Int
f (Just (Left x))  = 1
f (Just (Right x)) = 2
f Nothing          = 3
```

```
f :: Maybe (Either a b) -> Int
f = case ds of _ {
      Nothing -> I# 3;
      Just ds1 ->
        case ds1 of _ {
          Left x -> I# 1;
          Right x -> I# 2
        }
    }
```

Full Source

Renamer

Renaming Pass

Full Source

LLVM

LLVM

JIT Compilation

Just-in-time or JIT compilation is compilation done by dynamically generating executable code. It's a common technique used in many language runtimes to generate optimized code for hot code paths as well ahead of time compilation for various tasks.

So let's build a small LLVM-like intermediate language and JIT execution engine in Haskell. This will only function with modern Linux and x86-64 architecture, although in principle this will work on any platform if you can make the appropriate FFI calls to `mmap` and `mprotect` syscalls on your respective platform.

Source code is available here.

Types The x86-x64 instruction set is the 64-bit revision of x86 instruction set which was first developed for the Intel 8086 CPU family. The base types which hardware operates over are integers and floating point types. Let us just consider the integral types for now, these come in four major varieties:

On the Intel architecture numbers are represented *little endian* meaning lower significant bytes are stored in lower memory addresses. The whole memory representation for a value is partitioned into *high bits* and *low bits*. For example the hexadecimal number `0xc0ffee` as a DWORD is stored in memory as:

In Haskell unboxed integral machine types are provided by the `Data.Word` module.

```
data Word8  = W8# Word#
data Word16 = W16# Word#
data Word32 = W32# Word#
data Word64 = W64# Word#
```

Pointers are simply literal addresses to main memory, where the underlying access and storage are managed by the Linux kernel. To model this abstractly in Haskell we'll create a datatype containing the possible values we can operate over.

```
data Val
  = I Int64       -- Integer
  | R Reg         -- Register
  | A Word32      -- Addr
  deriving (Eq, Show)
```

To convert from Haskell types into byte sequences we'll use the binary library to convert integral types into little endian arrays of bytes.

```
bytes :: Integral a => a -> [Word8]
bytes x = fmap BS.c2w bs
  where
    bs = unpack $ runPut $ putWord32le (fromIntegral x)
```

For example given a hexadecimal literal this will expand it out into an array of it's bit components.

```
val = bytes 0xc0ffee -- [238,255,192,0]
```

Registers The x64 architecture contains sixteen general purpose 64-bit registers capable of storing a quadword. They major ones are labeled `rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsi`, `rdi`, and `rsp`.

Each of the registers is given a specific index (`r`), which will be used in the binary representation of specific instructions that operate over these registers.

	RAX	RBX	RCX	RDX	RBP	RSI	RDI	RSP
r =	0	1	2	3	4	5	6	7

Each of these registers can be addressed as a smaller register containing a subset of the lower bits. The 32-bit register of `rax` is `eax`. These are shown in the table below.

These smaller registers are given specific names with modified prefixes.

64-bit	32-bit	16-bit
rax	eax	ax
rbx	ebx	bx
rcx	ecx	cx
rdx	edx	dx
rsp	esp	sp
rbp	ebp	bp
rsi	esi	si
rdi	edi	di
rip	eip	ip

In Haskell we model this a sum type for each of the 64-bit registers. Consider just the 64-bit registers for now.

```
data Reg
  = RAX  -- Accumulator
```

```
  | RCX  -- Counter (Loop counters)
  | RDX  -- Data
  | RBX  -- Base / General Purpose
  | RSP  -- Current stack pointer
  | RBP  -- Previous Stack Frame Link
  | RSI  -- Source Index Pointer
  | RDI  -- Destination Index Pointer
  deriving (Eq, Show)
```

The index for each register is defined by a simple pattern match case expression.

```
index :: Reg -> Word8
index x = case x of
  RAX -> 0
  RCX -> 1
  RDX -> 2
  RBX -> 3
  RSP -> 4
  RBP -> 5
  RSI -> 6
  RDI -> 7
```

Monads Monads are an algebraic structure with two functions (`bind`) and (`return`) and three laws.

```
bind :: Monad m => m a -> (a -> m b) -> m b
return :: Monad m => a -> m a
```

The compiler will desugar do-blocks of the form into a canonical form involving generic bind and return statements.

```
f :: Monad m => m Int
f = do
  a <- x
  b <- y
  return (a+b)
```

Is transformed into:

```
f :: Monad m => m Int
f =
  bind x (\a ->
    bind y (\b ->
      return (a+b))
```

Monad is implemented as a typeclass indexed by a parameter `m`, that when instantiated with a typeclass instances replaces the bind and return functions with a specific implementation of the two functions (like State or Reader).

```
f :: State MyState Int
f =
  bindState x (\a ->
    bindState y (\b ->
      returnState (a+b))
```

The State monad is an instance of Monad with several functions for composing stateful logic.

```
get :: State s s                    -- get the state
put :: s -> State s ()              -- set the state
modify :: (s -> s) -> State s ()    -- apply a function over the state
```

For example a little state machine that holds a single Int value would be written like the following.

```
machine :: State Int Int
machine = do
  put 3
  modify (+1)
  get

val :: Int
val = execState machine 0
```

More common would be to have the state variable `s` be a record with multiple fields that can be modified. For managing our JIT memory we'll create a struct with the several fields.

```
data JITMem = JITMem
 { _instrs :: [Instr]
 , _mach   :: [Word8]
 , _icount :: Word32
 , _memptr :: Word32
 , _memoff :: Word32
 } deriving (Eq, Show)
```

This will be composed into our X86 monad which will hold the JIT memory as we assemble individual machine instructions and the pointer and memory offsets for the sequence of assembled instructions.

```
type X86 a = StateT JITMem (Except String) a
```

JIT Memory To start creating the JIT we first need to create a block of memory with executable permissions. Inside of C runtime we can get the flags needed to be passed to the various `mmap` syscall to create the necessary memory block.

