# Lingo:

## A prototypical functional language for linearity polymorphism

Benjamin Flin, Jay Karp, Sophia Kolak

# Agenda

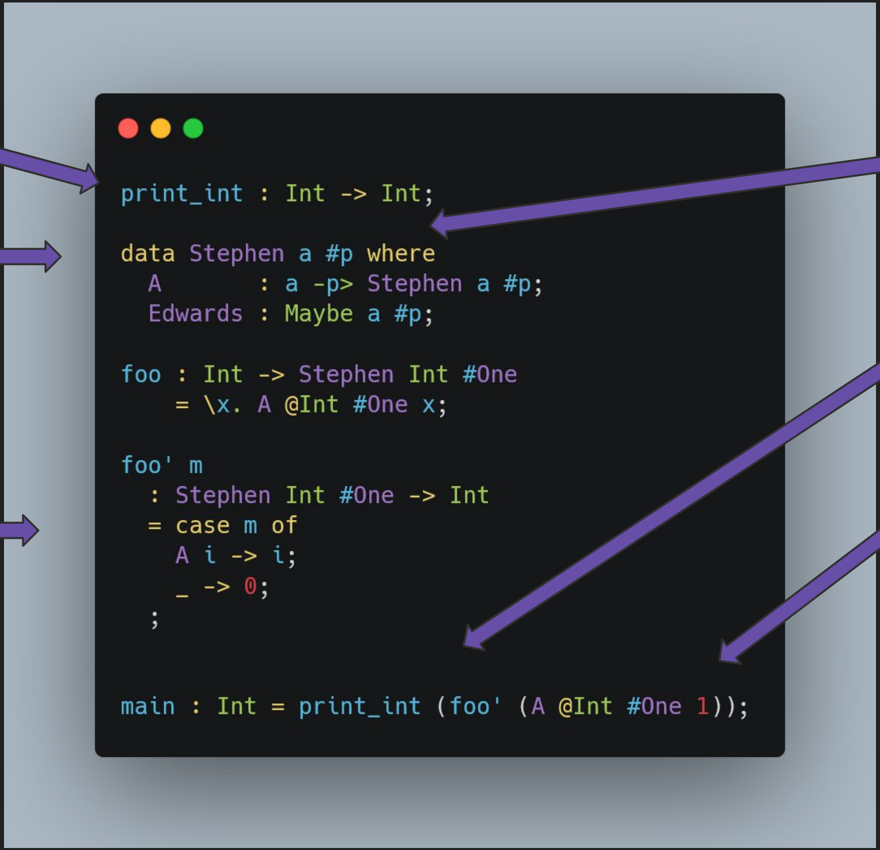
1. Lingo Overview
2. Linear Typing Motivation
3. Compiler Architecture
4. Testing
5. Demo
6. Future Work

# Lingo

- Linear Typing
- Functional/Strong type
- Algebraic Data Types
- Pattern matching
- Rank-n polymorphism
- First class functions / lambda calculus
- C interoperability

# Lingo in one Slide

# Lingo in One Slide



```
print_int : Int -> Int;

data Stephen a #p where
  A       : a -p> Stephen a #p;
  Edwards : Maybe a #p;

foo : Int -> Stephen Int #One
    = \x. A @Int #One x;

foo' m
  : Stephen Int #One -> Int
  = case m of
    A i -> i;
    _ -> 0;
  ;


main : Int = print_int (foo' (A @Int #One 1));
```

