Haskell

We now turn to look at another functional language, Haskell

Haskell

We now turn to look at another functional language, Haskell

Haskell has many things in common with Lisp, so we don't need to spend so much time on it

Haskell

We now turn to look at another functional language, Haskell

Haskell has many things in common with Lisp, so we don't need to spend so much time on it

You should get used to moving *concepts* between languages and not get hung up on things like syntax

Haskell

Haskell is a language that was designed from scratch to be functional

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional
- has first class functions

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional
- has first class functions
- has first class classes

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional
- has first class functions
- has first class classes
- does not have variable update

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional
- has first class functions
- has first class classes
- does not have variable update
- has lambdas (anonymous functions)

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional
- has first class functions
- has first class classes
- does not have variable update
- has lambdas (anonymous functions)
- has closures

Haskell

Haskell is a language that was designed from scratch to be functional

It

- is strongly functional
- has first class functions
- has first class classes
- does not have variable update
- has lambdas (anonymous functions)
- has closures
- has a garbage collector

Haskell

But in a couple of ways it is very different from Lisp

It differs from Lisp in

1. its syntax

Haskell

But in a couple of ways it is very different from Lisp

It differs from Lisp in

1. its syntax
2. its type system

Haskell

But in a couple of ways it is very different from Lisp

It differs from Lisp in

1. its syntax
2. its type system
3. and it is *lazy*

Haskell

There are a couple of implementations of the Haskell standard. The important ones being:

- GHC: Glasgow Haskell Compiler. This compiles to C, which can then be compiled to native code
- Hugs: Haskell User's Gofer System. Compiles to an interpreted bytecode, so is very portable

On BUCS machines `lcpu`

```
% ~masrjb/bin/hugs
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> ^D
[Leaving Hugs]
%
```

Haskell

Syntax

This is *Haskell User's Gofer System*, a Haskell interpreter

This is *Haskell User's Gofer System*, a Haskell interpreter

If we add the argument `+t`

```
~masrjb/bin/hugs +t
```

this makes Hugs give us interesting *type* information

```
> 1+2
3 :: Integer
> :1uit
Command not recognised.  Type :? for help
> :quit
[Leaving Hugs]
```

Hugs requires definitions to be in *modules*. In a file `Egs.hs` put

```
module Egs where
{- this is a comment -}
inc x = x+1
```

Hugs requires definitions to be in *modules*. In a file `Egs.hs` put

```
module Egs where
{- this is a comment -}
inc x = x+1
```

and load into Hugs by

```
> :load Egs.hs
```

Hugs requires definitions to be in *modules*. In a file `Egs.hs` put

```
module Egs where
{- this is a comment -}
inc x = x+1
```

and load into Hugs by

```
> :load Egs.hs
```

The 'E' in `module Egs where` must be upper case

Haskell

Syntax

You can reload a module after changing it by

```
> :reload Egs.hs
```

or simply

```
> :reload
```

will reload the last loaded module again

You can reload a module after changing it by

```
> :reload Egs.hs
```

or simply

```
> :reload
```

will reload the last loaded module again

Definitions must be in modules, but we must type expressions to be evaluated at the prompt. In the examples below we shall mix definitions and evaluations, but you must separate them when actually using Hugs

Functions are defined by equations

```
inc x = x+1          -- definition in a module
> inc 3              -- typed in at prompt
= 4 :: Integer       -- result
```

Functions are defined by equations

```
inc x = x+1          -- definition in a module
> inc 3              -- typed in at prompt
= 4 :: Integer       -- result
```

which is short for

```
inc = \x -> x+1
```

with $\backslash$ for λ

Using +t we see the types of objects

```
> 5
5 :: Integer
> "hello"
"hello" :: String
```

Using `+t` we see the types of objects

```
> 5
5 :: Integer
> "hello"
"hello" :: String
```

We can also directly query expressions for their type using `:t`

Haskell

Syntax

```
> :t 2
2 :: Num a => a
```

Haskell

Syntax

```
> :t 2
2 :: Num a => a
```

We read this in two parts: before and after the =>

Haskell

Syntax

```
> :t 2
2 :: Num a => a
```

We read this in two parts: before and after the =>

The “a” after the => stands for an arbitrary type

```
> :t 2
2 :: Num a => a
```

We read this in two parts: before and after the =>

The “a” after the => stands for an arbitrary type

It is a *type variable*; normally read as “alpha”, written as “α” in text

```
> :t 2
2 :: Num a => a
```

We read this in two parts: before and after the =>

The "a" after the => stands for an arbitrary type

It is a *type variable*; normally read as "alpha", written as "α" in text

Before the => we have the *restriction* `Num a`, meaning `a` is a numerical type

```
> :t 2
2 :: Num a => a
```

We read this in two parts: before and after the =>

The "a" after the => stands for an arbitrary type

It is a *type variable*; normally read as "alpha", written as "α" in text

Before the => we have the *restriction* `Num a`, meaning `a` is a numerical type

Haskell has *classes of types*, which are types of types, i.e., second order types

So why is `2` some unspecified numerical type, rather than the more obvious `Integer`?

