

# Getting Started In Haskell

Wayne Chang (wayne@wycd.net)

Nov 4th, 2017

## Slides and Exercises

With Git installed, you can run this command to download the slides and exercises.

```
$ git clone https://github.com/wyc/haskell-intro.git
```

## Table of Contents

These slides have coverage of:

- Language Overview
- Installation
- Basic Data Types
- Pattern Matching
- Algebraic Data Types
- Type and Value Constructors
- Introduction to Functors
- References and Resources

glhf!

## Introduction

Haskell is *statically-typed*, *functional*, *immutable*, and *lazy*.

## Statically-Typed

- Types are checked at compile time, as opposed to runtime.

## Functional

- “Code is data.”

## Immutable

- Values do not get overwritten.

## Lazy

- Evaluation is deferred for as long as possible.

## Statically-Typed

Types are checked at compile time, as opposed to runtime.

```
int add(int x, int y) {
    return x + y;
}

int main() {
    add(1, 2);
    add(1, "two");
    return 0;
}

$ gcc -o s s.c
s.c: In function 'main':
s.c:9:20: warning: passing argument 2 of 'add'
      makes integer from pointer without a cast
      add(1, "two");
            ^
```

## Functional

“Code is data.”

Functions are passed around as easily as any other values, getting “first-class” treatment. This style allows us operate on whole data structures as opposed to just pieces of them at a time.

```
$ node
> [1, 2, 3, 4, 5]
  .map((elem) => elem * 2)
  .reduce((acc, elem) => acc * elem, 1)
3840
```

## Immutable

Values do not get overwritten, or else you get yelled at.

```
$ node
> const a = 5
undefined
> a = 6
TypeError: Assignment to constant variable.
```

## Immutable

### What's the difference between Caching and Memoization?

- Caching is hoping that the result happens to be in a fast store, and then fetching it if it isn't.
- Memoization is remembering the mapping of specific inputs to their results for a function.

A function can only be memoized if it is *referentially transparent*, otherwise known as *immutable* or *pure*; the same inputs must always produce the same result.

## Lazy

Evaluation is deferred for as long as possible.

“Don't start cleaning the apartment until the doorbell rings, and then only clean the parts that they can see.”