External Function Declaration

Abstract Data Types

Cases + Pattern Matching

Type and Multiplicity Polymorphism

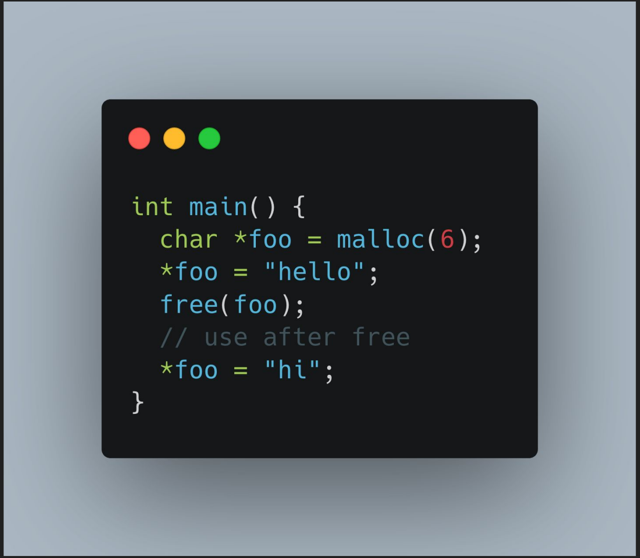
Function Application

Multiplicities and Types

# Motivation for Linear Typing

- Want a type system which eliminates classes of bugs violating resource protocols, i.e. double free, closing file pointers, use after free, etc.

- In order to do this, we need to characterize whether/how a function will use or evaluate its argument.



```c
int main() {
  char *foo = malloc(6);
  *foo = "hello";
  free(foo);
  // use after free
  *foo = "hi";
}
```

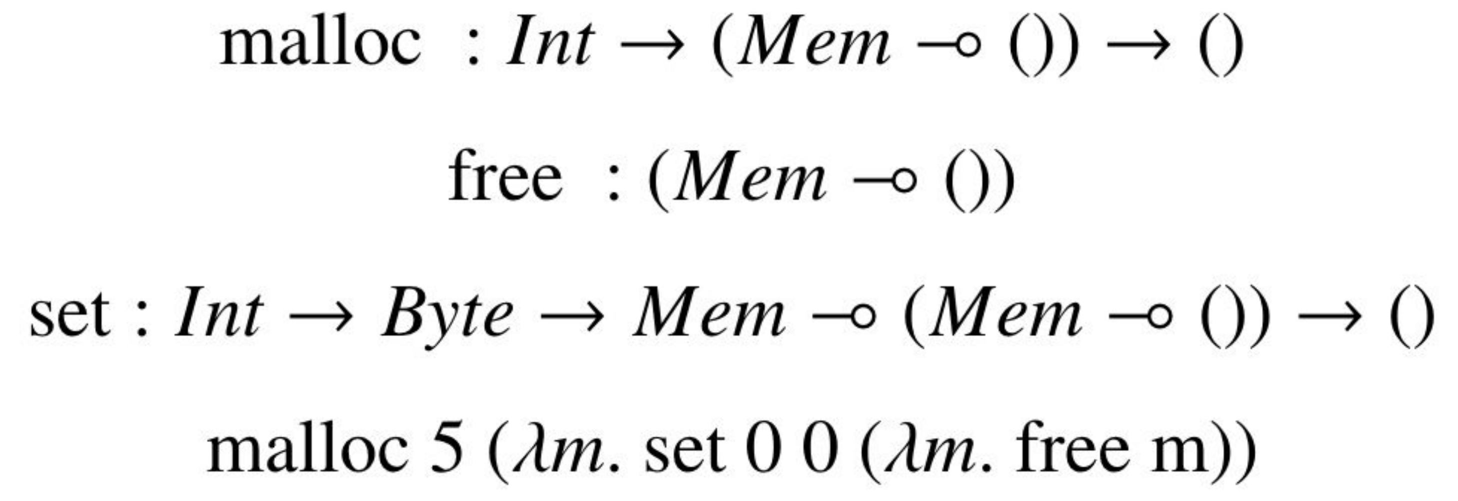
# Linear arrows

- Extend the normal function arrow → with a linear one -o.
- Linear arrow from a -o b functions same as a normal function arrow, with one additional guarantee: if the output of an application, is evaluated once, then the input will be evaluated exactly once.
- Guarantee is enough to ensure that the programmer follows a resource protocols.
    - `f : a -o b`
    - `u : a`
    - `f u : b (* u is used linearly *)`

$$(a \rightarrow b)$$

$$(a \multimap b)$$

# Linearity Example



$$\text{malloc} \;\; : \mathit{Int} \rightarrow (\mathit{Mem} \multimap ()) \rightarrow ()$$

