Overview

▶ Today: Look at the Haskell programming language
Today: Look at the Haskell programming language

Like LISP, Haskell is a functional language
Today: Look at the Haskell programming language

Like LISP, Haskell is a functional language

Unlike LISP, Haskell is statically typed and has an expressive type system
Today: Look at the Haskell programming language

Like LISP, Haskell is a functional language

Unlike LISP, Haskell is statically typed and has an expressive type system

Integrates many concepts we looked at: static typing, type inference, polymorphism, higher-order functions, ...
Overview

- **Today:** Look at the Haskell programming language
- Like LISP, Haskell is a functional language
- Unlike LISP, Haskell is statically typed and has an expressive type system
- Integrates many concepts we looked at: static typing, type inference, polymorphism, higher-order functions, . . .
- At the cutting edge of PL research: designed by researchers, lots of new research done in the context of Haskell
Haskell borrows ideas from both LISP and ML
Little Bit of History

- Haskell borrows ideas from both LISP and ML
- Like LISP and ML, functional language
Little Bit of History

- Haskell borrows ideas from both LISP and ML
- Like LISP and ML, functional language
- Like ML, has a strong, expressive static type system
Little Bit of History

- Haskell borrows ideas from both LISP and ML
- Like LISP and ML, functional language
- Like ML, has a strong, expressive static type system
- But also different from both in some ways: e.g., lazy functional language
History of Haskell

- Designed in the early 90’s by a committee of researchers
History of Haskell

- Designed in the early 90’s by a committee of researchers
- Goal was to unify research efforts in pure, lazy functional languages
History of Haskell

- Designed in the early 90’s by a committee of researchers
- Goal was to unify research efforts in pure, lazy functional languages
- Feeling was that widespread use of lazy functional languages was being hampered by lack of standard language
History of Haskell

- Designed in the early 90’s by a committee of researchers
- Goal was to unify research efforts in pure, lazy functional languages
- Feeling was that widespread use of lazy functional languages was being hampered by lack of standard language
- Goal of the committee was to design a lazy functional language that would become standard
History of Haskell

- Designed in the early 90’s by a committee of researchers
- Goal was to unify research efforts in pure, lazy functional languages
- Feeling was that widespread use of lazy functional languages was being hampered by lack of standard language
- Goal of the committee was to design a lazy functional language that would become standard
- People who played key role in designing Haskell: Simon Peyton Jones (Microsoft Research), Philip Wadler (Edinburgh), Paul Hudak (Yale), John Hughes (Chalmers)
Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry

The Haskell programming language is named after logician Haskell B. Curry.

Haskell curry well-known for Curry-Howard isomorphism

Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages

Specifically, it says:

1. Propositions are types
2. Proofs are programs
3. Proof checking is type checking
Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry

- Haskell curry well-known for Curry-Howard isomorphism

Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages

- Propositions are types
- Proofs are programs
- Proof checking is type checking
Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry

- Haskell curry well-known for Curry-Howard isomorphism

- Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages
Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry

- Haskell curry well-known for Curry-Howard isomorphism

- Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages

- Specifically, it says:

1. Propositions are types
2. Proofs are programs
3. Proof checking is type checking
Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry

- Haskell curry well-known for Curry-Howard isomorphism

- Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages

- Specifically, it says:

  1. Propositions are types
Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry

- Haskell curry well-known for Curry-Howard isomorphism

- Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages

Specifically, it says:

1. Propositions are types

2. Proofs are programs
Why Called Haskell?

The Haskell programming language is named after logician Haskell B. Curry

Haskell curry well-known for Curry-Howard isomorphism

Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages

Specifically, it says:

1. Propositions are types
2. Proofs are programs
3. Proof checking is type checking
Overview of Haskell Features

- Lazy evaluation
Overview of Haskell Features

- Lazy evaluation
- Expressive static type system
Overview of Haskell Features

- Lazy evaluation
- Expressive static type system
- Polymorphism and type classes
Overview of Haskell Features

- Lazy evaluation
- Expressive static type system
- Polymorphism and type classes
- Tagged unions, type constructors
Overview of Haskell Features

- Lazy evaluation
- Expressive static type system
- Polymorphism and type classes
- Tagged unions, type constructors
- Pattern matching
Overview of Haskell Features

- Lazy evaluation
- Expressive static type system
- Polymorphism and type classes
- Tagged unions, type constructors
- Pattern matching
- List comprehension
Lazy vs. Strict Programming Languages

- Haskell is different from many other functional languages in that it’s lazy.
Lazy vs. Strict Programming Languages

- Haskell is different from many other functional languages in that it’s lazy

- Lazy (or call-by-need, non-strict) languages delay evaluation of an expression until its value is actually required
Lazy vs. Strict Programming Languages

▶ Haskell is different from many other functional languages in that it’s lazy

▶ Lazy (or call-by-need, non-strict) languages delay evaluation of an expression until its value is actually required

▶ In contrast, eager (or strict, call-by-value) languages, an expression is evaluated as soon as it is bound to a variable, regardless of whether the variable is used or not
Lazy vs. Strict Programming Languages

- Haskell is different from many other functional languages in that it’s lazy.

- Lazy (or call-by-need, non-strict) languages delay evaluation of an expression until its value is actually required.

- In contrast, eager (or strict, call-by-value) languages, an expression is evaluated as soon as it is bound to a variable, regardless of whether the variable is used or not.