```
#define PROT_NONE       0x00    /* No access.  */
#define PROT_READ       0x04    /* pages can be read */
#define PROT_WRITE      0x02    /* pages can be written */
#define PROT_EXEC       0x01    /* pages can be executed */

#define MAP_FILE        0x0001  /* mapped from a file or device */
#define MAP_ANON        0x0002  /* allocated from memory, swap space */
#define MAP_TYPE        0x000f  /* mask for type field */
```

Then we simply allocate a given block of memory off the Haskell heap via mmap with the executable flags.

```
newtype MmapOption = MmapOption CInt
  deriving (Eq, Show, Ord, Num, Bits)

protExec    = ProtOption 0x01
protWrite   = ProtOption 0x02
mmapAnon    = MmapOption 0x20
mmapPrivate = MmapOption 0x02

allocateMemory :: CSize -> IO (Ptr Word8)
allocateMemory size = mmap nullPtr size pflags mflags (-1) 0
  where
    pflags = protRead <> protWrite
    mflags = mapAnon .|. mapPrivate
```

Haskell pointers can be passed to our JIT'd code by simply casting them into their respective addresses on the Haskell heap.

```
heapPtr :: Ptr a -> Word32
heapPtr = fromIntegral . ptrToIntPtr
```

For example if we want allocate a null-terminated character array and pass a pointer to it's memory to our JIT'd code we can write down a `asciz` to synthesize this memory from a Haskell string and grab the heap pointer.

```
asciz :: [Char] -> IO Word32
asciz str = do
  ptr <- newCString (str ++ ['\n'])
  return $ heapPtr ptr
```

For C functions we simply use the dynamic linker to grab the function pointer the given symbol in the memory space of the process. The Haskell runtime links against glibc's `stdio.h` and `math.h` so these symbols will all be floating around in memory.

```
extern :: String -> IO Word32
extern name = do
  dl <- dlopen "" [RTLD_LAZY, RTLD_GLOBAL]
  fn <- dlsym dl name
  return $ heapPtr $ castFunPtrToPtr fn
```

When we've compiled our byte vector of machine code we'll copy into executable memory.

```
jit :: Ptr Word8 -> [Word8] -> IO (IO Int)
jit mem machCode = do
  code <- codePtr machCode
  withForeignPtr (vecPtr code) $ \ptr -> do
    copyBytes mem ptr (8*6)
  return (getFunction mem)
```

Then we'll use the FFI to synthesize a function pointer to the memory and invoke it.

```
foreign import ccall "dynamic"
  mkFun :: FunPtr (IO Int) -> IO Int

getFunction :: Ptr Word8 -> IO Int
getFunction mem = do
  let fptr = unsafeCoerce mem :: FunPtr (IO Int)
  mkFun fptr
```

Assembly Before we start manually populating our executable code with instructions, let's look at *assembly* form of what we'll write and create a small little DSL in Haskell make this process closer to the problem domain. Assembly is the intermediate human readable representation of machine code. Both clang and gcc are capable of dumping out this representation before compilation. For example for the following C program takes two integers passed in registers, multiplies them respectively and adds the result.

```
int f(int x, int y)
{
    return (x*x)^y;
}
```

Internally the C compiler is condensing the Destructuring the expressions into a linear sequence instructions storing the intermediate results in scratch registers and writing the end computed result to return register. It then selects appropriate machine instruction for each of the abstract operations.

```
// pseudocode for intermediate C representation
int f() {
    int x = register(rdi);
    int y = register(rsi);
    int tmp1 = x*x;
    int tmp2 = tmp1^y;
    return tmp2;
}
```

We can output the assembly to a file `add.s`. We'll use the Intel Syntax which puts the destination operand before other operands. The alternate AT&T syntax reverses this convention.

```
$ clang -O2 -S --x86-asm-syntax=intel xor.c
```

The generated code will resemble the following. Notice that there are two kinds of statements: *directives* and *instructions*. Directive are prefixed with a period while instructions are an operation together with a list operands. Statements of instructions are grouped into labeled blocks are suffixed with a colon for example `f:` is the label containing the logic for the function `f`.

```
        .file   "xor.c"
        .text
        .globl  f
        .type   f, @function
        .intel_syntax noprefix
f:
        mov     eax, edi
        imul    eax, edi
        xor     eax, esi
        ret
```

The assembler will then turn this sequence of instructions into either an executable or an object file containing the generated machine code. To disassemble the output we can use objdump to view the hex sequence of machine instructions and the offsets within the file.

```
$ objdump -M intel -d xor.o

xor:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <f>:
   0:   89 f8                   mov    eax,edi
   2:   0f af c7                imul   eax,edi
   5:   31 f0                   xor    eax,esi
   7:   c3                      ret
```

The compiled program in memory is then a contiguous array of bytes, which is evaluated by moving the instruction pointer at the start address.

```
89 f8 0f af c7 31 f0 c3
```

Instructions Instructions consist of two parts, an *opcode* and a set of *operands* which specify labels, registers, or addresses to memory which the CPU will execute over for the give instruction. We'll consider a subset of interesting operations which operate over integers and manipulate the call stack.

```
data Instr
  = Ret
  | Mov Val Val
  | Add Val Val
  | Sub Val Val
  | Mul Val
  | IMul Val Val
  | Xor Val Val
  | Inc Val
  | Dec Val
  | Push Val
  | Pop Val
  | Call Val
  | Loop Val
  | Nop
  | Syscall
  deriving (Eq, Show)
```

To add to the JIT memory we'll simply modify the state by appending an instruction to the `_mach` field and adjusting the memory offset pointer by the length of instructions added.

```
emit :: [Word8] -> X86 ()
emit i = modify $ \s -> s
  { _mach   = _mach s ++ i
  , _memoff = _memoff s + fromIntegral (length i)
  }
```

Operands Registers

Registers are identified as lowercase (i.e. `rbp`, `rsp`). In our expression builder we'll simply write down several functions which construct a register value type from an underlying Reg value.