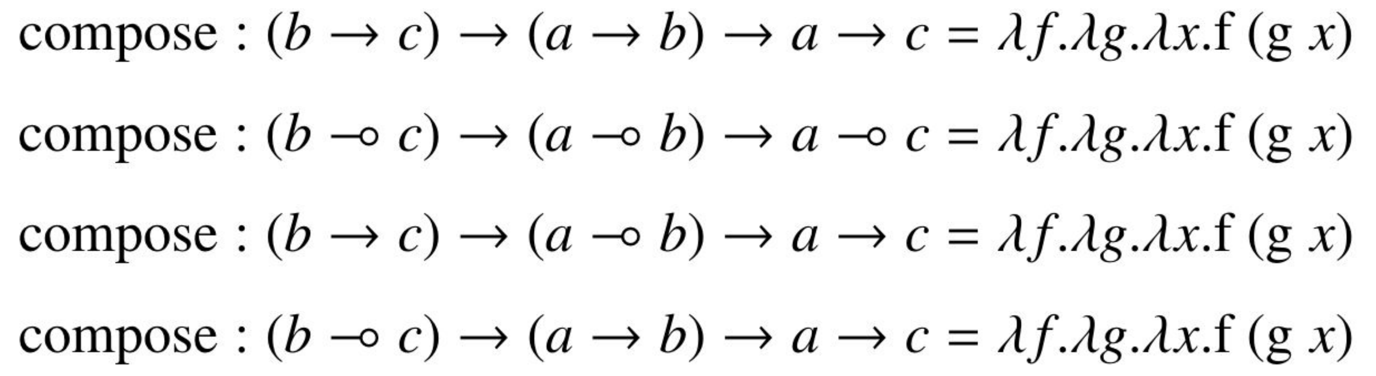
$$\text{free} \;\; : (\mathit{Mem} \multimap ())$$

$$\text{set} : \mathit{Int} \rightarrow \mathit{Byte} \rightarrow \mathit{Mem} \multimap (\mathit{Mem} \multimap ()) \rightarrow ()$$

$$\text{malloc } 5 \; (\lambda m.\; \text{set } 0 \; 0 \; (\lambda m.\; \text{free m}))$$

# A problem with compose

How should the following be typed?



$$\text{compose} : (b \to c) \to (a \to b) \to a \to c = \lambda f.\lambda g.\lambda x.\text{f}\ (\text{g}\ x)$$

$$\text{compose} : (b \multimap c) \to (a \multimap b) \to a \multimap c = \lambda f.\lambda g.\lambda x.\text{f}\ (\text{g}\ x)$$

$$\text{compose} : (b \to c) \to (a \multimap b) \to a \to c = \lambda f.\lambda g.\lambda x.\text{f}\ (\text{g}\ x)$$

$$\text{compose} : (b \multimap c) \to (a \to b) \to a \to c = \lambda f.\lambda g.\lambda x.\text{f}\ (\text{g}\ x)$$

Need 4 different types for the same function body. We answer with linearity polymorphism.

# System F

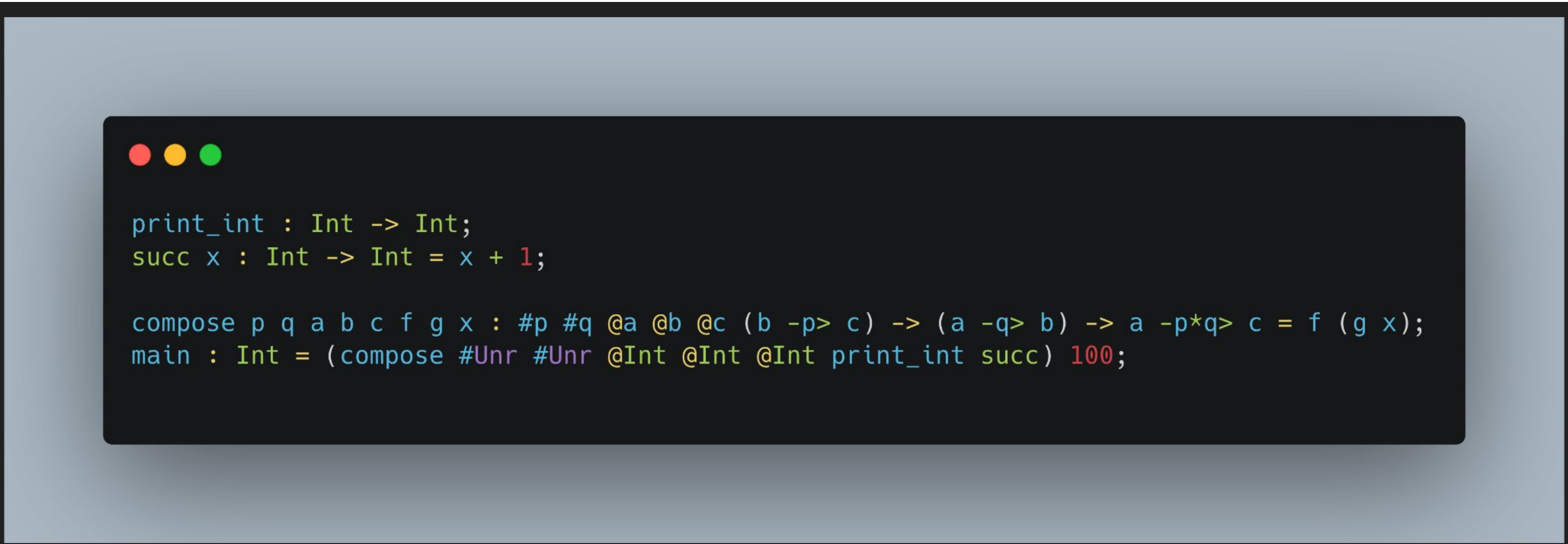
- Extends simply-typed lambda calculus with a a new type of abstraction for types denoted by uppercase lambda.
- Types can be applied and they are substituted on the type-level.
- **Idea:** Use this type system to characterize polymorphism in linearity