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600,601,602,603,604,605,606,607,608,609,610,611,612,613,614,615,616,617,618,619,620,621,622,623,624,625,626,627,628,629,630,631,632,633,634,635,636,637,638,639,640,641,642,643,644,645,646,647,648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,680,681,682,683,684,685,686,687,688,689,690,691,692,693,694,695,696,697,698,699,700,701,702,703,704,705,706,707,708,709,710,711,712,713,714,715,716,717,718,719,720,721,722,723,724,725,726,727,728,729,730,731,732,733,734,735,736,737,738,739,740,741,742,743,744,745,746,747,748,749,750,751,752,753,754,755,756,757,758,759,760,761,762,763,764,765,766,767,768,769,770,771,772,773,774,775,776,777,778,779,780,781,782,783,784,785,786,787,788,789,790,791,792,793,794,795,796,797,798,799,800,801,802,803,804,805,806,807,808,809,810,811,812,813,814,815,816,817,818,819,820,821,822,823,824,825,826,827,828,829,830,831,832,833,834,835,836,837,838,839,840,841,842,843,844,845,846,847,848,849,850,851,852,853,854,855,856,857,858,859,860,861,862,863,864,865,866,867,868,869,870,871,872,873,874,875,876,877,878,879,880,881,882,883,884,885,886,887,888,889,890,891,892,893,894,895,896,897,898,899,900,901,902,903,904,905,906,907,908,909,910,911,912,913,914,915,916,917,918,919,920,921,922,923,924,925,926,927,928,929,930,931,932,933,934,935,936,937,938,939,940,941,942,943,944,945,946,947,948,949,950,951,952,953,954,955,956,957,958,959,960,961,962,963,964,965,966,967,968,969,970,971,972,973,974,975,976,977,978,979,980,981,982,983,984,985,986,987,988,989,990,991,992,993,994,995,996,997,998,999,1000,1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1011,1012,1013,1014,1015,1016,1017,1018,1019,1020,1021,1022,1023,1024,1025,1026,1027,1028,1029,1030,1031,1032,1033,1034,1035,1036,1037,1038,1039,1040,1041,1042,1043,1044,1045,1046,1047,1048,1049,1050,1051,1052,1053,1054,1055,1056,1057,1058,1059,1060,1061,1062,1063,1064,1065,1066,1067,1068,1069,1070,1071,1072,1073,1074,1075,1076,1077,1078,1079,1080,1081,1082,1083,1084,1085,1086,1087,1088,1089,1090,1091,1092,1093,1094,1095,1096,1097,1098,1099,1100,1101,1102,1103,1104,1105,1106,1107,1108,1109,1110,1111,1112,1113,1114,1115,1116,1117,1118,1119,1120,1121,1122,1123,1124,1125,1126,1127,1128,1129,1130,1131,1132,1133,1134,1135,1136,1137,1138,1139,1140,1141,1142,1143,1144,1145,1146,1147,1148,1149,1150,1151,1152,1153,1154,1155,1156,1157,1158,1159,1160,1161,1162,1163,1164,1165,1166,1167,1168,1169,1170,1171,1172,1173,1174,1175,1176,1177,1178,1179,1180,1181,1182,1183,1184,1185,1186,1187,1188,1189,1190,1191,1192,1193,1194,1195,1196,1197,1198,1199,1200,1201,1202,1203,1204,1205,1206,1207,1208,1209,1210,1211,1212,1213,1214,1215,1216,1217,1218,1219,1220,1221,1222,1223,1224,1225,1226,1227,1228,1229,1230,1231,1232,1233,1234,1235,1236,1237,1238,1239,1240,1241,1242,1243,1244,1245,1246,1247,1248,1249,1250,1251,1252,1253,1254,1255,1256,1257,1258,1259,1260,1261,1262,1263,1264,1265,1266,1267,1268,1269,1270,1271,1272,1273,1274,1275,1276,1277,1278,1279,1280,1281,1282,1283,1284,1285,1286,1287,1288,1289,1290,1291,1292,1293,1294,1295,1296,1297,1298,1299,1300,1301,1302,1303,1304,1305,1306,1307,1308,1309,1310,1311,1312,1313,1314,1315,1316,1317,1318,1319,1320,1321,1322,1323,1324,1325,1326,1327,1328,1329,1330,1331,1332,1333,1334,1335,1336,1337,1338,1339,1340,1341,1342,1343,1344,1345,1346,1347,1348,1349,1350,1351,1352,1353,1354,1355,1356,1357,1358,1359,1360,1361,1362,1363,1364,1365,1366,1367,1368,1369,1370,1371,1372,1373,1374,1375,1376,1377,1378,1379,1380,1381,1382,1383,1384,1385,1386,1387,1388,1389,1390,1391,1392,1393,1394,1395,1396,1397,1398,1399,1400,1401,1402,1403,1404,1405,1406,1407,1408,1409,1410,1411,1412,1413,1414,1415,1416,1417,1418,1419,1420,1421,1422,1423,1424,1425,1426,1427,1428,1429,1430,1431,1432,1433,1434,1435,1436,1437,1438,1439,1440,1441,1442,1443,1444,1445,1446,1447,1448,1449,1450,1451,1452,1453,1454,1455,1456,1457,1458,1459,1460,1461,1462,1463,1464,1465,1466,1467,1468,1469,1470,1471,1472,1473,1474,1475,1476,1477,1478,1479,1480,1481,1482,1483,1484,1485,1486,1487,1488,1489,1490,1491,1492,1493,1494,1495,1496,1497,1498,1499,1500,1501,1502,1503,1504,1505,1506,1507,1508,1509,1510,1511,1512,1513,1514,1515,1516,1517,1518,1519,1520,1521,1522,1523,1524,1525,1526,1527,1528,1529,1530,1531,1532,1533,1534,1535,1536,1537,1538,1539,1540,1541,1542,1543,1544,1545,1546,1547,1548,1549,1550,1551,1552,1553,1554,1555,1556,1557,1558,1559,1560,1561,1562,1563,1564,1565,1566,1567,1568,1569,1570,1571,1572,1573,1574,1575,1576,1577,1578,1579,1580,1581,1582,1583,1584,1585,1586,1587,1588,1589,1590,1591,1592,1593,1594,1595,1596,1597,1598,1599,1600,1601,1602,1603,1604,1605,1606,1607,1608,1609,1610,1611,1612,1613,1614,1615,1616,1617,1618,1619,1620,1621,1622,1623,1624,1625,1626,1627,1628,1629