```
rax :: Val
rax = R RAX

rsi :: Val
rsi = R RSI
```

Immediate Operands

Immediate operands are direct references to constants (literals or memory addresses) or labels. For example:

```
add     eax,42
add     eax,0xff
```

For immediate values we simply push the array of bytes for the number directly on the byte sequence.

```
imm :: Integral a => a -> X86 ()
imm = emit . bytes
```

Opcodes The full instruction set for x86 is vast and including AVX, SSE and other specialized intrinsics there is an extraordinary amount of complexity and quirky specifications. Each of these abstract instructions can have multiple opcodes for each type of operands it may take. For example the `mov` instruction for register to register movement has opcode `0x89` while moving immediate data into a register has opcode `0xC7`.

The reference for the most common operations the x86asm.net site has a very useful quick reference. For the full set of possible instructions on your modern Intel processor refer to the 1500 page Intel Software Developer's Manual.

To lookup the numerical opcodes for a given instructions, we use a specific naming conventions for the operands.

Prefix	Description
`r<size>`	Register Operand
`imm<size>`	Immediate Operand
`m<size>`	Memory Operand

So for example:

Prefix	Description
`r64`	64 bit register
`imm8`	8 immediate operand
`m32`	32 memory operand

For opcodes that operate over a set of possible operands, these are demarcated with a slash, in the form `r8/r16/r32`.

For our limited set of instructions there are two types of opcodes.

1. 1-Byte Opcodes

2. 2-Byte Opcodes

Instruction	Opcode
CALL	E8
RET	C3
NOP	0D
MOV	89
PUSH	50
POP	58
LOOP	E2
ADD	83
SUB	29
MUL	F7
DIV	F7
INC	FF
DEC	FF
NEG	F7
CMP	39
AND	21
OR	09
XOR	31
NOT	F7
ADC	11
IDIV	F7
IMUL	F7
XCHG	87
BSWAP	C8
SHR	C1
SHL	C1
ROR	C0
RCR	C0
BT	BA
BTS	BA
BTR	B3
JMP	EB
JE	84
JNE	85
SYSCALL	05

On top of this opcodes may have an additional prefixes which modify the sizes of arguments involved. These were added to allow 32-bit compatibility in the transition between 32-bit and 64-bit systems and preserve the underlying opcodes of the 32-bit system. For instance the following `mov` instructions all operate over registers and perform the same action but over different sizes.

```
mov al,255
mov ax,255
mov eax,255
mov rax,255
```

But translate into different opcodes depending on size.

```
b0 ff                   mov    al,0xff
66 b8 ff 00             mov    ax,0xff
b8 ff 00 00 00          mov    eax,0xff
48 c7 c0 ff 00 00 00    mov    rax,0xff
```

prefix	opcode	data	assembly	meaning
66	b0	ff	mov al, 0xff	8-bit load
66	b8	ff 00	mov ax, 0xff	load with a 16-bit prefix (0x66)
66	b8	ff 00 00 00	mov eax, 0xff	load with default size of 32 bits
48	c7 c0	ff 00 00 00	mov rax, 0xff	Sign-extended load using REX 64-bit prefix (0x48)

Machine Code Ok, let's look at the full structure of an instruction. It consists of several parts.

The sizes of these parts depend on the size and type of the opcode as well as prefix modifiers.

Prefix	Opcode	Mod R/M	Scale Index Base	Displacement	Immediate
1-4 bytes	1-3 bytes	1 Byte	1 Byte	1-4 Bytes	1-4 Bytes

Prefix

The header fixes the first four bits to be constant `0b0100` while the next four bits indicate the pretense of W/R/X/B extensions.

The W bit modifies the operation width. The R, X and B fields extend the register encodings.

- REX.W – Extends the operation width
- REX.R – Extends ModRM.reg
- REX.X – Extends SIB.index
- REX.B – Extends SIB.base or ModRM.r/m

ModR/M byte

The Mod-Reg-R/M byte determines the instruction's operands and the addressing modes. These are several variants of addressing modes.

1. Immediate mode - operand is part of the instruction

2. Register addressing - operand contained in register
3. Direct Mode - operand field of instruction contains address of the operand
4. Register Indirect Addressing - used for addressing data arrays with offsets
5. Indexing - constant base + register
6. Indexing With Scaling - Base + Register Offset * Scaling Factor
7. Stack Addressing - A variant of register indirect with auto increment/decrement using the RSP register implicitly
8. Jump relative addressing - RIP + offset

mod	meaning
00	Register indirect memory addressing mode
01	Indexed or base/indexed/displacement addressing mode
10	Indexed or base/indexed/displacement addressing mode + displacement
11	R/M denotes a register and uses the REG field

reg	
rax	000
rcx	001
rdx	010
rbx	011
rsp	100
rbp	101
rsi	110
rdi	111

In the case of mod = 00, 01 or 10

r/m	meaning
000	[BX+SI] or DISP[BX][SI]
001	[BX+DI] or DISP[BX+DI]
010	[BP+SI] or DISP[BP+SI]
011	[BP+DI] or DISP[BP+DI]
100	[SI] or DISP[SI]
101	[DI] or DISP[DI]
110	Displacement-only or DISP[BP]
111	[BX] or DISP[BX]

For example given the following instruction that uses register direct mode and specifies the register operand in r/m.

```
mov    rbx,rax
```

We have:

```
mod   = 0b11
reg   = 0b000
r/m   = rbx
r/m   = 0b011
ModRM = 0b11000011
ModRM = 0xc3
```

Scale Index Base

Scale is the factor by which index is multipled before being added to base to specify the address of the operand. Scale can have value of 1, 2, 4, or 8. If scale is not specified, the default value is 1.

scale	factor
0b00	1
0b01	2
0b10	4
0b11	8

Both the index and base refer to register in the usual index scheme.

scale/base	
rax	000
rcx	001
rdx	010
rbx	011
rsp	100
rbp	101
rsi	110
rdi	111

Instruction Builder Moving forward we'll create several functions mapping to X86 monadic operators which assemble instructions in the state monad. Let's do some simple arithmetic logic first.