$$id = \Lambda\alpha.\lambda(x : \alpha).x : \forall\alpha.\alpha \to \alpha$$

$$\text{foo} = \text{id Int } 0 : \text{Int}$$

# Compose again

$$\text{compose} : \forall_m p.\forall_m q.\forall a.\forall b.\forall c.(b \to_p c) \to (a \to_q b) \to a \to_{pq} c = \Lambda_m p.\Lambda_m q.\Lambda a.\Lambda b.\Lambda c.\lambda f.\lambda g.\lambda x.\mathrm{f}\,(\mathrm{g}\ x)$$



```
print_int : Int -> Int;
succ x : Int -> Int = x + 1;

compose p q a b c f g x : #p #q @a @b @c (b -p> c) -> (a -q> b) -> a -p*q> c = f (g x);
main : Int = (compose #Unr #Unr @Int @Int @Int print_int succ) 100;
```

# Demo compose

# Typing rules

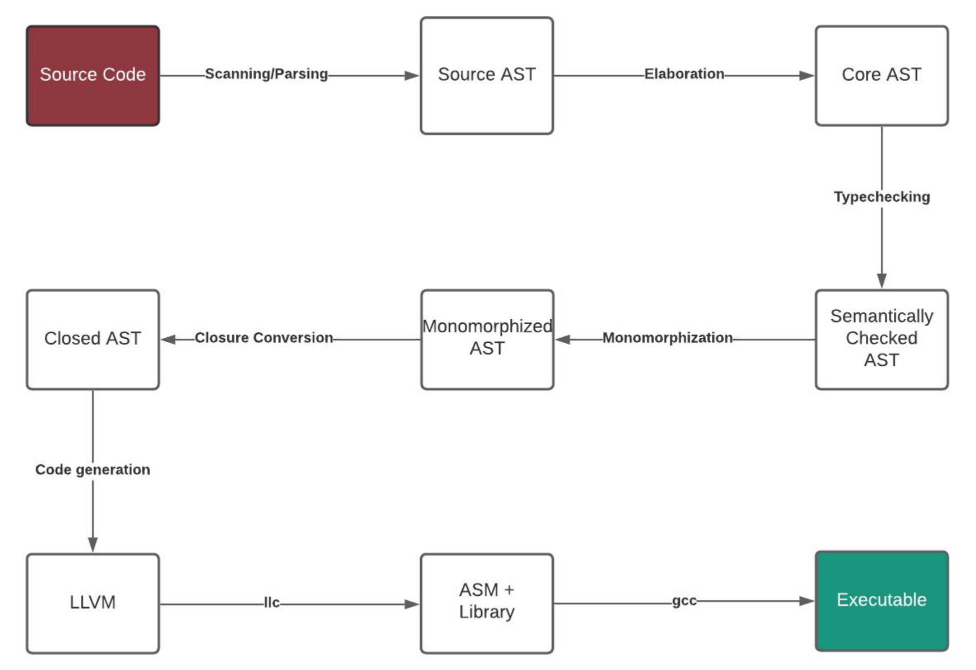
$$\frac{x \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow \{x \mapsto 1\}}\,\text{var}$$