- The default behavior in many languages used today is eager evaluation: Java, C, C++, ML, Python, Lisp, Scheme . . .
Example of Lazy vs. Eager

▶ Suppose you pass a divergent (i.e., non-terminating) expression as an argument to a function
Example of Lazy vs. Eager

- Suppose you pass a divergent (i.e., non-terminating) expression as an argument to a function

- In an eager language, the program will definitely not terminate
Suppose you pass a divergent (i.e., non-terminating) expression as an argument to a function.

In an eager language, the program will definitely not terminate.

In a lazy language, the program will terminate if the value of that argument is not needed in the called function.
Example of Lazy vs. Eager

- Suppose you pass a divergent (i.e., non-terminating) expression as an argument to a function.
- In an eager language, the program will definitely not terminate.
- In a lazy language, the program will terminate if the value of that argument is not needed in the called function.
- There are both advantages and disadvantages of lazy evaluation.
Advantages and Disadvantages of Lazy Evaluation

Advantages of Lazy Evaluation:

▶ Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive

▶ Can avoid divergent or buggy behavior in some cases

▶ Can create infinite data structures

Disadvantages of Lazy Evaluation:

▶ Can be difficult to predict program behavior

▶ Doesn’t work well in the presence of side effects

▶ Even in purely functional languages, things like IO become much more difficult
Advantages and Disadvantages of Lazy Evaluation

Advantages of Lazy Evaluation:

- Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive
- Can avoid divergent or buggy behavior in some cases
Advantages of Lazy Evaluation:

- Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive
- Can avoid divergent or buggy behavior in some cases
- Can create infinite data structures
Advantages and Disadvantages of Lazy Evaluation

Advantages of Lazy Evaluation:

▶ Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive

▶ Can avoid divergent or buggy behavior in some cases

▶ Can create infinite data structures

Disadvantages of Lazy Evaluation:

▶ Can be difficult to predict program behavior
Advantages of Lazy Evaluation:

▶ Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive

▶ Can avoid divergent or buggy behavior in some cases

▶ Can create infinite data structures

Disadvantages of Lazy Evaluation:

▶ Can be difficult to predict program behavior

▶ Doesn’t work well in the presence of side effects
Advantages and Disadvantages of Lazy Evaluation

Advantages of Lazy Evaluation:
▶ Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive
▶ Can avoid divergent or buggy behavior in some cases
▶ Can create infinite data structures

Disadvantages of Lazy Evaluation:
▶ Can be difficult to predict program behavior
▶ Doesn’t work well in the presence of side effects
▶ Even in purely functional languages, things like IO become much more difficult
Expressive Static Type System

- Haskell is a statically typed language with a sound type system.
Expressive Static Type System

- Haskell is a statically typed language with a sound type system
- Does not allow ways to subvert the type system, such as casting
Expressive Static Type System

- Haskell is a statically typed language with a sound type system
- Does not allow ways to subvert the type system, such as casting
- Thus, if the Haskell compiler assigns type $T$ to expression $e$, run-time value of $e$ will be in the set of values defined by $T$
Expressive Static Type System

- Haskell is a statically typed language with a sound type system
- Does not allow ways to subvert the type system, such as casting
- Thus, if the Haskell compiler assigns type $T$ to expression $e$, run-time value of $e$ will be in the set of values defined by $T$
- Haskell performs type inference, so you don’t have to explicitly annotate types of expressions
Expressive Static Type System

- Haskell is a statically typed language with a sound type system
- Does not allow ways to subvert the type system, such as casting
- Thus, if the Haskell compiler assigns type $T$ to expression $e$, run-time value of $e$ will be in the set of values defined by $T$
- Haskell performs type inference, so you don’t have to explicitly annotate types of expressions
- Furthermore, Haskell has polymorphism: Types of variables can contain (universally-quantified) type variables
Example of Polymorphism in Haskell

➤ Consider the id function in Haskell: \texttt{let id x = x}
Example of Polymorphism in Haskell

- Consider the id function in Haskell: \texttt{let id x = x}
- This function has the inferred polymorphic type \texttt{a -> a}
Example of Polymorphism in Haskell

Consider the id function in Haskell:

\[
\text{let } \text{id } x = x
\]

This function has the inferred polymorphic type \( a \rightarrow a \)

For any type \( a \), if the value of the input is \( a \), the output is also of type \( a \)
Example of Polymorphism in Haskell

- Consider the `id` function in Haskell: `let id x = x`
- This function has the inferred polymorphic type `a -> a`
- For any type `a`, if the value of the input is `a`, the output is also of type `a`
- The Haskell type `a -> a` is the same as $\forall \alpha. \alpha \to \alpha$
Let’s look at a more interesting example:

\[
\text{map } f \ [ ] = [ ] \\
\text{map } f \ (x:xs) = f \ x : \ \text{map} \ f \ xs
\]
Another Example

- Let’s look at a more interesting example:
  
  ```haskell
  map f [] = []
  map f (x:xs) = f x : map f xs
  ```

- Here, we define a higher-order `map` function which applies a function `f` to every element in the list
Another Example

Let’s look at a more interesting example:

\[
\begin{align*}
\text{map } f \ [ & ] = [ ] \\
\text{map } f \ (x:xs) & = f \ x : \ \text{map } f \ xs
\end{align*}
\]

Here, we define a higher-order \texttt{map} function which applies a function \texttt{f} to every element in the list.

