Programming Languages CSCE-314

Jaakko Järvi

TAMU

August 30, 2012

Outline

Outline

Haskell compilers and interpreters

- http://www.haskell.org/implementations.html
- Two main ones:
  - Hugs interpreter
  - GHC Compiler and interpreter
- Installation packages exist for all common platforms
- GHC installed on `[[linux.cs.tamu.edu]]`

Using GHC and GHCi

- From a shell window, the compiler is invoked as `ghc myfile.hs`, interpreter as `ghci` (or as `ghc` $--interactive$).
- We may need to use some options later to enable some non-standard language features
- For multi-file programs, $--make$ is useful option for the compiler
- Important: Make your edit-compile-run cycle convenient!
  - I use Emacs and "haskell-mode" http://projects.haskell.org/haskellmode-emacs/
  - Code highlighting, smart indenting, inspecting expressions' types, ...
  - Keybindings for compile/interpret, jump to an error, ...

Using GHCi

- Invoking the shell command `ghci` in UNIX (or Linux, Mac OS X, Cygwin, . . .), or clicking GHCI in Windows should result in a terminal window roughly as follows:

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.4.2, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\hspace{5ex}/\/ /_/\hspace{5ex}/|_|      Type :? for help.

Loading package base-1.0 ... linking ... done.
Prelude> _
```

- The interpreter is ready to accept commands. It operates on a so called *eval-print-loop*: user types in a Haskell expression, the interpreter evaluates it, prints out the result, and waits for the next expression.
- Instead of expressions, there are commands, not part of the Haskell language, that can be given to operate the interpreter, for example to open and close files, quit the interpreter, and so forth.

GHCi commands

- All commands start with the colon, `:`. Some useful commands:

`:?`	Help!
`:load test`	Open file `test.hs` or `test.lhs`
`:reload`	Reload the previously loaded file
`:main a1 a2`	Invoke `main` function with command line args `a1 a2`
`:!`	Execute a shell command
`:quit`	

- Commands can be abbreviated. E.g., `:r` is `:reload`
- Hint: GHCi executes commands from `$HOME/.ghci`, then from `./.ghci` at startup.

Using the interpreter

- At startup, the definitions of the "Standard Prelude" are loaded
- Interpreter waits for an expression, evaluates one when it gets it, and shows the value on the screen
- Example:

```
Prelude> 4 + 5
9
Prelude> head [1, 2, 3]
1
Prelude>
```

- Loading new scripts causes new definitions to be in scope
- Example

```
Prelude> :l myscript
Compiling Main         ( myscript.hs, interpreted )
Ok, modules loaded: Main.
*Main> myFunc 1
2
```

Outline

(Very Brief) History of Haskell

- 1997: An international committee of PL researchers initiates the development of a standard lazy functional language, Haskell
- 2003: Haskell 98 report published
- Next round of standardization has begun: Haskell' (Haskell prime)
  - a continuous standardization process
- Status in 2012?
  - A widely used and highly influential language *for programming language research*
  - Reasonably widely used in open-source software
  - Modest commercial use

Haskell is a lazy pure functional language

- Functional?
  - Language that supports a style of programming where the basic method of computation is application of functions to arguments

```
int s = 0;
for (int i=0; i <= 100: ++i)
  s = s + i;
```

```
sum [1..100]
```

- Lazy?
  - Evaluate as late as possible (the following program does not go to an infinite loop, or overflow):

```
omit x = 0
keep_going x = keep_going (x+1)
omit (keep_going 1)
```

- Pure?
  - No side-effects (or at least they are confined)
  - No destructive assignment: `a = 1; a = 2;` −− *illegal*
  - Helps reasoning—referential transparency

Other characteristics of Haskell

- Statically typed with type inference
- Rich type system
- Terse syntax, quite expressive $\rightarrow$ short programs
- Small peculiarities
  - Indentation matters
  - Capitalization of identifiers matters

Example

- What does this function do?

Example

```
f []      = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
              ys = [a | a <- xs, a <= x]
              zs = [b | b <- xs, b > x]
```

Outline

Functions

- Functions are defined as equations:

```
square x = x * x
add x y = x + y
```

- Syntax for function application is just juxtaposition

```
> square 7
49
> add 2 3
5
```

In C, these calls would be

```
square(7);
add(2, 3);
```

- Parentheses are often needed in Haskell too

```
> add (square 2) (add 2 3)
9
```

- Function application has the highest priority

```
square 2 + 3 == (square 2) + 3
```

Functions

- Functions are defined as equations:

```
square x = x * x
add x y = x + y
```

- Syntax for function application is just juxtaposition

```
> square 7
49
> add 2 3
5
```

In C, these calls would be

```
square(7);
add(2)(3);
```

- Parentheses are often needed in Haskell too

```
> add (square 2) (add 2 3)
9
```

- Function application has the highest priority

```
square 2 + 3 == (square 2) + 3
```

Evaluating functions

- Think of evaluating functions as substitution and reduction

```
add x y = x + y; square x = x * x

add (square 2) (add 2 3)
 -- apply square
add (2 * 2) (add 2 3)
 -- apply *
add 4 (add 2 3)
-- apply inner add
add 4 (2 + 3)
-- apply +
add 4 5
 -- apply add
4 + 5
 -- apply outer add
9
```

Evaluating functions

- There are many possible orders to evaluate a function

```
head (1:(reverse [2, 3, 4, 5]))
-- apply reverse
-- ...
-- ... many steps omitted here
-- ...
head (1 : [5, 4, 3, 2])
-- apply head
1
```

```
head (1:(reverse [2, 3, 4, 5]))
-- apply head
1
```

- In a *pure* language like Haskell, evaluation order does not affect the *value* of the computation
- It can, however, affect the *amount* of computation and whether the computation *terminates* or not (or fails with a run-time error)
- Haskell evaluates a function's argument lazily
  - "Call-by-need" $\sim$ only apply a function if it's value is needed, and "memoize" what's already been evaluated

Operators vs. functions

- Haskell allows combinations of most symbols to be defined as functions

```
x %$+@#$#$&><#$@ y = "bad language"
```

Operators vs. functions

- Haskell allows combinations of most symbols to be defined as functions

```
x %$+@#$#$&><#$@ y = "bad language"
```

- A more realistic example

```
x +/- y = (x - y, x + y)
```

```
> 100 +/- 5
(95, 105)
```

Operators vs. functions

- Haskell allows combinations of most symbols to be defined as functions

```
x %$+@#$#$&><#$@ y = "bad language"
```

- A more realistic example

```
x +/- y = (x - y, x + y)
```

```
> 100 +/- 5
(95, 105)
```

- Any function with two or more arguments can be used as an infix operator (enclosed in back quotes) and any infix operator as a function (enclosed in parentheses)

```
100 +/- 5          (+/-) 100 5
add 2 3            2 `add` 3
```

More examples

- Functions can consist of more than one equation, and equations can be recursive

```
not True = False
not False = True

factorial 0 = 1
factorial n = n * factorial (n-1)

length [] = 0
length (x:xs) = 1 + length xs
```

- Mutual recursion is fine too

```
even 0 = True              odd 0 = False
even n = odd (n-1)         odd n = even (n-1)
```

- Function call is matched to the left-hand sides (*patterns*) of the equations in order (top to bottom), first one to match is used → patterns do not have to be mutually exclusive

Lambda functions

- Functions are just values, and can be unnamed

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

- An unnamed function can be called right after it has been defined (not very useful)

```
(\x -> x + 1) 5
```

- More usefully it can be bound to a name (variable, function argument)

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

...
six = incrementer 5
```

Lambda functions

- Think of lambdas as the more primitive notation for defining functions. The equations we have seen are just syntactic sugar

As equations

```
square x = x * x
add x y  = x + y
```

Using lambdas

```
square = \x -> x * x
add    = \x y -> x + y
```

Example

```
g = (\f -> (\x -> f (f x)))
```

Example

```
g = (\f -> (\x -> f (f x)))
```

- `g` expects a function `f` and returns another function that applies `f` twice to its argument
- What values do these calls evaluate to?

```
g (\x -> x) True
g not True
g (\x -> x + 1) 3
```