$$\frac{\Gamma, x : A \vdash t : B \rightsquigarrow \{x \mapsto \mu, U\} \qquad \mu \leqslant \pi}{\Gamma \vdash \lambda(x{:}_{\pi}A).t : A \rightarrow_{\pi} B \rightsquigarrow \{U\}}\,\text{abs}$$

$$\frac{\Gamma \vdash t : A \rightarrow_{\pi} B \rightsquigarrow \{U\} \qquad \Gamma \vdash u : A \rightsquigarrow \{V\}}{\Gamma \vdash t\ u : B \rightsquigarrow \{U + \pi V\}}\,\text{app}$$

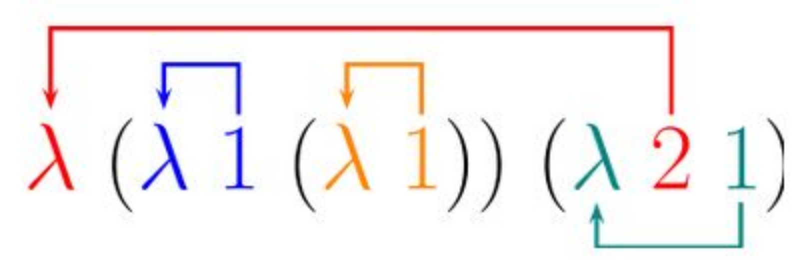
$$\frac{\Gamma \vdash t : A \rightsquigarrow \{U\} \qquad p \text{ fresh for } \Gamma}{\Gamma \vdash \lambda p.t : \forall p.A \rightsquigarrow \{U\}}\,\text{m.abs} \qquad \frac{\Gamma \vdash t : \forall p.A \rightsquigarrow \{U\}}{\Gamma \vdash t\ \pi : A[\pi/p] \rightsquigarrow \{U\}}\,\text{m.app}$$

# Compiler Architecture



# Elaboration

- Source Ast -> Core Ast
- During elaboration, all syntax is expanded into a form which the typechecker can understand.
- This includes
    - Converting all variable names to Debruijn indices
    - Each lambda into their corresponding abstractions (value, type, or multiplicity)
    - Variable lists into nested lambdas.