This function definition illustrates \texttt{pattern matching} in Haskell: here, we pattern match on the second argument.
Another Example

- Let’s look at a more interesting example:

  \[
  \text{map } f \ [\] = [] \\
  \text{map } f \ (x:xs) = f \ x : \text{ map } f \ xs
  \]

- Here, we define a higher-order \texttt{map} function which applies a function \( f \) to every element in the list

- This function definition illustrates \texttt{pattern matching} in Haskell: here, we pattern match on the second argument

- First line: if the input list is empty, then return the empty list
Another Example

Let’s look at a more interesting example:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Here, we define a higher-order `map` function which applies a function `f` to every element in the list.

This function definition illustrates pattern matching in Haskell: here, we pattern match on the second argument.

First line: if the input list is empty, then return the empty list.

Second line corresponds to case where list has at least one element.
map f [] = []
map f (x:xs) = f x : map f xs

- By using the notation \((x:xs)\), we bind first element of input list to \(x\) and rest of the list to \(xs\)
Example, cont.

map f [] = []
map f (x:xs) = f x : map f xs

- By using the notation \((x:xs)\), we bind first element of input list to \(x\) and rest of the list to \(xs\)
- Second line: Applies function \(f\) to head \(x\) of the list
Example, cont.

\[
\text{map } f \; [] = [] \\
\text{map } f \; (x:xs) = f \; x : \; \text{map } f \; xs
\]

- By using the notation \((x:xs)\), we bind first element of input list to \(x\) and rest of the list to \(xs\)
- Second line: Applies function \(f\) to head \(x\) of the list
- Then, recursively invokes \(\text{map}\) with function \(f\) on the remainder \(xs\) of the list
map \ f \ [] \ = \ []
map \ f \ (x:xs) \ = \ f \ x \ : \ map \ f \ xs

- By using the notation \ (x:xs) \, we bind first element of input list to \ x \ and rest of the list to \ xs
- Second line: Applies function \ f \ to head \ x \ of the list
- Then, recursively invokes map with function \ f \ on the remainder \ xs \ of the list
- Finally, returns a list which is the concatenation of \ f \ x \ and result of recursive call
Example, cont.

map f [] = []
map f (x:xs) = f x : map f xs

- Haskell compiler infers type of this function to be:
  \((a \to b) \to [a] \to [b]\)
Example, cont.

\[
\text{map } f \; [] = []
\]

\[
\text{map } f \; (x:xs) = f \; x : \; \text{map } f \; xs
\]

- Haskell compiler infers type of this function to be:

\[
(a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

- Again, this is a polymorphic type
map f [] = []
map f (x:xs) = f x : map f xs

▶ Haskell compiler infers type of this function to be:

\((a \rightarrow b) \rightarrow [a] \rightarrow [b]\)

▶ Again, this is a polymorphic type

▶ Says: Given any function of type \(a \rightarrow b\) and a list of values of type \(a\), return value is a list of values of type \(b\)
Example, cont.

\[
\begin{align*}
\text{map } f \; [] & = [] \\
\text{map } f \; (x:xs) & = f \; x : \; \text{map } f \; xs
\end{align*}
\]

- Haskell compiler infers type of this function to be:
  \[(a \to b) \to [a] \to [b]\]

- Again, this is a polymorphic type

- Says: Given any function of type \(a \to b\) and a list of values of type \(a\), return value is a list of values of type \(b\)

- Here, \(a\) and \(b\) are (implicitly) universally-quantified type variables
Type Classes

Consider the Haskell function: \( f \ x = x \times 2 \)
Consider the Haskell function: \( f \ x = x \times 2 \)

What is the type of this function?
Consider the Haskell function: \( f \ x = x \times 2 \)

What is the type of this function?

We can’t give it the polymorphic type \( \text{a} \rightarrow \text{a} \) because it only works on types for which multiplication is defined.
Consider the Haskell function: \( f \ x = x \times 2 \)

What is the type of this function?

We can’t give it the polymorphic type \( a \rightarrow a \) because it only works on types for which multiplication is defined.

But assigning type \( \text{Integer} \rightarrow \text{Integer} \) also too restrictive because works for floats or other types for which \( * \) is defined.
Consider the Haskell function: \( f \ x = x \times 2 \)

What is the type of this function?

We can’t give it the polymorphic type \( a \rightarrow a \) because it only works on types for which multiplication is defined.

But assigning type \( \text{Integer} \rightarrow \text{Integer} \) also too restrictive because works for floats or other types for which \( \times \) is defined.

Haskell solves this problem through type classes.
The type of the function \( f \ x = x \times 2 \) in Haskell is actually:

\[(\text{Num} \ a) \Rightarrow a \rightarrow a\]
The type of the function \( f \ x = x \times 2 \) in Haskell is actually:

\[(\text{Num } a) \Rightarrow a \rightarrow a\]

Says: For any type \( a \) that belongs to type class \( \text{Num} \), function \( f \) takes a value of type \( a \) and return a value of type \( a \).
The type of the function $f \ x = x \times 2$ in Haskell is actually:

$$(\text{Num } a) \Rightarrow a \rightarrow a$$

Says: For any type $a$ that belongs to type class $\text{Num}$, function $f$ takes a value of type $a$ and return a value of type $a$

Since integers, floats etc. all belong to the $\text{Num}$ type class, this type is very general!
Programming with Type Classes