Outline

Haskell Scripts

- Haskell programs consist of collections of scripts
- Scripts consist of definitions, such as those of functions
- Two conventions for file name suffixes: `filename.hs` — normal Haskell script `filename.lhs` — literate Haskell script
- Additional naming convention:
  - If a file defines a module, say `M`, use the name of the module as the file name (`M.hs`)
    (Modules are explained later)
- Typical mode of operation:
  - GHCi running in one window, some `script.hs` open in an editor window
  - Edit script, load it, edit more, load again (reload), ...

Normal scripts vs. literal scripts

myscript.hs

```
-- myscript.hs : a few simple functions

-- add computes the sum of two numbers
add x y = x + y

-- square multiplies its argument by itself
square x = x * x
```

myscript.lhs

```
myscript.hs : a few simple functions

add computes the sum of two numbers

> add x y = x + y

square multiplies its argument by itself

> square x = x * x
```

- Normal: comments marked with --, multiline comments with *{- a multiline comment -}*
- Literate: code marked with > (and empty lines before and after)

Layout

- Haskell uses layout of a script to determine the structure of definitions
- Can avoid braces and semicolons in most cases
- Braces and semicolons still allowed, and can be freely mixed with layout sensitive coding

```haskell
myfunc x =
    let a = 5
        b = 6
    in x + a + b
```

```haskell
myfunc x =
    let { a = 5 ; b = 6 }
    in x + a + b
```

- Definitions in a sequence start from the exact same column
  - Less indented — "close brace inserted"
  - More indented — "still on the same definition"
  - Same indentation — "semicolon inserted"
- The "Haskell way" seems to be to use layout rather than punctuation

Outline

Types

- A type is a collection of values (somehow related)
- Examples
  - **Bool** has values **True** and **False**
  - **Int** has values $-2^{29}, \ldots -1, 0, 1, \ldots, 2^{29} - 1$
- Types enable statically (at compilation time, or in general, prior to a program is run) detecting many programming errors

```
> 1 + False

<interactive>:1:2:
    No instance for (Num Bool)
      arising from use of `+' at <interactive>:1:2
    Probable fix: add an instance declaration for (Num Bool)
    In the definition of `it': it = 1 + False
```

  - `it` is the result of the most recently evaluated expression in GHCi
- Benefits of static type checking:
  - certain classes of defects eradicated
  - faster programs

Type annotations

- Haskell can (usually) infer types of expressions (type inference)
- Programmer can (and at times must) annotate expressions with types
- That an expression `e` has type `T` is written

```
e :: T
```

- Examples:

```
True :: Bool
5 :: Int              -- type is really (Num t) => t
(5 + 5) :: Int        -- likewise
(7 < 8) :: Bool
```

- Some expressions, e.g., `5` can have many types:
  `5::Int`, `5::Integer`, `5::Float`
- Aside: GHCi command `:t e` shows the type of `e`

Basic types

- **Bool**
- **Char**
  - Single characters, e.g., `'a'` `'A'` `'+'` `'n'` ...
- **String**
  - Lists of characters, e.g., `"string"` `"one two three"` `""`
  - Not really a primitive type: `type String = [Char]`
- **Int**
- **Integer**
  - Arbitrary-precision integers (usually slower than **Int**s)
- **Float**, **Double**
  - Single/double-precision floating point number types
- `Unit`
  - Singleton type, has only one value: `()`
- At least **Bool**, **String**, **Integer**, `Unit` are defined (definable) in a library with the help of a little syntactic sugar

Composite types : List types

- Composite types are built from other types using type constructors
- List types represent lists of values, all of the same type

```
['a', 'b', 'c'] :: [Char]
"abc" :: [Char]
[[True, True], []] :: [[Bool]]
```

  - `[a]` is the type of a list with elements of type `a`
- Lists do not have to be finite: `l = [1..]`

