Idris is a *pure functional programming language* with *dependent types*

- `cabal update; cabal install idris`
Idris is a pure functional programming language with dependent types

- cabal update; cabal install idris

In these lectures:

1. Today: An introduction to the language
   - Software correctness - why Idris?
   - Language overview; programming idioms; theorem proving

2. Tomorrow: Embedded Domain Specific Languages
   - A well-type interpreter
   - Dealing with effects and resources
IDRIS is a pure functional programming language with dependent types

- cabal update
- http://idris-lang.org

In these lectures:

1. Today: An introduction
   - Software correctness - why IDRIS?
   - Language overview; programming idioms; theorem proving

2. tomorrow: Embedded Domain Specific Languages
   - A well-type interpreter
   - Dealing with effects and resources
Preview quit unexpectedly.

Click Reopen to open the application again. Click Report to see more detailed information and send a report to Apple.

Ignore  Report...  Reopen
Modern programming languages help the programmer by checking that types are used consistently. Several choices are available:

- **Static vs Dynamic**
  - Does the language check for type errors *before* running the program (Static typing) or *while* running the program (Dynamic typing)?

- **Strong vs Weak**
  - Is it an error to violate all type distinctions (Strong typing) or can we have more flexibility (Weak typing; e.g. treating an `Int` as a `Float`)

I am interested in *strong, static* typing.
Idris Overview

- **General purpose** programming language
  - Compiled, supports foreign functions, ...

Influenced by Haskell

Pattern matching, where, ...

Type classes, do-notation, comprehensions, ...

Has full dependent types

Types may be predicated on values

Can encode (and check) program properties

Supports tactic based theorem proving

Support for Embedded Domain Specific Languages

Syntax overloading, dsl notation
Idris Overview

- **General purpose** programming language
  - Compiled, supports foreign functions, ...

- Influenced by *Haskell*
  - Pattern matching, *where*, ...
  - Type classes, *do*-notation, comprehensions, ...

*sicsa*
Idris Overview

- **General purpose** programming language
  - Compiled, supports foreign functions, ...

- Influenced by *Haskell*
  - Pattern matching, *where*, ...
  - Type classes, *do*-notation, comprehensions, ...

- Has **full dependent types**
  - Types may be *predicated* on values
  - Can encode (and check) program *properties*
  - Supports *tactic based* theorem proving
Idris Overview

- **General purpose** programming language
  - Compiled, supports foreign functions, ...
- Influenced by *Haskell*
  - Pattern matching, *where*, ...
  - Type classes, *do*-notation, comprehensions, ...
- Has **full dependent types**
  - Types may be *predicated* on values
  - Can encode (and check) program *properties*
  - Supports *tactic based* theorem proving
- Support for *Embedded Domain Specific Languages*
  - Syntax overloading, *dsl* notation
Unary natural numbers

data Nat = O | S Nat
**Unary natural numbers**

\[
data \text{Nat} = O \mid S \text{Nat}
\]

**Polymorphic lists**

\[
data \text{List} : \text{Type} \to \text{Type} \where
\begin{align*}
\text{Nil} & : \text{List} \ a \\
(::) & : a \to \text{List} \ a \to \text{List} \ a
\end{align*}
\]
### Unary natural numbers

```idris
data Nat = O | S Nat
```

### Polymorphic lists

```idris
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

### Vectors — polymorphic lists with length

```idris
data Vect : Type -> Nat -> Type where
  Nil : Vect a O
  (::) : a -> Vect a k -> Vect a (S k)
```
Dependent Types — Examples

Append

\[
\begin{align*}
++ & : \text{ Vect } a \ m \rightarrow \text{ Vect } a \ n \rightarrow \text{ Vect } a \ (m + n) \\
++ [\ ] & \quad ys = ys \\
++ (x \::\:: xs) & \quad ys = x \::\:: xs ++ ys
\end{align*}
\]
Append

\[ (++) : \text{Vect} \ a \ m \to \text{Vect} \ a \ n \to \text{Vect} \ a \ (m + n) \]
\[ (++) \ [] \ y = y \]
\[ (++) \ (x::xs) \ y = x :: xs ++ y \]

Pairwise addition

vAdd : \text{Num} \ a \Rightarrow \text{Vect} \ a \ n \Rightarrow \text{Vect} \ a \ n \Rightarrow \text{Vect} \ a \ n
vAdd [] [] = []
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
Dependent Types — Examples

Append

(+++) : Vect a m -> Vect a n -> Vect a (m + n)
(+++) [] ys = ys
(+++) (x::xs) ys = x :: xs ++ ys

Pairwise addition

total
vAdd : Num a => Vect a n -> Vect a n -> Vect a n
vAdd [] [] = []
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
Why Dependent Types?

Precise types

\[
\text{sort} : \text{List Int} \rightarrow \text{List Int}
\]
Precise types

\texttt{sort : Vect \text{Int} \ n \rightarrow Vect \text{Int} \ n}
Why Dependent Types?

*Precise types*