- To use this feature of Haskell, first need to declare type classes

Example:
```haskell
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
...```

This declares a type class called `Num` that supports operations `+` and `*`.
Programming with Type Classes

▶ To use this feature of Haskell, first need to declare type classes

▶ A type class is simply a set of types that support a common set of operations

Example:

```haskell
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  ...
```

This declares a type class called `Num` that supports operations `+` and `*`. 
Programming with Type Classes

- To use this feature of Haskell, first need to declare type classes

- A type class is simply a set of types that support a common set of operations

- To define the type class, declare the common operations and give type class a name

Example:

```haskell
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
...```

This declares a type class called `Num` that supports operations `+` and `*`
Programming with Type Classes

- To use this feature of Haskell, first need to declare type classes

- A type class is simply a set of types that support a common set of operations

- To define the type class, declare the common operations and give type class a name

Example:

```haskell
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
...
```
Programming with Type Classes

- To use this feature of Haskell, first need to declare type classes

- A type class is simply a set of types that support a common set of operations

- To define the type class, declare the common operations and give type class a name

Example:

```haskell
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
...
```

- This declares a type class called Num that supports operations + and *
Then, we declare instances of a type class

Example:

```haskell
instance Num Int where
  a + b = intPlus a b
  a * b = intTimes a b
...
```

Instance declarations show how `Num` operations are implemented for a particular type.

Using type class and instance declarations, Haskell can infer general and useful types such as:

```
(Num a) => a -> a
```
Then, we declare **instances** of a type class

**Example:**

```haskell
instance Num Int where
a+b = intPlus a b
a*b = intTimes a b
...
```
Then, we declare instances of a type class

Example:

```haskell
instance Num Int where
  a + b = intPlus a b
  a * b = intTimes a b
...
```

Instance declarations show how Num operations are implemented for a particular type.
Then, we declare instances of a type class

Example:

```
instance Num Int where
    a+b = intPlus a b
    a*b = intTimes a b
...
```

Instance declarations show how `Num` operations are implemented for a particular type

Using type class and instance declarations, Haskell can infer general and useful types such as:

```
(Num a) => a -> a
```
Another useful feature of Haskell is algebraic data types, also called tagged unions or discriminated unions.
Another useful feature of Haskell is algebraic data types, also called tagged unions or discriminated unions.

Suppose we want to have a type to represent students.
Another useful feature of Haskell is algebraic data types, also called tagged unions or discriminated unions.

Suppose we want to have a type to represent students.

For undergraduates, we want to track their name and department.
Another useful feature of Haskell is algebraic data types, also called tagged unions or discriminated unions.

Suppose we want to have a type to represent students.

For undergraduates, we want to track their name and department.

For masters students, we want to track their name, department, and specialty.
Another useful feature of Haskell is algebraic data types, also called tagged unions or discriminated unions.

Suppose we want to have a type to represent students.

For undergraduates, we want to track their name and department.

For masters students, we want to track their name, department, and specialty.

In Haskell, you can have a data type to represent both undergraduate and masters students:
Tagged Union Example

\[
data \text{Student} = \text{BS} (\text{Name, Dept}) | \text{MS} (\text{Name, Dept, Area})
\]

Here, \textbf{BS} and \textbf{MS} are constructors
Tagged Union Example

data Student = BS (Name, Dept) | MS (Name, Dept, Area)

- Here, BS and MS are constructors

- The BS constructor must be applied to a pair consisting of a name and department
Tagged Union Example

```haskell
data Student = BS (Name, Dept) | MS (Name, Dept, Area)
```

- Here, **BS** and **MS** are constructors

- The **BS** constructor must be applied to a pair consisting of a name and department

- The **MS** constructor must be applied to tuple consisting of name, department, and specialty
Tagged Union Example

```haskell
data Student = BS (Name, Dept) | MS (Name, Dept, Area)
```

- Here, `BS` and `MS` are constructors.
- The `BS` constructor must be applied to a pair consisting of a name and department.
- The `MS` constructor must be applied to tuple consisting of name, department, and specialty.
- Thus, `Student` is really a union of the two types `Name*Dept` and `Name*Dept*Area`. 
data Student = BS (Name, Dept) | MS (Name, Dept, Area)

- Here, \textbf{BS} and \textbf{MS} are constructors.

- The \textbf{BS} constructor must be applied to a pair consisting of a name and department.

- The \textbf{MS} constructor must be applied to tuple consisting of name, department, and specialty.

- Thus, \textbf{Student} is really a union of the two types \texttt{Name*Dept} and \texttt{Name*Dept*Area}.

- Here, \texttt{A*B} is a \textbf{product type} which corresponds to type of tuple whose elements are of type \texttt{A} and \texttt{B}.
Pattern Matching on Data Types

Now, suppose we want to write a \texttt{name} function that gives the name of any student.

We can conveniently do this in Haskell through pattern-matching:

\[
\text{name (BS}(n, d)\text{)} = n \\
\text{name (MS}(n, d, a)\text{)} = n
\]

Haskell compiler will infer type of \texttt{name} as \texttt{Student -> Name}.

This is another example of pattern matching in Haskell.

First line matches on students with type constructor \texttt{BS}.