```
arith :: X86 ()
arith = do
  mov rax (I 18)
  add rax (I 4)
  sub rax (I 2)
```

```
  imul rax (I 2)
  ret
```

Each of these functions takes in some set of operands given by the algebraic datatype `Val` and pattern matches on the values to figure out which x86 opcode to use and how to render the values to bytes.

ret

The simplest cases is simply the return function which takes no operands and is a 1-bit opcode.

```
ret :: X86 ()
ret = do
  emit [0xc3]
```

add <r64> <imm32>

Add for immediate values extends the operand with a REX.W flag to handle 64-bit immediate data.

```
0:  48 83 c0 01             add    rax,0x1
```

```
add :: Val -> Val -> X86 ()
add (R l) (I r) = do
  emit [0x48]              -- REX.W prefix
  emit [0x05]              -- ADD
  imm r
```

add <r64> <r64>

Register to register add uses the REX.W flag in the same manor but passes the source register in the ModRM.reg field using register direct mode. We do bitwise `or` over the mode `0xc0` and then shift 3 bits to specify the register in register index in the `reg` bits.

```
0:  48 01 e0                add    rax,rsp
```

```
add (R l) (R r) = do
  emit [0x48]              -- REX prefix
  emit [0x01]              -- ADD
  emit [0xc0 .|. opcode r `shiftL` 3 .|. opcode l]
```

mov <r64>, <r64>

Same logic applies for the mov instruction for both the register-to-register and immediate data cases.

```
0:  48 89 d8                mov    rax,rbx
```

```
mov :: Val -> Val -> X86 ()
mov (R dst) (R src) = do
  emit [0x48]              -- REX.W prefix
  emit [0x89]              -- MOV
  emit [0xC0 .|. opcode src `shiftL` 3 .|. opcode dst]
```

mov <r64>, <imm32>

```
0:  48 c7 c0 2a 00 00 00    mov    rax,0x2a
```

```
mov (R dst) (I src) = do
  emit [0x48]              -- REX.W prefix
  emit [0xC7]              -- MOV
  emit [0xC0 .|. (opcode dst .&. 7)]
  imm src
```

inc <r64>, dec <r64>

The inc and dec functions are slightly different in that they share the same opcode but modify the ModRM bit to specify the operation.

```
inc :: Val -> X86()
inc (R dst) = do
  emit [0x48]              -- REX prefix
  emit [0xFF]              -- INC
  emit [0xc0 + index dst]

dec (R dst) = do
  emit [0x48]              -- REX prefix
  emit [0xFF]              -- DEC
  emit [0xc0 + (index dst + 8)]
```

Putting everything together we'll JIT our function and call it from Haskell.

```
main :: IO ()
main = do
  mem <- allocateMemory jitsize              -- create jit memory
  let Right st = assemble mem arith          -- assemble symbolic program
  fn  <- jit mem (_mach st)                  -- jit compile
  res <- fn                                  -- call function
  putStrLn $ "Result: " <> show res
```

And running it we get the result.

```
$ stack exec example
Result: 40
```

Jumps & Loops Now let's write some logic that uses control flow and jumps between labeled blocks of instructions. Consider the factorial function that takes the value to compute in the `rcx` register and computes the result my repeatedly multiply the `rax` until reaching one. To do this we create a block `.factor` and use the `loop` instruction.

```
factorial:
        mov rcx, $5
        mov rax, $1
.factor:
        mul rax
        loop .factor
        ret
```

Let's look at the machine code for this assembly. Notice that the loop instruction takes a relative address in memory `fc` (i.e. go back 4 instructions) as it's operand.

```
00000000004004ed <main>:
  4004ed:       b9 05 00 00 00          mov    ecx,0x5
  4004f2:       b8 01 00 00 00          mov    eax,0x1

00000000004004f7 <.factor>:
  4004f7:       f7 e1                   mul    ecx
  4004f9:       e2 fc                   loop   4004f7 <.factor>
  4004fb:       c3                      ret
```

So let's create a label function which simply reaches into the monad and grabs the current pointer location in the JIT memory that we're at.

```
label :: X86 Val
label = do
  addr <- gets _memoff
  ptr  <- gets _memptr
  return (A addr)
```

When given an memory address, the `loop` instruction then simply emits the instruction simply emits the `0xE2` opcode and calculates the delta of the source and destination and the emits it's value as the immediate data for the instruction.

```
loop :: Val -> X86()
loop (A dst) = do
  emit [0xE2]
  src <- gets _memoff
  ptr <- gets _memptr
  emit [fromIntegral $ dst - src]
```

Now we'll create the symbolic representation of this factorial assembly our in Haskell DSL and parameterize it by a Haskell integer to compute.

```
factorial :: Int64 -> X86 ()
factorial n = do
  mov rcx (I n)
  mov rax (I 1)
  l1 <- label
  mul rcx
  loop l1
  ret
```

Putting everything together we'll JIT our function and call it from Haskell.

```
main :: IO ()
main = do
  mem <- allocateMemory jitsize
  let Right st = assemble mem (factorial 5)
  fn  <- jit mem (_mach st)
  res <- fn
  putStrLn $ "Result: " <> show res
```

And running it we get the result.

```
$ stack exec example
Result: 120
```

Calling Convention Final task is to be able to call out of the JIT into either Haskell runtime or a given C function pointer. To do this we need to look at the *calling convention* for moving in out of other logic and setting up the registers so we can hand them off to another subroutine and restore then when we jump back. In the 64 bit System V ABI calling convention, the first 5 arguments get passed in registers in order `rdi`, `rsi`, `rdx` `rcx`, `r8`, and `r9`. Subsequent arguments get passed on the stack.

For our call function we simply compute the delta of our given position and the address of the function we want to jump into.