So why is `2` some unspecified numerical type, rather than the more obvious `Integer`?

Because the type of this particular expression depends on the context in which it is used

So why is `2` some unspecified numerical type, rather than the more obvious `Integer`?

Because the type of this particular expression depends on the context in which it is used

In `2 + 1.0` the value 2 is automatically assumed to be a `Double`

So why is `2` some unspecified numerical type, rather than the more obvious `Integer`?

Because the type of this particular expression depends on the context in which it is used

In `2 + 1.0` the value 2 is automatically assumed to be a `Double`

So `Num a => a` is saying the expression “2” can be used anywhere we expect a number, of any type

So why is `2` some unspecified numerical type, rather than the more obvious `Integer`?

Because the type of this particular expression depends on the context in which it is used

In `2 + 1.0` the value 2 is automatically assumed to be a `Double`

So `Num a => a` is saying the expression “2” can be used anywhere we expect a number, of any type

Exercise. What do you expect from `:t 2.0`?

Haskell

Syntax

```
inc x = x+1
> :t inc
= inc :: Num a => a -> a
```

```
inc x = x+1
> :t inc
= inc :: Num a => a -> a
```

The `a -> a` means a function that takes an argument of type `a` and returns a value of type `a`

```
inc x = x+1
> :t inc
= inc :: Num a => a -> a
```

The `a -> a` means a function that takes an argument of type `a` and returns a value of type `a`

And `a` must be a `Num` type

`Num a => a -> a` can be read as $\forall\alpha \in \texttt{Num}.\alpha \rightarrow \alpha$, which is to say "for all numerical types . . . "

`Num a => a -> a` can be read as $\forall \alpha \in \texttt{Num}.\alpha \rightarrow \alpha$, which is to say "for all numerical types . . . "

$\texttt{inc 2} \rightarrow \texttt{3 :: Integer}$

Haskell

Syntax

`Num a => a -> a` can be read as $\forall\alpha \in \texttt{Num}.\alpha \rightarrow \alpha$, which is to say “for all numerical types ...”

`inc 2` $\rightarrow$ `3 :: Integer`

`inc 2.0` $\rightarrow$ `3.0 :: Double`

`Num a => a -> a` can be read as $\forall\alpha \in \texttt{Num}.\alpha \rightarrow \alpha$, which is to say "for all numerical types ..."

$\texttt{inc 2} \rightarrow \texttt{3 :: Integer}$

$\texttt{inc 2.0} \rightarrow \texttt{3.0 :: Double}$

The function `inc` takes an argument of any numerical type and returns a value of the same type

`Num a => a -> a` can be read as $\forall\alpha \in \texttt{Num}.\alpha \to \alpha$, which is to say "for all numerical types ..."

`inc 2` $\to$ `3 :: Integer`

`inc 2.0` $\to$ `3.0 :: Double`

The function `inc` takes an argument of any numerical type and returns a value of the same type

So: `Num a => a -> a`

`Num a => a -> a` can be read as $\forall \alpha \in \texttt{Num}.\alpha \rightarrow \alpha$, which is to say "for all numerical types …"

`inc 2` $\rightarrow$ `3 :: Integer`

`inc 2.0` $\rightarrow$ `3.0 :: Double`

The function `inc` takes an argument of any numerical type and returns a value of the same type

So: `Num a => a -> a`

`inc` is sometimes called a *polymorphic* function: the same function works on many types

Haskell

Type Inference

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

It can figure this out by *type inference*

Haskell

Type Inference

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

It can figure this out by *type inference*

In this case (`inc x = x+1`) it sees that `x` is an argument to the function +

Haskell

Type Inference

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

It can figure this out by *type inference*

In this case (`inc x = x+1`) it sees that `x` is an argument to the function +

But + takes numerical arguments

Haskell

Type Inference

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

It can figure this out by *type inference*

In this case (`inc x = x+1`) it sees that `x` is an argument to the function +

But + takes numerical arguments

Thus `x` must be numerical

Haskell

Type Inference

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

It can figure this out by *type inference*

In this case (`inc x = x+1`) it sees that `x` is an argument to the function +

But + takes numerical arguments

Thus `x` must be numerical

And the result of the `inc` is the result of the +

Haskell

Type Inference

Haskell manages to figure out the type of this function itself: the programmer didn't need to put type information in themselves