```haskell
sort : (xs : Vect Int n) ->
(y : Vect Int n ** Permutation xs y)
```
Why Dependent Types?

We can make types as precise as we require.

- *However*, precise types may require complex implementations/proofs.
In these lectures, we will see examples of:

- Precise types for supporting *machine-checked* proofs of correctness
  - Both functional and extra-functional correctness
In these lectures, we will see examples of:

- *Precise* types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness
- Generic programming
  - First class types, so types can be calculated by programs
Why Dependent Types?

In these lectures, we will see examples of:

- **Precise** types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness
- Generic programming
  - First class types, so types can be calculated by programs
- Expressivity
  - Using types to make more expressive libraries
Why Dependent Types?

In these lectures, we will see examples of:

- **Precise** types for supporting *machine-checked* proofs of correctness
  - Both *functional* and *extra-functional* correctness

- Generic programming
  - First class types, so types can be calculated by programs

- Expressivity
  - Using types to make more expressive libraries
  - (Or: why I will never write a monad transformer again)
There are several dependently typed languages available (e.g. Agda, Coq, Epigram, ...). Why did I make Idris?

- We are still learning about dependent types
  - Plenty of scope for experimenting, still
There are several dependently typed languages available (e.g. Agda, Coq, Epigram, . . .). Why did I make Idris?

- We are still *learning* about dependent types
  - Plenty of scope for experimenting, still
- Freedom to make *language design* decisions
  - e.g. partial functions, high level notation, primitive types, evaluation strategy, . . .
There are several dependently typed languages available (e.g. Agda, Coq, Epigram, . . .). Why did I make Idris?

- We are still *learning* about dependent types
  - Plenty of scope for experimenting, still
- Freedom to make *language design* decisions
  - e.g. partial functions, high level notation, primitive types, evaluation strategy, . . .
- Desire to experiment with *practical* aspects
  - Efficient compilation, Operating system interaction, foreign function calls, . . .
Demonstration: Basic Usage

EVERYTHING IS OKAY

[Smiley face]
By default, Idris uses a strict evaluation strategy. But what about...
By default, **Idris** uses a *strict* evaluation strategy. But what about... 

**Control Structures**

boolElim : Bool -> a -> a -> a

boolElim True  t e = t
boolElim False t e = e

syntax if [test] then [t] else [e]
    = boolElim test t e

foo : t
foo = if expr then largeexpr1 else largeexpr2

Both largeexpr1 and largeexpr2 have to be evaluated in full!
By default, **Idris** uses a *strict* evaluation strategy. But what about... 

### Control Structures with Laziness

**boolElim** : \(\text{Bool} \rightarrow (t : a) \rightarrow (e : a) \rightarrow a\)

- \(\text{boolElim True } t \ e = t\)
- \(\text{boolElim False } t \ e = e\)

**syntax** if \([\text{test}]\) then \([\text{t}]\) else \([\text{e}]\)

\[= \text{boolElim test } t \ e\]

**foo** : \(t\)

**foo** = if \(\text{expr}\) then \(\text{largeexpr1}\) else \(\text{largeexpr2}\)

Now only one of \(\text{largeexpr1}\) and \(\text{largeexpr2}\) will be evaluated, depending which is needed.
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

<table>
<thead>
<tr>
<th>Some proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td>data \text{Or} \ a \ b \ = \ \text{Inl} \ a \</td>
</tr>
<tr>
<td>data \text{And} \ a \ b \ = \ \text{And_intro} \ a \ b</td>
</tr>
</tbody>
</table>
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

### Some proofs

