System F in the real world.
Haskell and functional programming.

Today's talk is to answer the question:
"Great, now why do we care?"

We have spoken about "programs" and "computability", but havent spoken about how to do so on an actual computer.

Computers are very good at shifting memory around, and following instructions quickly and accurately. (it is what they were built to do).
This makes sense! If you were the first person to think about how to automate some process, then the first natural thing to think of is to write down a list of instructions for some diligent person to follow as accurately as they can.

As a concrete excample, say we wanted to concatinate 2 sequences together.
Imperative approach: Create new sequence with blank entries whose length is the sum of the two lengths of the two sequences to be concatinated.
Then fill in the blanks.

As we saw last week, there are other ways.

Using a function composition approach is referred to as "functional programming".

There are computer languages that allow us to code "functionally". The language "Haskell" is one such example.

Haskell compiles to a language called "Core", which has a formal mathematical definition [An External Representation for the GHC Core Language. Tolmach, Chevalier, the GHC Team, 2010].

When we say "Haskell is system F", we are being a bit liberal with the word "is".

Core is an extension of System F.
Core is intended to be a human readible/writable version of an extension of System F called System FC.
The C stands for "equality constraints" and "coercions", which are higher order concepts, and are beyond the scope of this talk.

over the page

We will now go through the grammar of Core, and point out the bits we can currently understand.

Important bits:

$\backslash @a = \Lambda a$ $\quad$ %forall a.b = $\forall a.b$

$\backslash (x::a) = \lambda x^a$

In exp -> \ {binder}$^+$ -> exp, -> = . in $\lambda x.M$.

binder -> @tbind
| vbind

Change to

binder -> @tyvar
| vbind.

(Similar for ty -> %forall {tbind}$^+$. ty)

A word on literals:
We want to make use of the ALU (Arithmetic Logic Unit) inside our computers which perform equality comparisons and > relation on integers, as this will be faster than using Church Numerals, etc, so we have special expressions called literals.

These pages are taken from "An External Representation for the GHC Core Language", by Andrew Tolmach ~~and~~, Tim Chevalier, and the GHC Team.

- All type abstractions and applications are given in full, even though some of them (e.g., for tuples) could be reconstructed; this means a parser for Core does not have to reconstruct types.[3]
- The syntax of identifiers is heavily restricted (to just alphanumerics and underscores); this again makes Core easier to parse but harder to read.

We use the following notational conventions for syntax:

[pat]	optional
{ pat }	zero or more repetitions
{ pat }$^{+}$	one or more repetitions
pat_1 | pat_2	choice
`fibonacci`	terminal syntax in typewriter font

These pages present the full language, we cross out higher level concepts to present a simplified version.

Crossing off: Coercions,
Kinds,
Constructors.

[3] These choices are certainly debatable. In particular, keeping type applications on tuples and case arms considerably increases the size of Core files and makes them less human-readable, though it allows a Core parser to be simpler.

Module	*module*	→	`%module` *mident* `{` *tdef* `;` `}` `{` *vdefg* `;` `}`	
Type defn.	~~*tdef*~~	~~→~~	~~`%data` *qtycon* `{` *tbind* `}` `=` `{` `/` *cdef* `{` `;` *cdef* `}` `/` `}`~~	~~algebraic type~~
		~~|~~	~~`%newtype` *qtycon qtycon* `{` *tbind* `}` `=` *ty*~~	~~newtype~~
Constr. defn.	*cdef*	→	*qdcon* `{` `@` ~~*tbind*~~ *tyvar* `}` `{` *aty* `}`$^+$	
Value defn.	*vdefg*	→	`%rec` `{` *vdef* `{` `;` *vdef* `}` `}`	recursive
		|	*vdef*	non-recursive
	vdef	→	*qvar* `::` *ty* `=` *exp*	
Atomic expr.	*aexp*	→	*qvar*	variable
		~~|~~	~~*qdcon*~~	~~data constructor~~
		|	*lit*	literal
		|	`(` *exp* `)`	nested expr.
Expression	*exp*	→	*aexp*	atomic expresion
		|	*aexp* `{` *arg* `}`$^+$	application
		|	`\` `{` *binder* `}`$^+$ `->` *exp*	abstraction
		|	`%let` *vdefg* `%in` *exp*	local definition
		|	`%case` `(`*aty*`)` *exp* `%of` *vbind* `{` *alt* `{` `;` *alt* `}` `}`	case expression
		~~|~~	~~`%cast` *exp aty*~~	~~type coercion~~
		~~|~~	~~`%note` `"` `{` *char* `}` `"` *exp*~~	~~expression note~~
		~~|~~	~~`%external ccall "` `{` *char* `}` `"` *aty*~~	~~external reference~~
		~~|~~	~~`%dynexternal ccall` *aty*~~	~~external reference (dynamic)~~
		~~|~~	~~`%label "` `{` *char* `}` `"`~~	~~external label~~
Argument	*arg*	→	`@` *aty*	type argument
		|	*aexp*	value argument
Case alt.	*alt*	→	*qdcon* `{` `@` *tbind* `}` `{` *vbind* `}` `->` *exp*	constructor alternative
		|	*lit* `->` *exp*	literal alternative
		|	`%_` `->` *exp*	default alternative
Binder	*binder*	→	`@` ~~*tbind*~~ *tyvar*	type binder
		|	*vbind*	value binder
Type binder	*tbind*	→	*tyvar*	implicitly of kind *
		|	`(` *tyvar* `::` *kind* `)`	explicitly kinded
Value binder	*vbind*	→	`(` *var* `::` *ty* `)`	
Literal	*lit*	→	`(` *[-]* `{` *digit* `}`$^+$ `::` *ty* `)`	integer
		|	`(` *[-]* `{` *digit* `}`$^+$ `%` `{` *digit* `}`$^+$ `::` *ty* `)`	rational
		|	`(` `'` *char* `'` `::` *ty* `)`	character
		|	`(` `"` `{` *char* `}` `"` `::` *ty* `)`	string
Character	*char*	→	*any ASCII character in range 0x20-0x7E except 0x22,0x27,0x5c*	
		|	`\x` *hex hex*	ASCII code escape sequence
	hex	→	`0` |...| `9` | `a` |...| `f`	