,1630,1631,1632,1633,1634,1635,1636,1637,1638,1639,1640,1641,1642,1643,1644,1645,1646,1647,1648,1649,1650,1651,1652,1653,1654,1655,1656,1657,1658,1659,1660,1661,1662,1663,1664,1665,1666,1667,1668,1669,1670,1671,1672,1673,1674,1675,1676,1677,1678,1679,1680,1681,1682,1683,1684,1685,1686,1687,1688,1689,1690,1691,1692,1693,1694,1695,1696,1697,1698,1699,1700,1701,1702,1703,1704,1705,1706,1707,1708,1709,1710,1711,1712,1713,1714,1715,1716,1717,1718,1719,1720,1721,1722,1723,1724,1725,1726,1727,1728,1729,1730,1731,1732,1733,1734,1735,1736,1737,1738,1739,1740,1741,1742,1743,1744,1745,1746,1747,1748,1749,1750,1751,1752,1753,1754,1755,1756,1757,1758,1759,1760,1761,1762,1763,1764,1765,1766,1767,1768,1769,1770,1771,1772,1773,1774,1775,1776,1777,1778,1779,1780,1781,1782,1783,1784,1785,1786,1787,1788,1789,1790,1791,1792,1793,1794,1795,1796,1797,1798,1799,1800,1801,1802,1803,1804,1805,1806,1807,1808,1809,1810,1811,1812,1813,1814,1815,1816,1817,1818,1819,1820,1821,1822,1823,1824,1825,1826,1827,1828,1829,1830,1831,1832,1833,1834,1835,1836,1837,1838,1839,1840,1841,1842,1843,1844,1845,1846,1847,1848,1849,1850,1851,1852,1853,1854,1855,1856,1857,1858,1859,1860,1861,1862,1863,1864,1865,1866,1867,1868,1869,1870,1871,1872,1873,1874,1875,1876,1877,1878,1879,1880,1881,1882,1883,1884,1885,1886,1887,1888,1889,1890,1891,1892,1893,1894,1895,1896,1897,1898,1899,1900,1901,1902,1903,1904,1905,1906,1907,1908,1909,1910,1911,1912,1913,1914,1915,1916,1917,1918,1919,1920,1921,1922,1923,1924,1925,1926,1927,1928,1929,1930,1931,1932,1933,1934,1935,1936,1937,1938,1939,1940,1941,1942,1943,1944,1945,1946,1947,1948,1949,1950,1951,1952,1953,1954,1955,1956,1957,1958,1959,1960,1961,1962,1963,1964,1965,1966,1967,1968,1969,1970,1971,1972,1973,1974,1975,1976,1977,1978,1979,1980,1981,1982,1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021,2022,2023,2024,2025,2026,2027,2028,2029,2030,2031,2032,2033,2034,2035,2036,2037,2038,2039,2040,2041,2042,2043,2044,2045,2046,2047,2048,2049,2050,2051,2052,2053,2054,2055,2056,2057,2058,2059,2060,2061,2062,2063,2064,2065,2066,2067,2068,2069,2070,2071,2072,2073,2074,2075,2076,2077,2078,2079,2080,2081,2082,2083,2084,2085,2086,2087,2088,2089,2090,2091,2092,2093,2094,2095,2096,2097,2098,2099,2100,2101,2102,2103,2104,2105,2106,2107,2108,2109,2110,2111,2112,2113,2114,2115,2116,2117,2118,2119,2120,2121,2122,2123,2124,2125,2126,2127,2128,2129,2130,2131,2132,2133,2134,2135,2136,2137,2138,2139,2140,2141,2142,2143,2144,2145,2146,2147,2148,2149,2150,2151,2152,2153,2154,2155,2156,2157,2158,2159,2160,2161,2162,2163,2164,2165,2166,2167,2168,2169,2170,2171,2172,2173,2174,2175,2176,2177,2178,2179,2180,2181,2182,2183,2184,2185,2186,2187,2188,2189,2190,2191,2192,2193,2194,2195,2196,2197,2198,2199,2200,2201,2202,2203,2204,2205,2206,2207,2208,2209,2210,2211,2212,2213,2214,2215,2216,2217,2218,2219,2220,2221,2222,2223,2224,2225,2226,2227,2228,2229,2230,2231,2232,2233,2234,2235,2236,2237,2238,2239,2240,2241,2242,2243,2244,2245,2246,2247,2248,2249,2250,2251,2252,2253,2254,2255,2256,2257,2258,2259,2260,2261,2262,2263,2264,2265,2266,2267,2268,2269,2270,2271,2272,2273,2274,2275,2276,2277,2278,2279,2280,2281,2282,2283,2284,2285,2286,2287,2288,2289,2290,2291,2292,2293,2294,2295,2296,2297,2298,2299,2300,2301,2302,2303,2304,2305,2306,2307,2308,2309,2310,2311,2312,2313,2314,2315,2316,2317,2318,2319,2320,2321,2322,2323,2324,2325,2326,2327,2328,2329,2330,2331,2332,2333,2334,2335,2336,2337,2338,2339,2340,2341,2342,2343,2344,2345,2346,2347,2348,2349,2350,2351,2352,2353,2354,2355,2356,2357,2358,2359,2360,2361,2362,2363,2364,2365,2366,2367,2368,2369,2370,2371,2372,2373,2374,2375,2376,2377,2378,2379,2380,2381,2382,2383,2384,2385,2386,2387,2388,2389,2390,2391,2392,2393,2394,2395,2396,2397,2398,2399,2400,2401,2402,2403,2404,2405,2406,2407,2408,2409,2410,2411,2412,2413,2414,2415,2416,2417,2418,2419,2420,2421,2422,2423,2424,2425,2426,2427,2428,2429,2430,2431,2432,2433,2434,2435,2436,2437,2438,2439,2440,2441,2442,2443,2444,2445,2446,2447,2448,2449,2450,2451,2452,2453,2454,2455,2456,2457,2458,2459,2460,2461,2462,2463,2464,2465,2466,2467,2468,2469,2470,2471,2472,2473,2474,2475,2476,2477,2478,2479,2480,2481,2482,2483,2484,2485,2486,2487,2488,2489,2490,2491,2492,2493,2494,2495,2496,2497,2498,2499,2500,2501,2502,2503,2504,2505,2506,2507,2508,2509,2510,2511,2512,2513,2514,2515,2516,2517,2518,2519,2520,2521,2522,2523,2524,2525,2526,2527,2528,2529,2530,2531,2532,2533,2534,2535,2536,2537,2538,2539,2540,2541,2542,2543,2544,2545,2546,2547,2548,2549,2550,2551,2552,2553,2554,2555,2556,2557,2558
```

## Goodbye to Pretense

You don't need category theory, abstract algebra, or graduate-level algorithms to use or understand Haskell.

## Hello, functional programming

To to remove the first three elements:

OOP-like:

```
[1,2,3,4,5].drop(3)  // [4,5]
```

Functional:

```
drop 3 [1,2,3,4,5]  -- [4,5]
```

In functional programming, we focus on the functions! When you focus on the objects themselves, you're "object-oriented".

## Functional programming

Focus on functions.

- We ask "what can this function operate on?" and not "what can this object do?"
- What if we didn't have countless POJOs, and instead some core functions that we can use over and over again?

Composability is key!

```
Prelude> map (drop 1) [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
[[2,3],[5,6],[8,9]]
```

## Getting Started (1/5)

1. Install Stack and the Glasgow Haskell Compiler (GHC) for GNU/Linux, Mac OSX, or Windows from <https://docs.haskellstack.org/en/stable/README/>
2. Setup Stack.  

```
$ stack setup
```
3. Write `hello.hs`:

```
main :: IO ()
main = do
  putStrLn "Hello, World!"
```

## Getting Started (2/5)

4. Compile it and run!

```
$ stack ghc hello.hs
$ ./hello
Hello, World!
```

5. Alternatively, you can use runhaskell:

```
$ stack runhaskell hello.hs
Hello, World!
```

## Getting Started (3/5)

7. Write a function to use in GHC's interactive environment, ghci.  
myfuncs.hs:

```
salesTax :: Double -> Double
salesTax price = price * 0.089
```

8. Enter the ghci shell and load your source file.

```
$ stack ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
Prelude> :load myfuncs
[1 of 1] Compiling Main ( myfuncs.hs, interpreted )
Ok, modules loaded: Main.
*Main> salesTax 100
8.9
```

## Getting Started (4/5)

7. Importing modules:

```
-- Haskell
import Data.Char
import Data.Char (ord)
import qualified Data.Char as C
```

```

ord 'a'      -- 97
C.ord 'a'    -- 97

# Python
import data.char
import ord from data.char
import data.char as C

ord('a')     # 97
C.ord('a')    # 97

```

## Getting Started (5/5)

8. Try importing from `ghci`.