```
call :: Val -> X86 ()
call (A dst) = do
  emit [0xE8]
  src <- gets _memoff
  imm (dst - (src + 5))
call _ = nodef
```

Before and after we call the function we are responsible for handling it's arguments and the push and popping the stack frame on entry and exit. On entry we call the function *prologue* and on exit we call the *epilogue*, in between lies the function logic.

```
prologue :: X86 ()
prologue = do
  push rbp
  mov rbp rsp

epilogue :: X86 ()
epilogue = do
  pop rax
  mov rsp rbp
  pop rbp
  ret
```

So for example let's call out to the libc `printf` function passing a string pointer to it from inside our JIT. To do this we use `dlsym` to grab the symbol reference and then pass it as an address to the `call` instruction after pushing the string pointer on the argument stack.

```
printf :: Word32 -> Word32 -> X86 ()
printf fnptr msg = do
  push rbp
  mov rbp rsp
  mov rdi (A msg)
  call (A fnptr)
  pop rbp
  mov rax (I 0)
  ret
```

Putting everything together we invoke it by grabbing the `printf` address and passing a pointer to Haskell string using our `asciz` function.

```
main :: IO ()
main = do
  let jitsize = 256*1024

  fn  <- extern "printf"
  msg <- asciz "Hello Haskell"
  mem <- allocateMemory jitsize

  let Right st = assemble mem (printf fn msg)
  join $ jit mem (_mach st)
```

Running it we get our friendly greeting by reaching outside the JIT.

```
$ stack exec example
Hello Haskell
```

LLVM

LLVM is a statically typed intermediate representation and an associated toolchain for manipulating, optimizing and converting this intermediate form into native code.

So for example consider a simple function which takes two arguments, adds them, and xors the result. Writing in IR it would be formed as such:

```
define i32 @test1(i32 %x, i32 %y, i32 %z) {
  %a = and i32 %z, %x
  %b = and i32 %z, %y
  %c = xor i32 %a, %b
  ret i32 %c
}
```

Running this through the LLVM toolchain we can target our high level IR into multiple different assembly codes mapping onto various architectures and CPUs all from the same platform agnostic intermediate representation.

x86-64

```
test1:
        .cfi_startproc
        andl    %edx, %esi
        andl    %edx, %edi
        xorl    %esi, %edi
        movl    %edi, %eax
        ret
```

ARM

```
test1:
        and     r1, r2, r1
        and     r0, r2, r0
        eor     r0, r0, r1
        mov     pc, lr
```

PowerPC

```
.L.test1:
        .cfi_startproc
        and 4, 5, 4
        and 3, 5, 3
        xor 3, 3, 4
        blr
        .long   0
        .quad   0
```

A uncommonly large amount of hardware manufacturers and software vendors (Adobe, AMD, Apple, ARM, Google, IBM, Intel, Mozilla, Qualcomm, Samsung, Xilinx) have come have converged on the LLVM toolchain as service agnostic way to talk about generating machine code.

What's even more impressive is that many of the advances in compiler optimizations and static analysis have been mechanized in the form of optimization passes so that all compilers written on top of the LLVM platform can take advantage of the same advanced optimizers that would often previously have to be developed independently.

Types

Primitive

```
i1     ; Boolean type
i8     ; char
i32    ; 32 bit integer
i64    ; 64 bit integer
float  ; 32 bit
double ; 64 bit
```

Arrays

```
[10 x float]      ; Array of 10 floats
[10 x [20 x i32]] ; Array of 10 arrays of 20 integers.
```

Structs

```
{float, i64}           ; structure
{float, {double, i3}}  ; nested structure
<{float, [2 x i3]}>    ; packed structure
```

Vectors

```
<4 x double>
<8 x float>
```

Pointers

```
float*        ; Pointer to a float
[25 x float]* ; Pointer to an array
```

The traditional `void*` pointer in C is a `i8*` pointer in LLVM with the appropriate casts.

Constants

```
[i1 true, i1 false]    ; constant bool array
<i32 42, i32 10>       ; constant vector
float 1.23421e+2       ; floating point constant
null                   ; null pointer constant
```

The `zeroinitializer` can be used to instantiate any type to the appropriate zero of any type.

```
<8 x float> zeroinitializer ; Zero vector
```

Named Types

```
%vec4 = type <4 x i32>
%pair = type { i32, i32 }
```

Recursive types declarations are supported.

```
%f = type { %f*, i32 }
```

Platform Information

```
target datalayout = "
   e-
   p      : 64  : 64   : 64-
   i1     : 8   : 8-
   i8     : 8   : 8-
   i16    : 16  : 16-
   i32    : 32  : 32-
   i64    : 64  : 64-
   f32    : 32  : 32-
   f64    : 64  : 64-
   v64    : 64  : 64-
   v128   : 128 : 128-
   a0     : 0   : 64-
   s0     : 64  : 64-
   f80    : 128 : 128-
   n8     : 16  : 32   : 64-
   S128
   "
target triple = "x86_64-unknown-linux-gnu"
```

Specifications are delimited by the minus sign `-`.

- The `e` indicates the platform is little-endian.

- The `i<n>` indicate the bitsize and alignment of the integer type.
- The `f<n>` indicate the bitsize and alignment of the floating point type.
- The `p<n>` indicate the bitsize and alignment of the pointer type.
- The `v<n>` indicate the bitsize and alignment of the vector type.
- The `a<n>` indicate the bitsize and alignment of the aggregate type.
- The `n<n>` indicate the widths of the CPU registers.
- The `S<n>` indicate the alignment of the stack.

Variables

Symbols used in an LLVM module are either global or local. Global symbols begin with `@` and local symbols begin with `%`. All symbols must be defined or forward declared.

Instructions in LLVM are either numbered sequentially (`%0`, `%1`, …) or given explicit variable names (`%a`, `%foo`, ..). For example the arguments to the following function are named values, while the result of the add instructions unnamed.