Second line matches on students with type constructor \texttt{MS}.
Pattern Matching on Data Types

- Now, suppose we want to write a `name` function that gives the name of any student

- We can conveniently do this in Haskell through pattern-matching:

  ```haskell
  name (BS(n,d)) = n
  name (MS(n,d, a)) = n
  ```
Pattern Matching on Data Types

- Now, suppose we want to write a `name` function that gives the name of any student

- We can conveniently do this in Haskell through pattern-matching:
  
  ```haskell
  name (BS(n,d)) = n
  name (MS(n,d, a)) = n
  ```

- Haskell compiler will infer type of `name` as `Student -> Name`
Pattern Matching on Data Types

▶ Now, suppose we want to write a `name` function that gives the name of any student

▶ We can conveniently do this in Haskell through pattern-matching:

```haskell
name (BS(n,d)) = n
name (MS(n,d, a)) = n
```

▶ Haskell compiler will infer type of `name` as `Student -> Name`

▶ This is another example of pattern matching in Haskell
Pattern Matching on Data Types

Now, suppose we want to write a `name` function that gives the name of any student

We can conveniently do this in Haskell through pattern-matching:

\[
\text{name } (\text{BS}(n,d)) = n \\
\text{name } (\text{MS}(n,d, a)) = n
\]

Haskell compiler will infer type of `name` as \text{Student} \rightarrow \text{Name}

This is another example of pattern matching in Haskell

First line matches on students with type constructor BS
Pattern Matching on Data Types

- Now, suppose we want to write a function that gives the name of any student

- We can conveniently do this in Haskell through pattern-matching:

  \[ \text{name} \ (\text{BS}(n,d)) = n \]
  \[ \text{name} \ (\text{MS}(n,d, a)) = n \]

- Haskell compiler will infer type of \text{name} as \text{Student} $\rightarrow$ Name

- This is another example of pattern matching in Haskell

- First line matches on students with type constructor BS

- Second line matches on students with type constructor MS
Recursive Data Types

- Data types in Haskell can also be recursive

Example:

```
data Tree = Leaf Int | Node (Tree, Tree)
```

Here is how you would write a function in Haskell that checks if a given integer is in the tree:

```
inTree x (Leaf y) = x == y
inTree x (Node(y,z)) = inTree x y || inTree x z
```
Recursive Data Types

- Data types in Haskell can also be recursive

- i.e., data type being declared can appear as arguments of type constructors

  Example:
  ```haskell
data Tree = Leaf Int | Node (Tree, Tree)
```

  Here is how you would write a function in Haskell that checks if a given integer is in the tree:
  ```haskell
inTree x (Leaf y) = x==y
inTree x (Node(y,z)) = inTree x y || inTree x z
  ```
Recursive Data Types

- Data types in Haskell can also be recursive
- i.e., data type being declared can appear as arguments of type constructors

Example:

```haskell
data Tree = Leaf Int | Node (Tree, Tree)
```

Here is how you would write a function in Haskell that checks if a given integer is in the tree:

```haskell
inTree x (Leaf y) = x==y
inTree x (Node(y,z)) = inTree x y || inTree x z
```
Recursive Data Types

- Data types in Haskell can also be recursive
- i.e., data type being declared can appear as arguments of type constructors

Example:

```haskell
data Tree = Leaf Int | Node (Tree, Tree)
```

Here is how you would write a function in Haskell that checks if a given integer is in the tree:

```haskell
inTree x (Leaf y) = x == y
inTree x (Node(y,z)) = inTree x y || inTree x z
```
List Comprehension

- Haskell has another very convenient feature called **list comprehension**

**Example:**

Let's consider a simple example. Suppose we want to create a list of all even numbers in a given list. In Haskell, we can define such a list using list comprehension as follows:

```
let evenNumbers = [x | x <- [1..10], x `mod` 2 == 0]
```

This code creates a list `evenNumbers` that contains all the even numbers from 1 to 10. The `|` symbol in list comprehension is similar to the `|` symbol in set-builder notation, which is used to specify conditions on elements in the resulting list. In this case, it specifies that an element `x` should be included in the list if `x` is even (i.e., `x mod 2 == 0`).
List Comprehension

- Haskell has another very convenient feature called list comprehension
- List comprehension allows conveniently building data structures from existing data structures
List Comprehension

- Haskell has another very convenient feature called list comprehension

- List comprehension allows conveniently building data structures from existing data structures

- List comprehension is similar to set-builder notation in math:

$$E = \{ x | x \in A, x \% 2 = 0 \}$$
List Comprehension

- Haskell has another very convenient feature called **list comprehension**.
- List comprehension allows conveniently building data structures from existing data structures.
- List comprehension is similar to **set-builder notation** in math:

  \[ E = \{ x | x \in A, x \% 2 = 0 \} \]

  This says set \( E \) contains all even numbers in set \( A \), i.e., build new set from existing set.
List Comprehension

- Haskell has another very convenient feature called list comprehension

- List comprehension allows conveniently building data structures from existing data structures

- List comprehension is similar to set-builder notation in math:

  \[ E = \{ x \mid x \in A, x \% 2 = 0 \} \]

- This says set \( E \) contains all even numbers in set \( A \), i.e., build new set from existing set

- List comprehension in Haskell has similar notation and is very convenient
Here is an example of list comprehension in Haskell:

```
myList = [1,2,3,4,5,6,7,8]
twiceMyList = [2*x | x<-myList]
```
List Comprehension Example

- Here is an example of list comprehension in Haskell:

  \[
  \text{myList} = [1,2,3,4,5,6,7,8] \\
  \text{twiceMyList} = [2 \times x \mid x \leftarrow \text{myList}]
  \]