Tuple types

- A tuple type is composed of a finite sequence of other types
- Example: If `t1`, `t2`, `t3` are types, `(t1, t2, t3)` is a tuple type, whose *components* are `t1`, `t2`, `t3`.
- Number of elements in a tuple is its arity
- More examples:

```
('a', True) :: (Char, Bool)
("Hello", True, "World") :: ([Char], Bool, [Char])
```

- How many (and which) values do the types below have?

```
(Bool, Bool)
((), (), ())
```

- Note: tuples with arity one are not supported
  - `(f)` is parsed as `f`, parentheses are ignored

Function types

- Function is a mapping from values of one type (`T1`) to values of another type (`T2`)
- Written as `T1 -> T2`
- Examples:

```
not :: Bool -> Bool
isAlpha :: Char -> Bool
toUpper :: Char -> Char
(&&) :: Bool -> Bool -> Bool
```

- It is usually a good idea to annotate functions with types
  - important part of documentation
- Annotations can give a function a less general type than what the compiler infers

```
square :: Int -> Int
square x = x * x
```

```
add :: Int -> Int -> Int
add x y = x + y
```

Curried functions

- Haskell's functions types are always of the form `T -> U`
- Only single parameter functions!
- Multiple parameters or results with lists:

```
zeroTo :: Int -> [Int]
zeroTo n = [1..n]
```

- Or with tuples:

```
plusMinus :: (Int, Int) -> (Int, Int)
plusMinus (x, y) = (x - y, x + y)
```

- But currying is more common

Curried functions

- Example:

```
add :: Int -> (Int -> Int)
add x y  = x + y

add3 :: Int -> Int
add3 = add 3

z :: Int
z = add3 4
```

- Functions returning functions enable multiple arguments
  - Aside: our first example of higher-order functions
- Type constructor -> associates to the right; the following types are the same:

```
a -> b -> c -> d -> e
a -> (b -> (c -> (d -> e)))
```

- Function application associates to the left; the following are the same

```
add 1 2                     ((add 1) 2)
```

. . . curried functions

- Currying extends to any number of arguments

```
stringify :: Char -> Char -> Char -> Char -> [Char]
stringify a b c d = '(':[a] ++ [b] ++ [c] ++ d:[')']
```

- Currying is useful for partial function application (and for obfuscated code, at least until one gets used to it)
- Partially applied functions may be useful themselves

```
(+) :: Int -> Int -> Int

addN :: Int -> Int
addN n = (+) n

replicate :: n -> a -> [a]

triList :: a -> [a]
triList = replicate 3

> triList 'a'
"aaa"
```

- Later: currying applies to type constructors too

Totality of functions

- Reminder:
  - (Total) function maps every element in the function's domain to an element in its co-domain
  - Partial function maps zero or more elements in the function's domain to an element in its co-domain, and can leave some elements undefined
- Haskell functions can be partial. E.g., what happens with the call **head ""**?

```
head :: [a] -> a
head (x:_) = x
```

Totality of functions

- Reminder:
  - (Total) function maps every element in the function's domain to an element in its co-domain
  - Partial function maps zero or more elements in the function's domain to an element in its co-domain, and can leave some elements undefined
- Haskell functions can be partial. E.g., what happens with the call **head ""**?

```
head :: [a] -> a
head (x:_) = x
```

```
> head ""
*** Exception: Prelude.head: empty list
```

- Another example: `"10elements" !! 10`

Totality of functions

- Reminder:
  - (Total) function maps every element in the function's domain to an element in its co-domain
  - Partial function maps zero or more elements in the function's domain to an element in its co-domain, and can leave some elements undefined
- Haskell functions can be partial. E.g., what happens with the call `head ""`?

```
head :: [a] -> a
head (x:_) = x
```

```
> head ""
*** Exception: Prelude.head: empty list
```

- Another example: `"10elements" !! 10`

```
> "10elements" !! 10
*** Exception: Prelude.(!!): index too large
```

Outline

Parametrically polymorphic functions

- polymorphism — "the occurrence of something in several different forms"
- Polymorphic functions work with many types of arguments
- Question: What are the types of these functions?

```
id x = x
```

```
length [] = 0
length (_:xs) = 1 + length xs
```

Parametrically polymorphic functions

- polymorphism — "the occurrence of something in several different forms"
- Polymorphic functions work with many types of arguments
- Question: What are the types of these functions?

```
id :: a -> a
id x = x

length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Parametrically polymorphic functions

- polymorphism — "the occurrence of something in several different forms"
- Polymorphic functions work with many types of arguments
- Question: What are the types of these functions?