```haskell
data Or a b = Inl a | Inr b
data And a b = And_intro a b

theorem1 : a -> Or a b
theorem1 x = Inl x

theorem2 : a -> b -> And a b
theorem2 x y = And_intro x y
```
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

**Some proofs**

```
data Or a b = Inl a | Inr b
data And a b = And_intro a b

theorem1 : a -> Or a b
theorem1 x =
```
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

```
Some proofs

data Or a b = Inl a | Inr b
data And a b = And_intro a b

theorem1 : a -> Or a b
  theorem1 x = Inl x

theorem2 : a -> b -> And a b
  theorem2 x y = And_intro x y
```
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

### Some proofs

```haskell
data Or a b = Inl a | Inr b

data And a b = And_intro a b

theorem1 : a -> Or a b
theorem1 x = Inl x

theorem2 : a -> b -> And a b
theorem2 x y = And_intro x y
```
Thanks to the *Curry-Howard Correspondence*, we can view a type as a specification, and a program as a proof of a specification, e.g.

### Some proofs

```haskell
data Or a b = Inl a | Inr b
data And a b = And_intro a b

theorem1 : a -> Or a b
theorem1 x = Inl x

theorem2 : a -> b -> And a b
theorem2 x y = And_intro x y
```
### Built-in types

```
data (=) : a -> b -> Type where
  refl : x = x

data _|_ where    {- empty type -}
```
Equality Proofs

**Built-in types**

\[
data \ (\ =) \ : \ a \to \ b \to \ Type \ \text{where} \\
\quad \text{refl} \ : \ x = x
\]

\[
data \ _\ | \ _ \ \text{where} \quad \{- \ \text{empty type} \ -\}
\]

**Rewriting**

\[
\text{replace} \ : \ \{P \ : \ a \to \ Type\} \to x = y \to P x \to P y
\]
Equality Proofs

Example

twoPlusTwo : 2 + 2 = 4
twoPlusTwo = refl
Demonstration: Equality Proofs

(http://xkcd.com/285/)
The with rule

Parity

data Parity : Nat -> Type where
  even : Parity (n + n)
  odd : Parity (S (n + n))

Every number has a parity

parity : (n : Nat) -> Parity n
Demonstration:
Implementing parity
The with rule

Parity

<table>
<thead>
<tr>
<th>data Parity : Nat -&gt; Type where</th>
</tr>
</thead>
<tbody>
<tr>
<td>even : Parity (n + n)</td>
</tr>
<tr>
<td>odd  : Parity (S (n + n))</td>
</tr>
</tbody>
</table>

Every number has a parity

| parity : (n : Nat) -> Parity n |
The with rule

Parity

data Parity : Nat -> Type where
  even : Parity (n + n)
  odd : Parity (S (n + n))

Every number has a parity

parity : (n : Nat) -> Parity n

Demonstration: Implementing parity
Decidable equality

**Type**

```haskell
data Dec : Type -> Type where
  Yes : a -> Dec a
  No : (a -> _|_) -> Dec a
```

**Checking Equality**

```haskell
class DecEq t where
  decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)
```
Decidable equality

**Type**

```haskell
data Dec : Type -> Type where
  Yes : a -> Dec a
  No : (a -> _|_) -> Dec a
```

**Checking Equality**

```haskell
class DecEq t where
decEq : (x1 : t) -> (x2 : t) -> Dec (x1 = x2)
```

A **Bool** tells us the result of an equality test. **Dec** tells us *why*. 
List Membership

data Elem : a -> List a -> Type where
  Here : {xs : List a} -> Elem x (x :: xs)
  There : {xs : List a} -> 
  Elem x xs -> Elem x (y :: xs)
Membership predicates

List Membership

data Elem : a -> List a -> Type where
  Here : {xs : List a} -> Elem x (x :: xs)
  There : {xs : List a} ->
            Elem x xs -> Elem x (y :: xs)

Example

inList : Elem 2 [1,2,3,4]
inList = There Here
Membership predicates

List Membership

data Elem : a -> List a -> Type where
  Here : {xs : List a} -> Elem x (x :: xs)
  There : {xs : List a} ->
    Elem x xs -> Elem x (y :: xs)

Example

inList : Elem 2 [1,2,3,4]
inList = There Here

Demonstration: Building membership predicates