- This code creates new list `twiceMyList` by going over every element in `myList` and multiplying it by two

- Another example:

  \[
  \text{mysteryList} = [(x,y) \mid x \leftarrow \text{list1}, y \leftarrow \text{list2}]
  \]

- What does this code snippet do?

  takes cross product of two lists
List Comprehension Example

▶ Here is an example of list comprehension in Haskell:

\[
\text{myList} = [1,2,3,4,5,6,7,8] \\
\text{twiceMyList} = [2*x \mid x \leftarrow \text{myList}]
\]

▶ This code creates new list \texttt{twiceMyList} by going over every element in \texttt{myList} and multiplying it by two

▶ Just syntactic sugar, but very convenient
List Comprehension Example

- Here is an example of list comprehension in Haskell:
  
  ```haskell
  myList = [1,2,3,4,5,6,7,8]
  twiceMyList = [2*x | x<-myList]
  ```

- This code creates new list `twiceMyList` by going over every element in `myList` and multiplying it by two

- Just syntactic sugar, but very convenient

- Another example:
  
  ```haskell
  mysteryList = [ (x,y) | x<-list1, y<-list2]
  ```
List Comprehension Example

- Here is an example of list comprehension in Haskell:
  
  ```haskell
  myList = [1,2,3,4,5,6,7,8]
  twiceMyList = [2*x | x<-myList]
  ```

- This code creates new list `twiceMyList` by going over every element in `myList` and multiplying it by two

- Just syntactic sugar, but very convenient

- Another example:
  
  ```haskell
  mysteryList = [ (x,y) | x<-list1, y<-list2]
  ```

- What does this code snippet do?
Here is an example of list comprehension in Haskell:

\[
\text{myList} = [1,2,3,4,5,6,7,8] \\
\text{twiceMyList} = [2 \times x \mid x \leftarrow \text{myList}]
\]

This code creates new list \text{twiceMyList} by going over every element in \text{myList} and multiplying it by two.

Just syntactic sugar, but very convenient.

Another example:

\[
\text{mysteryList} = [(x,y) \mid x \leftarrow \text{list1}, y \leftarrow \text{list2}]
\]

What does this code snippet do? takes cross product of two lists.
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax

Here is an anonymous Haskell function that adds one to its argument:

```
x -> x + 1
```

Anonymous functions especially useful for simple functions passed as arguments to higher order functions

Example:

```
map (x -> x + 1) myList
```

Here, argument to `map` is an anonymous function that adds one to its argument

Result of this expression is a list where every element is one greater than corresponding element in `myList`
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax

- Here is an anonymous Haskell function that adds one to its argument: \[ x \rightarrow x + 1 \]

- Anonymous functions especially useful for simple functions passed as arguments to higher order functions

- Example:
  \[ \text{map } (x \rightarrow x + 1) \text{ myList} \]

  Here, argument to \text{map} is an anonymous function that adds one to its argument

  Result of this expression is a list where every element is one greater than corresponding element in \text{myList}
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax

- Here is an anonymous Haskell function that adds one to its argument: \( x \rightarrow x+1 \)

- Anonymous functions especially useful for simple functions passed as arguments to higher order functions
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax

- Here is an anonymous Haskell function that adds one to its argument: \( x \rightarrow x + 1 \)

- Anonymous functions especially useful for simple functions passed as arguments to higher order functions

- Example: ```map (\x \rightarrow x+1) myList```
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax

- Here is an anonymous Haskell function that adds one to its argument: \( \ x \rightarrow x+1 \)

- Anonymous functions especially useful for simple functions passed as arguments to higher order functions

- Example: \( \text{map (\ x \rightarrow x+1) myList} \)

- Here, argument to map is an anonymous function that adds one to its argument
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax
  \[ x \rightarrow x + 1 \]

- Anonymous functions especially useful for simple functions passed as arguments to higher order functions

- Example: \( \text{map} (\rightarrow x + 1) \text{ myList} \)

  Here, argument to \( \text{map} \) is an anonymous function that adds one to its argument

  Result of this expression is a list where every element is one greater than corresponding element in \text{myList}
Now that we are more familiar with Haskell syntax, let’s revisit lazy evaluation in Haskell.

Consider the following function `magic`:

- `magic 0 _ = []`
- `magic m n = m : (magic n (m+n))`

What is `magic 1 1`?

The list of Fibonacci numbers: 

`[1,1,2,3,5,8, ...]`

Clearly, `magic 1 1` does not terminate since this list is infinite.
Revisiting Lazy Evaluation in Haskell

- Now that we are more familiar with Haskell syntax, let’s revisit lazy evaluation in Haskell.