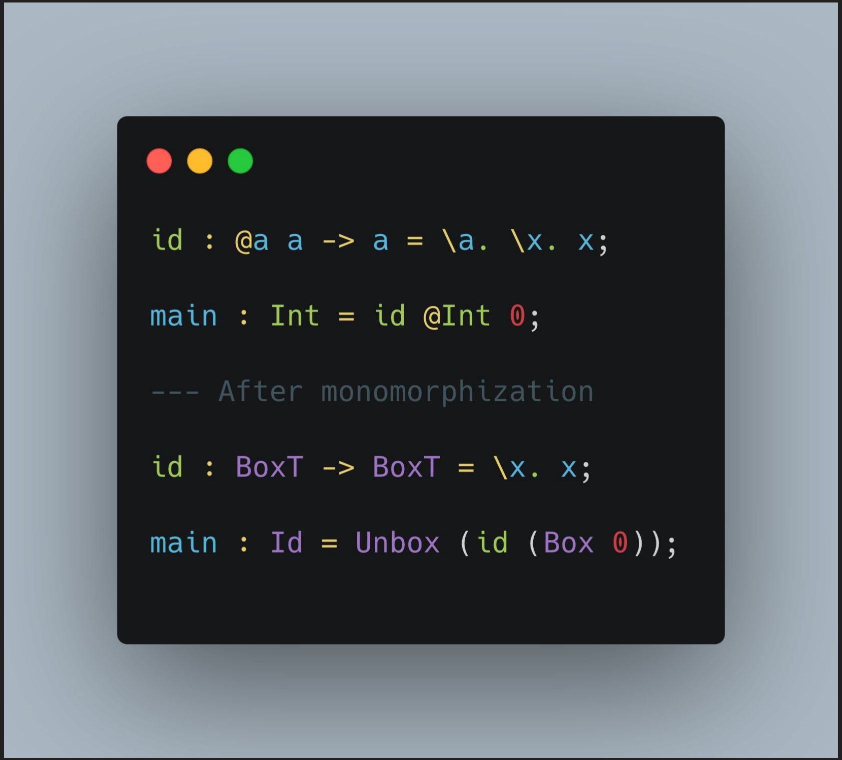


# Typechecking

- Core ast -> semantically checked **Sast** with annotated types.
- linearity checking and type checking happens.
- Multiplicities are stripped out
- If there is a type or multiplicity mismatch of any sort, or any other type related error, the typechecker will throw an error.

# Monomorphization

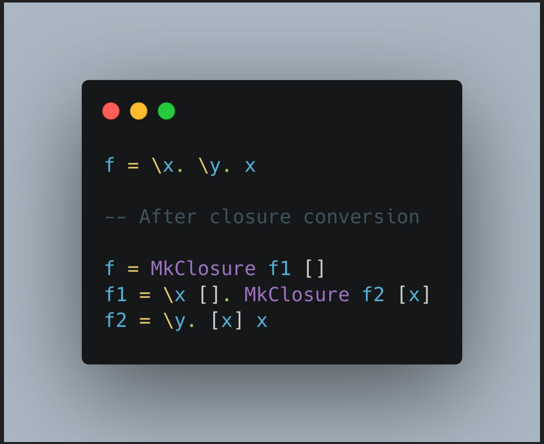
- polymorphic / rank-n code -> non-polymorphic / rank-1 code.
- LLVM doesn’t have polymorphism!
- Our strategy: **BoxT**, a single type which represents all polymorphic variables.
- Convert to and from **BoxT** through the use of new expressions **Box** and **Unbox**.
- **Box:** turns any value into a **BoxT**
- **Unbox:** turns any **Box** value back into a given type.
- Application to a lambda of type **BoxT -> BoxT** boxes the argument and unboxes the result.
- Later down the pipeline during code gen, this will result in a simple cast in LLVM.



```
id : @a a -> a = \a. \x. x;

main : Int = id @Int 0;

--- After monomorphization

id : BoxT -> BoxT = \x. x;

main : Id = Unbox (id (Box 0));
```

# Closure Conversion

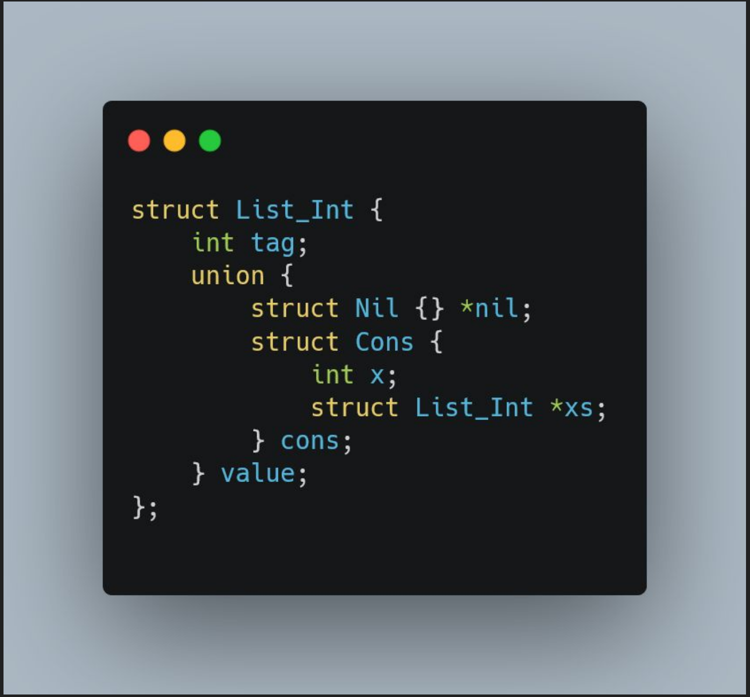
- **Mast** -> performs lambda lifting until everything is a **top level** declaration
- Uniquely named lambdas
- Takes every free variable and adds it to a closure environment



```
f = \x. \y. x

-- After closure conversion

f = MkClosure f1 []
f1 = \x []. MkClosure f2 [x]
f2 = \y. [x] x
```