```
define i32 @add(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  ret i32 %1
}
```

Instructions

```
%result = add i32 10, 20
```

Logical

- `shl`
- `lshr`
- `ashr`
- `and`
- `or`
- `xor`

Binary Operators

- `add`
- `fadd`
- `sub`
- `fsub`
- `mul`

- `fmul`
- `udiv`
- `sdiv`
- `fdiv`
- `urem`
- `srem`
- `frem`

Comparison

op	unsigned	signed	floating
lt	ULT	SLT	OLT
gt	UGT	SGT	OGT
le	ULE	SLE	OLE
ge	UGE	SGE	OGE
eq	EQ	EQ	OEQ
ne	NE	NE	ONE

```
%c = udiv i32 %a, %b
%d = sdiv i32 %a, %b
%e = fmul float %a, %b
%f = fdiv float %a, %b

%g = icmp eq i32 %a, %b
%i = icmp slt i32 %a, %b
%j = icmp ult i32 %a, %b
%k = fcmp olt float, %a, %b
```

Data

```
i1 1
i32 299792458
float 7.29735257e-3
double 6.62606957e-34
```

Blocks

Function definitions in LLVM introduce a sequence of labeled *basic blocks* containing any number of instructions and a final *terminator* instruction which indicates how control flow yields after the instructions of the basic block are evaluated.

```
define i1 @foo() {
entry:
  br label %next
next:
  br label %return
return:
  ret i1 0
}
```

A basic block has either zero (for entry block) or a fixed number of *predecessors*. A graph with basic blocks as nodes and the predecessors of each basic block as edges constitutes a *control flow graph*. LLVM's opt command can be used to dump this graph using graphviz.

```
$ opt -view-cfg module.ll
$ dot -Tpng module.dot -o module.png
```

We say a basic block A *dominates* a different block B in the control flow graph if it's impossible to reach B without passing through "A, equivalently A is the *dominator* of B.

All logic in LLVM is written in *static single assignment* (SSA) form. Each variable is assigned precisely once, and every variable is defined before it is used. Updating any existing variable reference creates a new reference with for the resulting output.

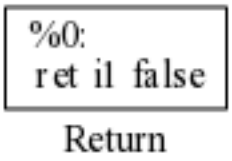
Control Flow

- Unconditional Branch
- Conditional Branch
- Switch
- Return
- Phi

Return

The `ret` function simply exits the current function yielding the current value to the virtual stack.

```
define i1 @foo() {
  ret i1 0
}
```

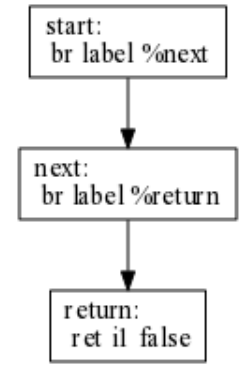


Return

Unconditional Branch

The unconditional branch `br` simply jumps to any basic block local to the function.

```
define i1 @foo() {
  entry:
    br label %next
  next:
    br label %return
  return:
    ret i1 0
}
```

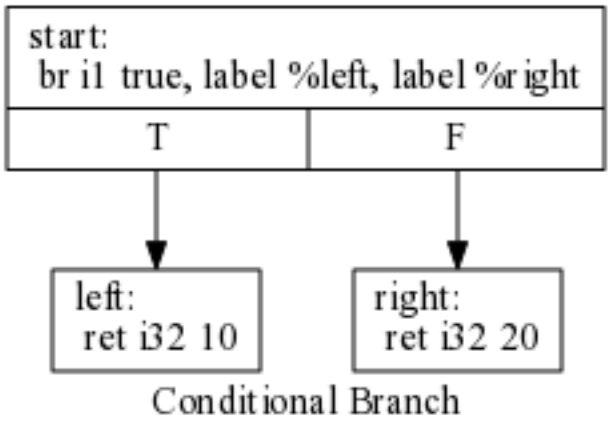


Unconditional Branch

Conditional Branch

The conditional branch `br` jumps to one of two basic blocks based on whether a test condition is `true` or `false`. This corresponds the logic of a traditional "if statement".

```
define i32 @foo() {
start:
  br i1 true, label %left, label %right
left:
  ret i32 10
right:
  ret i32 20
}
```

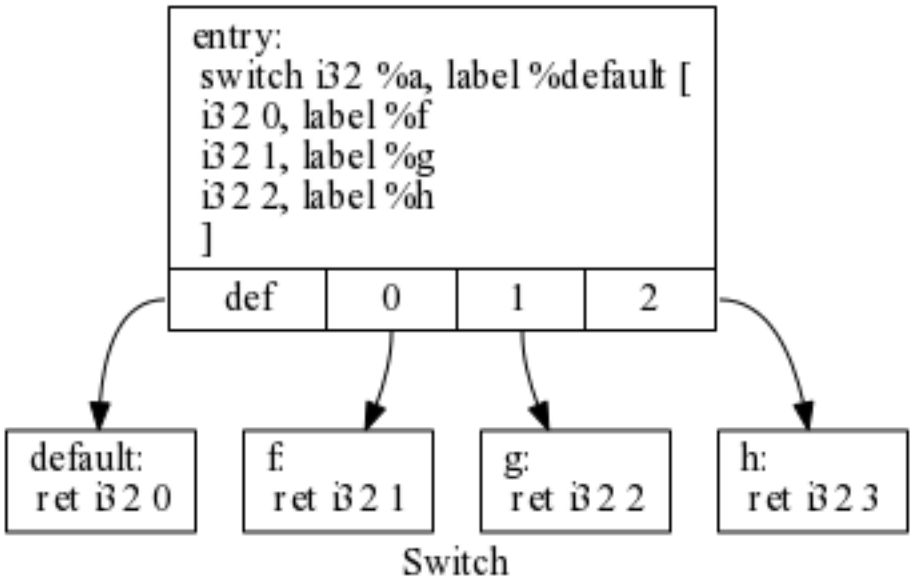


Conditional Branch

Switch

The switch statement `switch` jumps to any number of branches based on the equality of value to a jump table matching values to basic blocks.