- Consider the following function `magic`:

  ```haskell
  magic 0 _ = []
magic m n = m : (magic n (m+n))
  ```
Now that we are more familiar with Haskell syntax, let's revisit lazy evaluation is Haskell

Consider the following function `magic`:

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
```

What is `magic 1 1`?
Revisiting Lazy Evaluation in Haskell

▶ Now that we are more familiar with Haskell syntax, let’s revisit lazy evaluation in Haskell

▶ Consider the following function \texttt{magic}:

\[
\begin{align*}
\text{magic} \ 0 \ _ & = [] \\
\text{magic} \ m \ n & = m : (\text{magic} \ n \ (m+n))
\end{align*}
\]

▶ What is \texttt{magic 1 1}?

▶ The list of Fibonacci numbers: \([1,1,2,3,5,8, \ldots]\)
Now that we are more familiar with Haskell syntax, let’s revisit lazy evaluation in Haskell.

Consider the following function \texttt{magic}:

\begin{verbatim}
magic 0 _ = []
magic m n = m : (magic n (m+n))
\end{verbatim}

What is \texttt{magic 1 1}?

The list of Fibonacci numbers: \([1,1,2,3,5,8, \ldots]\)

Clearly, \texttt{magic 1 1} does not terminate since this list is infinite.
Lazy Evaluation in Haskell Example

Now, let’s write a function to get n’th element from a list

```
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```
Lazy Evaluation in Haskell Example

- Now, let’s write a function to get n’th element from a list

  get_nth [] _ = 0
  get_nth (x:xs) 1 = x
  get_nth (x:xs) n = get_nth xs (n-1)

- Will \texttt{get\_nth (magic 1 1) 3} terminate?
Now, let’s write a function to get n’th element from a list

\[
\begin{align*}
\text{get\_nth} & \; \; [\;] \; \; _\; \; \; \; = \; \; 0 \\
\text{get\_nth} & \; \; (x:xs) \; \; 1 \; \; = \; \; x \\
\text{get\_nth} & \; \; (x:xs) \; \; n \; \; = \; \; \text{get\_nth} \; \; xs \; \; (n-1)
\end{align*}
\]

Will \text{get\_nth} (\text{magic} \; 1 \; 1) \; 3 \; \text{terminate}?

Yes – let’s see why
Lazy Evaluation in Haskell Example

- Now, let’s write a function to get n’th element from a list
  
  \[
  \begin{align*}
  \text{get\_nth} \; \text{[]} \; \_ & = 0 \\
  \text{get\_nth} \; (x:xs) \; 1 & = x \\
  \text{get\_nth} \; (x:xs) \; n & = \text{get\_nth} \; xs \; (n-1)
  \end{align*}
  \]

- Will \text{get\_nth (magic 1 1)} \; 3 terminate?

- Yes – let’s see why

- First, recall that since Haskell is lazy, \text{magic 1 1} will not be evaluated until it is needed in \text{get\_nth}
Lazy Evaluation in Haskell Example

- Now, let’s write a function to get n’th element from a list

  \[
  \text{get\_nth\ }[\ ]\ _ = 0 \\
  \text{get\_nth\ }(x:xs)\ 1 = x \\
  \text{get\_nth\ }(x:xs)\ n = \text{get\_nth\ }xs\ (n-1)
  \]

- Will \text{get\_nth\ } (\text{magic\ 1\ 1})\ 3 \text{ terminate?}

- Yes – let’s see why

- First, recall that since Haskell is lazy, \text{magic\ 1\ 1} will not be evaluated until it is needed in \text{get\_nth}

- In \text{get\_nth}, we need to figure out which pattern is matched
Now, let’s write a function to get n’th element from a list

\[
\begin{align*}
\text{get\_nth \ [\] \_} &= 0 \\
\text{get\_nth \ (x:xs) \ 1} &= x \\
\text{get\_nth \ (x:xs) \ n} &= \text{get\_nth \ xs \ (n-1)}
\end{align*}
\]

Will \text{get\_nth \ (magic \ 1 \ 1) \ 3} terminate?

Yes – let’s see why

First, recall that since Haskell is lazy, \text{magic \ 1 \ 1} will not be evaluated until it is needed in \text{get\_nth}

In \text{get\_nth}, we need to figure out which pattern is matched

This forces one step in the evaluation of \text{magic \ 1 \ 1}
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

- After one step in evaluation of `magic 1 1`, we get: 
  \[
  1 : (magic 1 (1+1))
  \]
Example, cont.

magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)

▶ After one step in evaluation of \texttt{magic 1 1}, we get:
\[
1 : (\texttt{magic 1 (1+1)})
\]

▶ Now, in \texttt{get_nth}, we match on the third case (since second argument is 3)
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

- After one step in evaluation of `magic 1 1`, we get: `1 : (magic 1 (1+1))`

- Now, in `get_nth`, we match on the third case (since second argument is 3)

- Thus, we now evaluate `get_nth (magic 1 (1+1)) (3-1)`
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

- After one step in evaluation of `magic 1 1`, we get: `1 : (magic 1 (1+1))`

- Now, in `get_nth`, we match on the third case (since second argument is 3)