It can figure this out by *type inference*

In this case (`inc x = x+1`) it sees that `x` is an argument to the function +

But + takes numerical arguments

Thus `x` must be numerical

And the result of the `inc` is the result of the +

And the result of + is the same as the type of its argument, namely numerical

Some things do not make sense:

```
> 2+"a"
ERROR - Cannot infer instance
*** Instance   : Num [Char]
*** Expression : 2 + "a"
```

Some things do not make sense:

```
> 2+"a"
ERROR - Cannot infer instance
*** Instance   : Num [Char]
*** Expression : 2 + "a"
```

Type inference is a important subject in CS, particularly in compilers

Some things do not make sense:

```
> 2+"a"
ERROR - Cannot infer instance
*** Instance   : Num [Char]
*** Expression : 2 + "a"
```

Type inference is a important subject in CS, particularly in compilers

Exercise. In Haskell, putting `()` around an infix operator makes it into a normal function (not infix). Try `:t (+)`

The class `Num` contains the types `Integer`, `Float` and `Double` amongst others

The class `Num` contains the types `Integer`, `Float` and `Double` amongst others

These numerical types are also members of the class `Ord` of objects that support comparison, i.e., $<$

The class `Num` contains the types `Integer`, `Float` and `Double` amongst others

These numerical types are also members of the class `Ord` of objects that support comparison, i.e., $<$

For example, strings can be ordered, but they are not numbers

Haskell

Syntax

The class `Num` contains the types `Integer`, `Float` and `Double` amongst others

These numerical types are also members of the class `Ord` of objects that support comparison, i.e., $<$

For example, strings can be ordered, but they are not numbers

And complexes are numbers that don't have a natural order

The class `Num` contains the types `Integer`, `Float` and `Double` amongst others

These numerical types are also members of the class `Ord` of objects that support comparison, i.e., $<$

For example, strings can be ordered, but they are not numbers

And complexes are numbers that don't have a natural order

Use, e.g., `:info Ord` to see details of a class or any other object

Exercise. Strings in Haskell are actually arrays of characters: find out what Haskell does for `Ord` and arrays

Exercise. Think about writing a polymorphic `sort` function that works on any list whose elements admit an ordering, i.e., a function of type `Ord => [a] -> [a]`

For the function definition

```
positive x = if x > 0 then True else False
```

or, less clumsily,

```
positive x = x > 0
```

We get

```
> :t positive
= positive :: (Ord a, Num a) => a -> Bool
```

Haskell

Syntax

```
(Ord a, Num a) => a -> Bool
```

This is type $\forall \alpha \in \texttt{Ord} \cap \texttt{Num}. \alpha \rightarrow \texttt{Bool}$

```
(Ord a, Num a) => a -> Bool
```

This is type $\forall \alpha \in \mathtt{Ord} \cap \mathtt{Num}. \alpha \rightarrow \mathtt{Bool}$

A function that takes an argument of type `a` and returns a Boolean (true/false) value

```
(Ord a, Num a) => a -> Bool
```

This is type $\forall\alpha \in \texttt{Ord} \cap \texttt{Num}.\alpha \to \texttt{Bool}$

A function that takes an argument of type `a` and returns a Boolean (true/false) value

And the type `a` must be both of class `Num` (numerical) *and* class `Ord` (have comparison)

```
(Ord a, Num a) => a -> Bool
```

This is type $\forall\alpha \in \texttt{Ord} \cap \texttt{Num}.\alpha \rightarrow \texttt{Bool}$

A function that takes an argument of type `a` and returns a Boolean (true/false) value

And the type `a` must be both of class `Num` (numerical) *and* class `Ord` (have comparison)

So this works for any numeric type that also has comparison (recall that complex numbers don't have comparison)

Haskell

Syntax

```
(Ord a, Num a) => a -> Bool
```

This is type $\forall\alpha \in \texttt{Ord} \cap \texttt{Num}.\alpha \to \texttt{Bool}$

A function that takes an argument of type `a` and returns a Boolean (true/false) value

And the type `a` must be both of class `Num` (numerical) *and* class `Ord` (have comparison)

So this works for any numeric type that also has comparison (recall that complex numbers don't have comparison)

Exercise. Work through the type inference of this for yourself

Haskell can define functions by case:

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs
```

Haskell

Syntax

Haskell can define functions by case:

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs
```

We start by declaring the type of the function `len`

Haskell can define functions by case:

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs
```

We start by declaring the type of the function `len`

Why? Explanation in a moment

Haskell

Syntax

Haskell can define functions by case:

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs
```

We start by declaring the type of the function `len`

Why? Explanation in a moment

Then there are *two* equations that define the behaviour of `len` on

(a) the empty list `[]`
(b) a non-empty list that has `car` `x` and `cdr` `xs`

This is very similar to the standard way of writing recursive functions, but the cases are split out and the interpreter can pick the right case by *pattern matching*

This is very similar to the standard way of writing recursive functions, but the cases are split out and the interpreter can pick the right case by *pattern matching*

The pattern “`len []`” only matches `len` called on the empty list

This is very similar to the standard way of writing recursive functions, but the cases are split out and the interpreter can pick the right case by *pattern matching*

The pattern "`len []`" only matches `len` called on the empty list

The pattern "`len (x:xs)`" matches the case when the argument is a cons. The Haskell version of `cons` is an infix `:`

This is very similar to the standard way of writing recursive functions, but the cases are split out and the interpreter can pick the right case by *pattern matching*

The pattern "`len []`" only matches `len` called on the empty list

The pattern "`len (x:xs)`" matches the case when the argument is a cons. The Haskell version of `cons` is an infix `:`

So `x:xs` is a list with car (`head`) `x` and cdr (`tail`) `xs`

This kind of definition by cases specified by patterns is common in non-Lispy functional languages

Haskell

Syntax

This kind of definition by cases specified by patterns is common in non-Lispy functional languages

It can be added to Lisp, but most Lispers don't care for this style

Haskell

Syntax

This kind of definition by cases specified by patterns is common in non-Lispy functional languages

It can be added to Lisp, but most Lispers don't care for this style

```
:t len
len :: [a] -> Integer
```

Haskell

Syntax

This kind of definition by cases specified by patterns is common in non-Lispy functional languages

It can be added to Lisp, but most Lispers don't care for this style

```
:t len
len :: [a] -> Integer
```

A function that takes an argument of type *list* of `a` and returns an integer

This kind of definition by cases specified by patterns is common in non-Lispy functional languages

It can be added to Lisp, but most Lispers don't care for this style

```
:t len
len :: [a] -> Integer
```

A function that takes an argument of type *list* of `a` and returns an integer

Elements in a list in Haskell are all the same type, unlike Lisp where we can mix as we please

Haskell

Syntax

This kind of definition by cases specified by patterns is common in non-Lispy functional languages

It can be added to Lisp, but most Lispers don't care for this style

```
:t len
len :: [a] -> Integer
```

A function that takes an argument of type *list* of `a` and returns an integer

Elements in a list in Haskell are all the same type, unlike Lisp where we can mix as we please

Haskell has to do this to get type inference to work

Why did we start with the declaration of the type of `len`?

Why did we start with the declaration of the type of `len`?

Mostly to show we can do it if we wish

Why did we start with the declaration of the type of `len`?

Mostly to show we can do it if we wish

Without the declaration Haskell infers the type of `len` to be
`Num a => [b] -> a`

(Haskell uses type variables α, then β, etc.)

Why did we start with the declaration of the type of `len`?

Mostly to show we can do it if we wish

Without the declaration Haskell infers the type of `len` to be
`Num a => [b] -> a`

(Haskell uses type variables α, then β, etc.)

This is less precise: it can only conclude that the length will be some numerical type

Haskell

Syntax

Why did we start with the declaration of the type of `len`?

Mostly to show we can do it if we wish

Without the declaration Haskell infers the type of `len` to be
`Num a => [b] -> a`

(Haskell uses type variables α, then β, etc.)

This is less precise: it can only conclude that the length will be some numerical type

We know that the length will be actually an integer, so we can help Haskell by declaring the type ourselves

So we can give help if we want, or want to restrict how a function is used

So we can give help if we want, or want to restrict how a function is used

It also allows Haskell to typecheck our code, making sure the types of the functions we define match the types we have declared

So we can give help if we want, or want to restrict how a function is used

It also allows Haskell to typecheck our code, making sure the types of the functions we define match the types we have declared

If we declare

```
len :: [a] -> Integer
len [] = 0.0
```

then Haskell produces an error

Exercise. What are the types of

- `head [1,2]`
- `head [1.0, 2.0]`
- `tail [1, 2]`
- `tail [1.0, 2.0]`
- `tail [1]`
- `tail [1.0]`
- `[]`

Types are central to Haskell

Haskell

Syntax

Types are central to Haskell

Through type inference Haskell can figure out precisely what a function is supposed to be doing