Atomic type	*aty*	→	*tyvar*	type variable
		|	~~*qtycon*~~	~~type constructor~~
		|	(*ty*)	nested type
Basic type	*bty*	→	*aty*	atomic type
		|	*bty aty*	type application
		|	~~`%trans` *aty aty*~~	~~transitive coercion~~
		|	~~`%sym` *aty*~~	~~symmetric coercion~~
		|	~~`%unsafe` *aty aty*~~	~~unsafe coercion~~
		|	~~`%left` *aty*~~	~~left coercion~~
		|	~~`%right` *aty*~~	~~right coercion~~
		|	~~`%inst` *aty aty*~~	~~instantiation coercion~~
Type	*ty*	→	*bty*	basic type
		|	`%forall` { ~~*tbind*~~ }$^+$. *ty*	type abstraction
		|	*bty* `->` *ty*	arrow type construction
~~Atomic kind~~	~~*akind*~~	→	~~*~~	~~lifted kind~~
		|	~~#~~	~~unlifted kind~~
		|	~~?~~	~~open kind~~
		|	~~*bty :=: bty*~~	~~equality kind~~
		|	~~(*kind*)~~	~~nested kind~~
~~Kind~~	~~*kind*~~	→	~~*akind*~~	~~atomic kind~~
		|	~~*akind -> kind*~~	~~arrow kind~~
Identifier	*mident*	→	*pname* : *uname*	module
	tycon	→	*uname*	type constr.
	qtycon	→	*mident* . *tycon*	qualified type constr.
	tyvar	→	*lname*	type variable
	dcon	→	*uname*	data constr.
	qdcon	→	*mident* . *dcon*	qualified data constr.
	var	→	*lname*	variable
	qvar	→	[*mident* .] *var*	optionally qualified variable
Name	*lname*	→	*lower* { *namechar* }	
	uname	→	*upper* { *namechar* }	
	pname	→	{ *namechar* }$^+$	
	namechar	→	*lower* | *upper* | *digit*	
	lower	→	`a` | `b` | ... | `z` | `_`	
	upper	→	`A` | `B` | ... | `Z`	
	digit	→	`0` | `1` | ... | `9`	

3 Informal Semantics

At the term level, Core resembles a explicitly-typed polymorphic lambda calculus (F_ω), with the addition of local `let` bindings, algebraic type definitions, constructors, and `case` expressions, and primitive types, literals and operators. Its type system is richer than that of System F, supporting explicit type equality coercions and type functions. [Sulzmann et al., 2007]

In this section we concentrate on the less obvious points about Core.

3.1 Program Organization and Modules

Core programs are organized into *modules*, corresponding directly to source-level Haskell modules. Each module has a identifying name *mident*. A module identifier consists of a *package name* followed by a module name, which may be hierarchical: for example, `base:GHC.Base` is the module identifier for GHC's Base module. Its name is `Base`, and it lives in the `GHC`

Now that we understand Core, let's look at some operations in Haskell, and think about how they compile to Core.

Say we want to append an element to the left of a list.

Desire: Input: `1, [2,3]`

Output: `[1,2,3]`

Can do this very easily:

`1 : [2,3]` —> `[1,2,3]`

Why does this work?

":" is an operator (ie, is a function, as everything is).

A list in the form $[x_1, x_2, \ldots, x_n]$ is syntactic sugar for:

$x_1 : (x_2 : (\ldots x_n : [\,]))\ldots)$

So `1 : [2,3]` compiles to `1 : (2 : (3 : [])`, then this becomes:

well...

One very long Core expression. (See next page)

We have a portion of code which shows the way parts of this function compile. (over the page)

We have main8 :: Type.Integer (main8 is of type integer)
main8 is 1.

⋮

":" take in 3 arguments.