- Thus, we now evaluate `get_nth (magic 1 (1+1)) (3-1)`

- Now, again, we need to figure out which pattern matches; this forces one more evaluation step
Example, cont.

```
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

▶ Thus, after one step in evaluation of \((\text{magic } 1 \ (1+1))\), we get:  
\[1:\text{(magic} \ (1+1) \ (1+(1+1)))\]
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

Thus, after one step in evaluation of \( \text{magic } 1 \ (1+1) \), we get:

\[
1 : (\text{magic } (1+1) \ (1+(1+1)))
\]

Thus, our expression is now:

\[
\text{get_nth } (1 : (\text{magic } (1+1) \ (1+(1+1)))) \ (3-1)
\]
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

▶ Thus, after one step in evaluation of \((\text{magic } 1 (1+1))\), we get: \(1 : (\text{magic } (1+1) (1+(1+1)))\)

▶ Thus, our expression is now:
\[
get\_nth \left(1 : (\text{magic } (1+1) (1+(1+1)))\right) (3-1)
\]

▶ Now, to figure out if we match second or third case, we evaluate \(3-1\):
\[
get\_nth \left(1 : (\text{magic } (1+1) (1+(1+1)))\right) (2)
\]
Example, cont.

\[
\begin{align*}
magic \ 0 \ _\ & = \ [] \\
magic \ m \ n & = \ m : (\text{magic n} \ (m+n)) \\
get\_nth \ [] \ _ & = \ 0 \\
get\_nth \ (x:xs) \ 1 & = x \\
get\_nth \ (x:xs) \ n & = \text{get\_nth} \ xs \ (n-1) \\
\end{align*}
\]

\[
\begin{align*}
\text{get\_nth} \ (1:(\text{magic} \ (1+1) \ (1+(1+1)))) \ (2) \\
\text{Clearly, we are in the third case; thus, we evaluate:} \\
& \text{get\_nth} \ ((\text{magic} \ (1+1) \ (1+(1+1)))) \ (2-1)
\end{align*}
\]
Example, cont.

```
magic 0 _ = []
magic m n = m : (magic n (m + n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

Clearly, we are in the third case; thus, we evaluate:
```
get_nth ((magic (1+1) (1+(1+1)))) (2-1)
```

Continuing, we again need to know which pattern matches; hence forces one more step in evaluation of `magic`
Example, cont.

```haskell
magic 0 _ = []

magic m n = m : (magic n (m+n))

get_nth [] _ = 0

get_nth (x:xs) 1 = x

get_nth (x:xs) n = get_nth xs (n-1)
```

Now, to figure out which pattern matches in `magic`, we need to evaluate first argument; this yields: `magic 2 (1+(1+1))`
Example, cont.

\[
\text{magic } 0 \ _ = [\]
\text{magic } m \ n = m : (\text{magic } n \ (m+n))
\text{get_nth } [] \ _ = 0
\text{get_nth } (x:xs) \ 1 = x
\text{get_nth } (x:xs) \ n = \text{get_nth } xs \ (n-1)
\]

\[
\text{get_nth } ((\text{magic } (1+1) \ (1+(1+1)))) \ (2-1)
\]

- Now, to figure out which pattern matches in \text{magic}, we need to evaluate first argument; this yields: \textbf{magic } 2 \ (1+(1+1))

- Now, second case matches, thus we have:

\[
2 : (\text{magic } 1+(1+1) \ (2+(1+(1+1))))
\]
Example, cont.

\[
\text{magic} \ 0 \ _ = []
\]

\[
\text{magic} \ m \ n = m : (\text{magic} \ n \ (m+n))
\]

\[
\text{get\_nth} \ [\] \ _ = 0
\]

\[
\text{get\_nth} \ (x:xs) \ 1 = x
\]

\[
\text{get\_nth} \ (x:xs) \ n = \text{get\_nth} \ xs \ (n-1)
\]

\[
\text{get\_nth} \ ((\text{magic} \ (1+1) \ (1+(1+1)))) \ (2-1)
\]

- Now, to figure out which pattern matches in \text{magic}, we need to evaluate first argument; this yields: \text{magic} \ 2 \ (1+(1+1))

- Now, second case matches, thus we have:

\[
2 : (\text{magic} \ 1+(1+1) \ (2+(1+(1+1))))
\]

- Now, we continue evaluating:

\[
\text{get\_nth} \ (2 : (\text{magic} \ 1+(1+1) \ (2+(1+(1+1)))) ) \ (2-1)
\]
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

get_nth ((magic (1+1) (1+(1+1)))) (2-1)

- Now, to figure out which pattern matches in `magic`, we need to evaluate first argument; this yields: `magic 2 (1+(1+1))`

- Now, second case matches, thus we have:
  ```haskell
  2 : (magic 1+(1+1) (2+(1+(1+1))))
  ```

- Now, we continue evaluating:
  ```haskell
  get_nth (2 : (magic 1+(1+1) (2+(1+(1+1)))))) (2-1)
  ```

- This forces evaluation of `2-1`
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

▶ This means we match on second case!
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

- This means we match on second case!
- Thus, the whole expression evaluates to 2!
Example, cont.

magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)

\[
\text{get_nth } (2 : (\text{magic } 1 + (1+1) \ (2 + (1 + (1+1)))))) \ 1
\]

▶ This means we match on second case!

▶ Thus, the whole expression evaluates to 2!

▶ Although we wrote a function to generate infinite list, expression to extract element from this infinite list terminates!
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

- This means we match on second case!
- Thus, the whole expression evaluates to 2!
- Although we wrote a function to generate infinite list, expression to extract element from this infinite list terminates!
- This is one of the nice aspects of lazy evaluation
Haskell Summary

- Haskell is a lazy, pure-functional language.
Haskell Summary

- Haskell is a lazy, pure-functional language
- Integrates a lot of research from PL community: polymorphism, type classes, type inference, ...
Haskell Summary

- Haskell is a lazy, pure-functional language
- Integrates a lot of research from PL community: polymorphism, type classes, type inference, ...
- Statically typed, no escape hatches (e.g., casts) from type system
Haskell Summary

- Haskell is a lazy, pure-functional language
- Integrates a lot of research from PL community: polymorphism, type classes, type inference, ...
- Statically typed, no escape hatches (e.g., casts) from type system
- Considered by many to be a very elegant language