```
id :: a -> a
id x = x

length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

- **id** maps a value of *any type* a to itself
- **length** computes the length of a list whose element type is of *any type* a
- a is a type variable

Parametrically polymorphic functions

- polymorphism — "the occurrence of something in several different forms"
- Polymorphic functions work with many types of arguments
- Question: What are the types of these functions?

```
id :: a -> a
id x = x

length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

```
-- Example uses
id 5
id [1, 2, 3]
id id

length []
length [1, 2, 3]
```

- `id` maps a value of *any type* `a` to itself
- `length` computes the length of a list whose element type is of *any type* `a`
- `a` is a type variable

Polymorphic types and type variables

- A polymorphic type is a type that contains one or more type variables
- Think of it as a *schema* or *template* from which to instantiate other types by binding values to the type variables

expression	polymorphic type	type variable bindings	resulting type
`id`	`a -> a`	`a=Int`	`Int -> Int`
`id`	`a -> a`	`a=Bool`	`Bool -> Bool`
`length`	`[a] -> Int`	`a=Char`	`[Char] -> Int`
`fst`	`(a, b) -> a`	`a=Char, b=Bool`	`Char`
`snd`	`(a, b) -> b`	`a=Char, b=Bool`	`Bool`
`([], [])`	`([a], [b])`	`a=Char, b=Bool`	`([Char], [Bool])`

- Type variables must start with lowercase letters
- Typical conventions: `a`, `b`, `c`, . . . , `t`, `u`, . . . , `a1`, `a2`, . . . , `a'`, `a''`, . . .

More about polymorphic types

- What does the following function do, and what is its type?

```
twice f x = f (f x)
```

More about polymorphic types

- What does the following function do, and what is its type?

```
twice :: (t -> t) -> t -> t
twice f x = f (f x)

> twice tail "abcd"
"cd"
```

More about polymorphic types

- What does the following function do, and what is its type?

```
twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

```
> twice tail "abcd"
"cd"
```

- What is the type of `twice twice`?

More about polymorphic types

- What does the following function do, and what is its type?

```
twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

```
> twice tail "abcd"
"cd"
```

- What is the type of `twice twice`?
  - The parameter and return type of `twice` are the same `(t -> t)`

More about polymorphic types

- What does the following function do, and what is its type?

```
twice :: (t -> t) -> t -> t
twice f x = f (f x)

> twice tail "abcd"
"cd"
```

- What is the type of `twice twice`?
  - The parameter and return type of `twice` are the same `(t -> t)`
  - Therefore, `twice` and `twice twice` have the same type

More about polymorphic types

- What does the following function do, and what is its type?

```
twice :: (t -> t) -> t -> t
twice f x = f (f x)

> twice tail "abcd"
"cd"
```

- What is the type of `twice twice`?
  - The parameter and return type of `twice` are the same `(t -> t)`
  - Therefore, `twice` and `twice twice` have the same type
  - As `twice :: (t -> t) -> (t -> t)`, then also `twice twice :: (t -> t) -> (t -> t)` has to be an instance of type `t -> t`

Class constraints

- Parametrically polymorphic types can be instantiated with **all** types!
- Notice how in `id :: a -> a`, `length :: [a] -> Int` no operation is subjected to values of type `a`
- What are the types of these functions?

```
min x y = if x < y then x else y
```

```
elem x (y:xs) | x == y = True
elem x (y:ys) = elem x ys
elem x [] = False
```

Class constraints

- Parametrically polymorphic types can be instantiated with **all** types!
- Notice how in `id :: a -> a`, `length :: [a] -> Int` no operation is subjected to values of type `a`
- What are the types of these functions?