: —Compiles→ \@τ->\(s::τ)->\(t::Lτ)->\@α->\(x::α)->\(y::τ->α->α)->ys(t@α xy)

[] —Compiles→ \@α->\(x::α)->\(y::τ->α->α)->x

But this is just:

: —really is→ $\Lambda\tau.\lambda s^{\tau}.\lambda t^{L\tau}\Lambda\alpha\lambda x^{\alpha}.\lambda y^{\tau\to\alpha\to\alpha}.ys(t\alpha xy)$

[] -> $\Lambda\alpha.\lambda x^{\alpha}.\lambda y^{\tau\to\alpha\to\alpha}.x$

Where $L\tau = \forall\alpha.\alpha\to(\tau\to\alpha\to\alpha)\to\alpha$.

System F in disguise!
Ready to β-reduce.

But naively β-reducing may be inefficient...

aws-credentials — ubuntu@ip-172-31-22-16: ~ — ssh -i Virginia.pem ubuntu@limitordinal.org — 89×32

```
ubuntu@ip-172-31-22-16:~$ cat test.hs
main :: IO ()
main = print u
v = [2,3]
u = 1 : v
ubuntu@ip-172-31-22-16:~$ ~/.cabal/bin/ghc-core --no-cast --no-asm test.hs
```

This is Haskell code which will return Core language (see over page)

aws-credentials — ubuntu@ip-172-31-22-16: ~ — ssh -i Virginia.pem ubuntu@limit

```
main8 :: Type.Integer
[GblId,
 Unf=Unf{Src=<vanilla>, TopLvl=True, Arity=0, Value=True,
         ConLike=True, WorkFree=True, Expandable=True,
         Guidance=IF_ARGS [] 100 0}]
main8 = __integer 1

main7 :: Type.Integer
[GblId,
 Unf=Unf{Src=<vanilla>, TopLvl=True, Arity=0, Value=True,
         ConLike=True, WorkFree=True, Expandable=True,
         Guidance=IF_ARGS [] 100 0}]
main7 = __integer 2

main6 :: Type.Integer
[GblId,
 Unf=Unf{Src=<vanilla>, TopLvl=True, Arity=0, Value=True,
         ConLike=True, WorkFree=True, Expandable=True,
         Guidance=IF_ARGS [] 100 0}]
main6 = __integer 3

main5 :: [Type.Integer]
[GblId,
 Unf=Unf{Src=<vanilla>, TopLvl=True, Arity=0, Value=True,
         ConLike=True, WorkFree=False, Expandable=True,
         Guidance=IF_ARGS [] 10 30}]
main5 =
  :
    @ Type.Integer
    main6
    ([] @ Type.Integer)
:
```

main8 is of type integer, as expected.

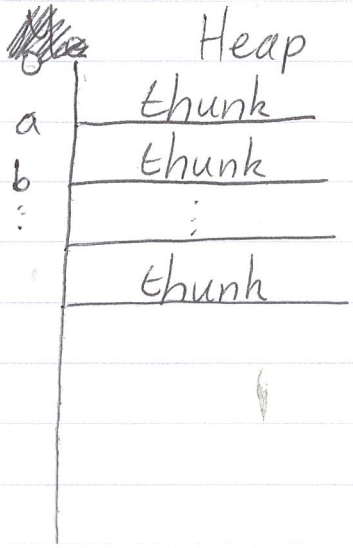
The 3 inputs ":" receives.

What if we had a more complicated program?

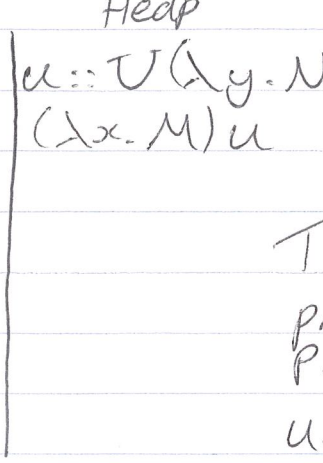
length (2 : [2,2,2])

length is a function which has one inputs, so looks like $\lambda x.M$.
: is a function with 2 inputs, so looks like $\lambda y.N.(N=\lambda z.W)$.
So we have $(\lambda x.M)((\lambda y.N)P)$.

Core is categorized into ~~expressions~~ "expressions" on a "heap".



Each thunk is a collection of modules.
We have pointers from each thunk which declares a variable to where this variable appears in other thunks.
Might have:



This gives 3 choices of how to proceed.
P -> N, result -> M.
u -> M, P -> N, result -> u.
Sub $(\lambda y.M)P \to u$, put this -> M.
Which is best?

Haskell is call-by-name, so we can put a marker in all the places an evaluated function needs to go, then compute the function once, and place the value in all the markers rather than evaluating the function at each marker.
This allows for "lazy evaluation", and even computation of infinite functions if the value of the infinite function is to be placed in 0 positions, then we can discard it.

Other languages (your favourite ~~functional~~ object-oriented languages) are likely to be call-by-value, and so are unable to do this.

So why bother?

The only way forward is to make things simple!

So we can argue about programs.

"Simple Made Easy", Rich Hickey.