```
define i32 @foo(i32 %a) {
entry:
  switch i32 %a, label %default [
    i32 0, label %f
    i32 1, label %g
    i32 2, label %h
  ]
f:
  ret i32 1
g:
  ret i32 2
h:
  ret i32 3
default:
  ret i32 0
}
```

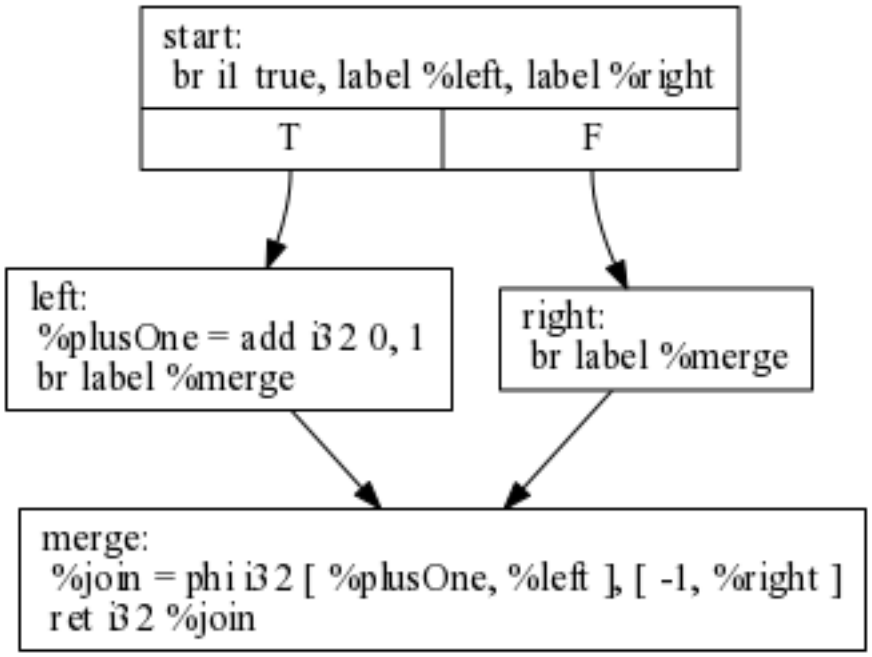


Switch

Phi

A `phi` node selects a value based on the predecessor of the current block.

```
define i32 @foo() {
start:
  br i1 true, label %left, label %right
left:
  %plusOne = add i32 0, 1
  br label %merge
right:
  br label %merge
merge:
  %join = phi i32 [ %plusOne, %left ], [ -1, %right ]
  ret i32 %join
}
```



Phi node

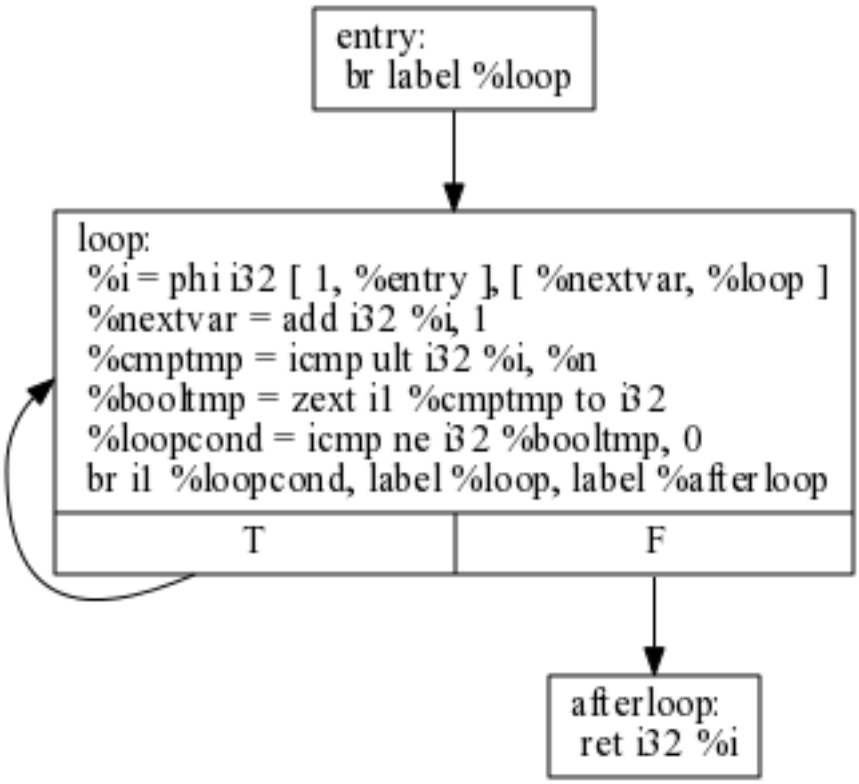
Loops

The traditional `while` and `for` loops can be written in terms of the simpler conditional branching constructrs. For example in C we would write:

```
int count(int n)
{
  int i = 0;
  while(i < n)
  {
    i++;
  }
  return i;
}
```

Whereas in LLVM we write:

```
define i32 @count(i32 %n) {
entry:
   br label %loop

loop:
   %i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
   %nextvar = add i32 %i, 1

   %cmptmp = icmp ult i32 %i, %n
   %booltmp = zext i1 %cmptmp to i32
   %loopcond = icmp ne i32 %booltmp, 0

   br i1 %loopcond, label %loop, label %afterloop

afterloop:
   ret i32 %i
}
```


entry:
br label %loop
loop:
%i = phi i32 [1, %entry], [%nextvar, %loop]
%nextvar = add i32 %i, 1
%cmptmp = icmp ult i32 %i, %n
%booltmp = zext i1 %cmptmp to i32
%loopcond = icmp ne i32 %booltmp, 0
br i1 %loopcond, label %loop, label %afterloop
T
F
afterloop:
ret i32 %i

For loop

Select

Selects the first value if the test value is true, the second if false.