Haskell

Syntax

Types are central to Haskell

Through type inference Haskell can figure out precisely what a function is supposed to be doing

It can sometimes produce compiled code equal or better than C

Haskell

Syntax

Types are central to Haskell

Through type inference Haskell can figure out precisely what a function is supposed to be doing

It can sometimes produce compiled code equal or better than C

More information leads to better compilation

Types are central to Haskell

Through type inference Haskell can figure out precisely what a function is supposed to be doing

It can sometimes produce compiled code equal or better than C

More information leads to better compilation

By restricting what code we can write, we supposedly write better code (this was the idea behind Java, too)

Haskell

Syntax

Types are central to Haskell

Through type inference Haskell can figure out precisely what a function is supposed to be doing

It can sometimes produce compiled code equal or better than C

More information leads to better compilation

By restricting what code we can write, we supposedly write better code (this was the idea behind Java, too)

Lisp is freewheeling on types: they are there but they don't try to stop you doing what you want

Haskell

Syntax

The functions in Haskell are much like those of Lisp: you just need to discover their names

Haskell

Syntax

The functions in Haskell are much like those of Lisp: you just need to discover their names

For example, `map`

```
map (\n -> n*n) [1, 2, 3]
```
$\rightarrow$
```
[1, 4, 9] :: [Integer]
```

Haskell

Syntax

Once given a value, a symbol cannot be reassigned (within a module)

```
x = 1
x = 2
ERROR haskell.hs:17 - "x" multiply defined
```

Once given a value, a symbol cannot be reassigned (within a module)

```
x = 1
x = 2
ERROR haskell.hs:17 - "x" multiply defined
```

though it can be locally rebound

```
> let { x = 1; y = 2 } in 2*x+y
= 4 :: Integer
```

Once given a value, a symbol cannot be reassigned (within a module)

```
x = 1
x = 2
ERROR haskell.hs:17 - "x" multiply defined
```

though it can be locally rebound

```
> let { x = 1; y = 2 } in 2*x+y
= 4 :: Integer
```

The only way to change an assignment is to edit the module and reload it. This is so Haskell can have referential transparency

There's no way to update a variable once it has a value

There's no way to update a variable once it has a value

Haskell is described as a *single assignment* language

The other important difference between Lisp and Haskell is that Haskell is lazy:

```
from n = n : from(n+1)
> :t from
= from :: Num a => a -> [a]
```

This defines `from` as a function returning an *infinite* list of numbers starting from *n*

Haskell

Lazy

```
ints = from 0
> :t ints
= ints :: [Integer]
```

Haskell

Lazy

```
ints = from 0
> :t ints
= ints :: [Integer]

> head ints
= 0 :: Integer
```

Haskell

Lazy

```
ints = from 0
> :t ints
= ints :: [Integer]

> head ints
= 0 :: Integer
> head(tail ints)
= 1 :: Integer
```

Haskell

Lazy

```
ints = from 0
> :t ints
= ints :: [Integer]
```

```
> head ints
= 0 :: Integer
> head(tail ints)
= 1 :: Integer
```

Don't type in "`ints`" unless you have a lot of spare time!

We can do as many `tails` as we wish and get the appropriate integer as the `head`

Haskell

Lazy

We can do as many `tails` as we wish and get the appropriate integer as the `head`

`ints` is acting like an *infinite* list of integers

We can do as many `tails` as we wish and get the appropriate integer as the `head`

`ints` is acting like an *infinite* list of integers

Try this is Lisp (and most other languages) and you never get past the call to `(from 0)`

Haskell

Lazy

We can do as many `tails` as we wish and get the appropriate integer as the `head`

`ints` is acting like an *infinite* list of integers

Try this is Lisp (and most other languages) and you never get past the call to `(from 0)`

Most languages are *eager* and try to evaluate everything as soon as they can

We can do as many `tails` as we wish and get the appropriate integer as the `head`

`ints` is acting like an *infinite* list of integers

Try this is Lisp (and most other languages) and you never get past the call to `(from 0)`

Most languages are *eager* and try to evaluate everything as soon as they can

If we try to evaluate `(from 0)` they eagerly go into a infinite loop of evaluating `(from 1)` then `(from 2)` and so on

We can do as many `tails` as we wish and get the appropriate integer as the `head`

`ints` is acting like an *infinite* list of integers

Try this is Lisp (and most other languages) and you never get past the call to `(from 0)`

Most languages are *eager* and try to evaluate everything as soon as they can

If we try to evaluate `(from 0)` they eagerly go into a infinite loop of evaluating `(from 1)` then `(from 2)` and so on

In practice they soon run out of memory