```
min :: Ord a => a -> a -> a
min x y = if x < y then x else y

elem :: Eq a => a -> [a] -> Bool
elem x (y:xs) | x == y = True
elem x (y:ys) = elem x ys
elem x [] = False
```

- `Ord a` and `Eq a` are *class constraints*.
- Type variables can only be bound to types that satisfy the constraints

About class constraints

- Constraints arise because values of the generic types are subjected to operations that are not defined to all types

```
min :: Ord a => a -> a -> a
min x y = if x < y then x else y
elem :: Eq a => a -> [a] -> Bool
elem x (y:xs) | x == y -> True
elem x (y:ys) = elem x ys
elem x [] = False
```

- **Ord** and **Eq** are type classes
- A function whose type contains one or more class constraints is said to be overloaded

About class constraints

- Constraints arise because values of the generic types are subjected to operations that are not defined to all types

```
min :: Ord a => a -> a -> a
min x y = if x < y then x else y
elem :: Eq a => a -> [a] -> Bool
elem x (y:xs) | x == y -> True
elem x (y:ys) = elem x ys
elem x [] = False
```

- **Ord** and **Eq** are type classes
- A function whose type contains one or more class constraints is said to be overloaded

Example type classes

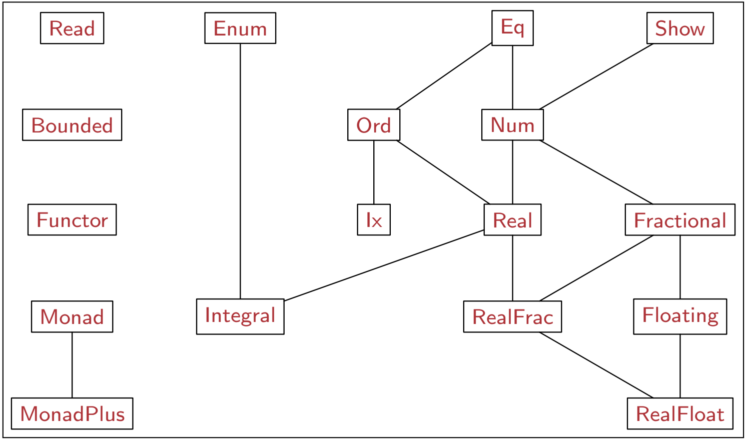
- Type classes define signatures of a set of operations. For example:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

class Eq a => Ord a where    (<), (<=), (>), (>=) ::
    a -> a -> Bool
  min, max :: a -> a -> a
```

- To belong in a type class, a type needs to define all the required operations
- Some standard type classes
  - **Eq**, **Ord**, **Show**, **Read**, **Num**, **Integral**

Haskell 98 class hierarchy



Show and Read classes

```
class Show a where
  show :: a -> String
```

```
class Read a where
  read :: String -> a
```

- Many types are showable and/or readable.

```
> show 10
"10"
> show [1, 2, 3]
"[1,2,3]"
```

```
> read "10" :: Int
10
> read ("[1, 2, 3]") :: [Int]
[1,2,3]
> map (* 2.0) (read ("[1, 2]"))
[2.0,4.0]
```

- Why do some uses of `read` need type annotations?
- Try to evaluate some function names, such as `id` in the interpreter. What happens?

Drills: type these functions

```
second xs = head (tail xs)
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)
```

```
swap (x, y) = (y, x)
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)


pair x y = (x, y)
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)


double x = x * 2
```

Drills: type these functions

```haskell
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)

double :: Num a => a -> a
double x = x * 2
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)

double :: Num a => a -> a
double x = x * 2


palindrome xs = reverse xs == xs
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)

double :: Num a => a -> a
double x = x * 2

palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)

double :: Num a => a -> a
double x = x * 2

palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs


lessThanHalf x y = x * 2 < y
```

Drills: type these functions

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

pair :: a -> b -> (a, b)
pair x y = (x, y)

double :: Num a => a -> a
double x = x * 2

palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs

lessThanHalf :: (Ord a, Num a) => a -> a -> Bool
lessThanHalf x y = x * 2 < y
```