```
%x = select i1 true, i8 10, i8 20   ; gives 10
%y = select i1 false, i8 10, i8 20  ; gives 20
```

Calls

- `ccc`: The C calling convention
- `fastcc`: The fast calling convention

```
%result = call i32 @exp(i32 7)
```

Memory

LLVM uses the traditional load/store model:

- `load`: Load a typed value from a given reference
- `store`: Store a typed value in a given reference
- `alloca`: Allocate a pointer to memory on the virtual stack

```
%ptr = alloca i32
store i32 3, i32* %ptr
%val = load i32* %ptr
```

Specific pointer alignment can be specified:

```
%ptr = alloca i32, align 1024
```

For allocating in main memory we use an external reference to the C stdlib memory allocator which gives us back a (`i8*`).

```
%ptr = call i8* @malloc(i32 %objectsize)
```

For structures:

```
extractvalue {i32, float} %a, 0              ; gives i32
extractvalue {i32, {float, double}} %a, 0, 1 ; gives double
extractvalue [2 x i32] %a, 0                 ; yields i32

%x = insertvalue {i32, float} %b, float %val, 1         ; gives {i32 1, float %b}
%y = insertvalue {i32, float} zeroinitializer, i32 1, 0 ; gives {i32 1, float 0}
```

GetElementPtr

Casts

- trunc
- zext
- sext
- fptoui
- fptosi
- uitofp
- sitofp
- fptrunc
- fpext
- ptrtoint
- inttoptr
- bitcast

```
trunc i32 257 to i8         ; yields i8  1
zext i32 257 to i64         ; yields i64 257
sext i8 -1 to i16           ; yields i16 65535
bitcast <2 x i32> %a to i64 ; yields i64 %a
```

Toolchain

```
$ llc example.ll -o example.s             # compile
$ lli example.ll                          # execute
$ opt -S example.bc -o example.ll         # to assembly
$ opt example.ll -o example.bc            # to bitcode
$ opt -O3 example.ll -o example.opt.ll -S # run optimizer
```

Individual modules can be linked together.

```
$ llvm-link a.ll b.ll -o c.ll -S
```

Link time optimization.

```
$ clang -O4 -emit-llvm a.c -c -o a.bc
$ clang -O4 -emit-llvm a.c -c -o a.bc
$ llvm-link a.bc b.bc -o all.bc
$ opt -std-compile-opts -std-link-opts -O3 all.bc -o optimized.bc
```

The clang project is a C compiler that targets LLVM as it's intermediate representation. In the case where we'd like to know how some specific C construct maps into LLVM IR we can ask clang to dump its internal IR using the `-emit-llvm` flag.

```
# clang -emit-llvm -S add.c -o -
int add(int x)
{
  return x+1;
}

; ModuleID = 'add.c'
define i32 @add(i32 %x) nounwind uwtable {
entry:
  %x.addr = alloca i32, align 4
  store i32 %x, i32* %x.addr, align 4
  %0 = load i32* %x.addr, align 4
  %add = add nsw i32 %0, 1
  ret i32 %add
}
```

LLVM is using a C++ API underneath the hood of all these tools. If you need to work directly with the API it can be useful to be able to expand out the LLVM IR into the equivalent C++ code.

```
$ llc example.ll -march=cpp -o -

define i32 @test1(i32 %x, i32 %y, i32 %z) {
  %a = and i32 %z, %x
  %b = and i32 %z, %y
  %c = xor i32 %a, %b
  ret i32 %c
}

Function* func_test1 = mod->getFunction("test1");
if (!func_test1) {
func_test1 = Function::Create(
 /*Type=*/FuncTy_0,
 /*Linkage=*/GlobalValue::ExternalLinkage,
 /*Name=*/"test1", mod);
func_test1->setCallingConv(CallingConv::C);
}
AttrListPtr func_test1_PAL;
func_test1->setAttributes(func_test1_PAL);

{
 Function::arg_iterator args = func_test1->arg_begin();
 Value* int32_x = args++;
 int32_x->setName("x");
 Value* int32_y = args++;
 int32_y->setName("y");
```

```
 Value* int32_z = args++;
 int32_z->setName("z");

 BasicBlock* label_1 = BasicBlock::Create(mod->getContext(), "",func_test1,0);

 BinaryOperator* int32_a = BinaryOperator::Create(
   Instruction::And, int32_z, int32_x, "a", label_1);
 BinaryOperator* int32_b = BinaryOperator::Create(
   Instruction::And, int32_z, int32_y, "b", label_1);
 BinaryOperator* int32_c = BinaryOperator::Create(
   Instruction::Xor, int32_a, int32_b, "c", label_1);
 ReturnInst::Create(mod->getContext(), int32_c, label_1);

}
```

llvm-general

The LLVM bindings for Haskell are split across two packages:

- **llvm-general-pure** is a pure Haskell representation of the LLVM IR.
- **llvm-general** is the FFI bindings to LLVM required for constructing the C representation of the LLVM IR and performing optimization and compilation.

llvm-general-pure does not require the LLVM libraries be available on the system.

GHCi can have issues with the FFI and can lead to errors when working with `llvm-general`. If you end up with errors like the following, then you are likely trying to use `GHCi` or `runhaskell` and it is unable to link against your LLVM library. Instead compile with standalone `ghc`.

```
Loading package llvm-general-3.3.8.2
... linking
... ghc: /usr/lib/llvm-3.3/lib/libLLVMSupport.a: unknown symbol '_ZTVN4llvm14error_categoryE'
ghc: unable to load package 'llvm-general-3.3.8.2'
```

Code Generation (LLVM)

Resources

- LLVM Language Reference
- Implementing a JIT Compiled Language with Haskell and LLVM