# Code Generation

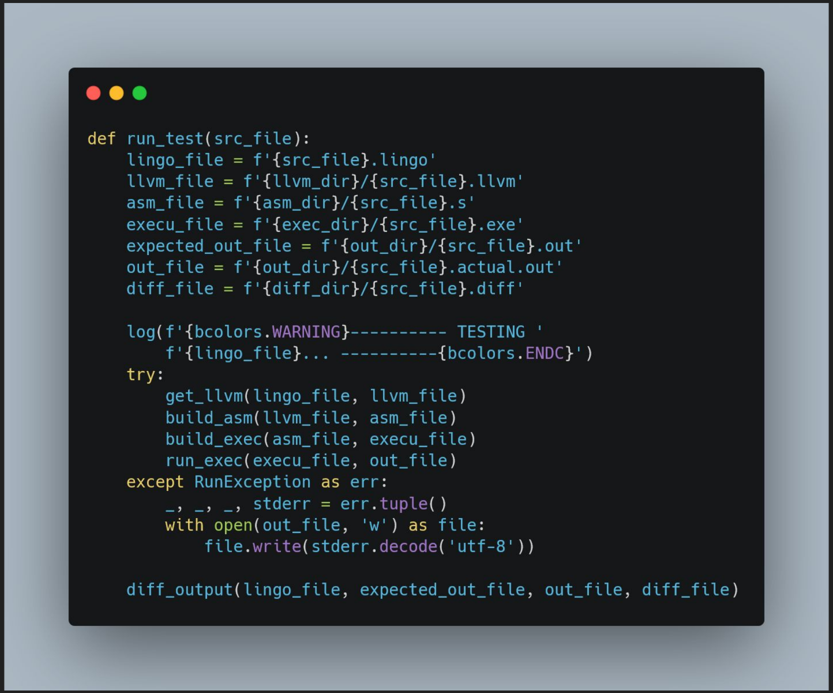
- Conversion of Closure Converted Cast -> llvm IR
- Abstract data types become tagged unions, i.e. { i64, i8* }. The first parameter, the tag, tells us what constructor is pointed to as the second parameter.
- Everything becomes function application and building closures.
- External C Calls



```
struct List_Int {
    int tag;
    union {
        struct Nil {} *nil;
        struct Cons {
            int x;
            struct List_Int *xs;
        } cons;
    } value;
};
```

# Testing

- Testing run inside docker container before every git commit
- Passing and Failing Tests
- Syntax
- Memory Allocation
- Function Composition
- Polymorphism
- Basic Operations
- Abstract Data Types



```python
def run_test(src_file):
    lingo_file = f'{src_file}.lingo'
    llvm_file = f'{llvm_dir}/{src_file}.llvm'
    asm_file = f'{asm_dir}/{src_file}.s'
    execu_file = f'{exec_dir}/{src_file}.exe'
    expected_out_file = f'{out_dir}/{src_file}.out'
    out_file = f'{out_dir}/{src_file}.actual.out'
    diff_file = f'{diff_dir}/{src_file}.diff'

    log(f'{bcolors.WARNING}---------- TESTING '
        f'{lingo_file}... ----------{bcolors.ENDC}')
    try:
        get_llvm(lingo_file, llvm_file)
        build_asm(llvm_file, asm_file)
        build_exec(asm_file, execu_file)
        run_exec(execu_file, out_file)
    except RunException as err:
        _, _, _, stderr = err.tuple()
        with open(out_file, 'w') as file:
            file.write(stderr.decode('utf-8'))

    diff_output(lingo_file, expected_out_file, out_file, diff_file)
```

# Demo Safe File (Time Permitting)

# Future Work

- Better monomorphization? (may be intractable)
- Qualified types and interfaces
- Replace current type checker with quantitative type theory (dependent types + linear types)
  - This actually may have been easier to implement in the long-run as types and terms exist on the same level. Typechecking will become a little more costly, however.
- Garbage Collection / better memory management
- Fix bugs (hopefully there are no big ones :) )
  - “A segfault a day keeps the sanity away” - Ben Flin

# Thank You

Questions?