```

$ stack ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
Prelude> import Data.Char (ord)
Prelude Data.Char> ord 'a'
97

```

## Basic Types

Types start with a Capital letter, variables (including functions and type variables) start with lowercase.

- `Int`  
– Fixed-Precision Whole Number
- `Integer`  
– Unbounded Whole Number
- `Float`  
– Single-Precision Floating
- `Double`  
– Double-Precision Floating
- `Bool`  
– True or False
- `Char`  
– Single character, such as `'a'`

## Lists (1/2)

- `List`

- For example, list of `Char` looks like `[Char]`
- The type resembles generic lists in Java such as `List<T>` (instantiated to `List<Character>`), but that's where the similarities end.
- Lisp-style construction (Horribly inefficient for, e.g., binary data? Yes. More on this later.) Illustration in JSON:

```
{
  'value': 5,
  'next': {
    'value': 4,
    'next': {
      value: 3,
      'next': null
    }
  }
}
```

## Lists (2/2)

- `List` (continued...)
  - Literal notation: `[1,2,3]`
  - Syntactic sugar for: `1:2:3:[]`, where `:` prepends a single element to a list, and `[]` is an empty list:
 

```
Prelude> 1:[2,3]
[1,2,3]
```
  - List items must all be of the same type. Heterogenous lists are possible using advanced language features, but they are usually an anti-pattern. When have you ever been happy to see `List<Object>` in Java?

## Operations on Lists (1/4)

- Take first element with `head`:
 

```
Prelude> head [1,2,3]
1
Prelude> head []
*** Exception: Prelude.head: empty list
```
- Take *n*th element using the index (`!!`) operator:
 

```
Prelude> [10, 11, 12] !! 0
10
Prelude> [10, 11, 12] !! 2
12
```

```
Prelude> [10, 11, 12] !! 5
*** Exception: Prelude.!!: index too large
```

## Operations on Lists (2/4)

- Check for the presence of a value using `elem`:

```
Prelude> elem 99 [42, 52, 62]
False
Prelude> elem 42 [42, 52, 62]
True
```

- Reverse a list with `reverse`:

```
Prelude> reverse [1,2,3,4]
[4,3,2,1]
```

## Operations on Lists (3/4)

- Add two lists together with `++`:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
Prelude> "type" ++ "writer"
"typewriter"
```

- Grab the first  $n$  elements with `take`:

```
Prelude> take 2 [1,2,3,4,5]
[1,2]
```

## Operations on Lists (4/4)

- Mapping a function across a list with `map`:

```
Prelude> let addFive x = x + 5
Prelude> map addFive [1,2,3]
[6,7,8]
```

- Reducing a list with `foldl`:

```
Prelude> foldl (\acc elem -> acc + elem) 0 [1,2,3]
6
```



## Tuples (1/2)

- Tuple
  - Store several values into a single value. They can be of any type, but you must know how many values there will be.
  - For example,

```
Prelude> let point = (3,4)
Prelude> fst point
3
Prelude> snd point
4
Prelude> let point3d = (3,4,5)
Prelude> let location = ("Germany", (51.1657, 10.4515))
Prelude> fst location
"Germany"
Prelude> snd location
(51.1657, 10.4515)
```

## Tuples (2/2)

- Tuple (continued...)
  - An associative list is [(a, b)], for example, one of type [(String, Int)]:

```
Prelude> let eatingContestScores =
      [("Jimmy", 5), ("Sally", 8)]
Prelude> lookup "Jimmy" eatingContestScores
Just 5
```
  - Tuples are of different types when they vary in size or constituent types:

```
Prelude> (5, 5) :: (Int, Int)
Prelude> (5, 5, 5) :: (Int, Int, Int)
Prelude> (5, "Five") :: (Int, String)
```

## Reading Types (1/5)

`::` can generally be read as “has type”, and it is a more of a declaration than an observation.

```
-- 5 ``has type'' Int
Prelude> 5 :: Int
-- 5 ``has type'' Integer
Prelude> 5 :: Integer
-- 5 ``has type'' String...uh oh, we can't do that!
```

```
Prelude> 5 :: String
```

```
<interactive>:1:1: error:
```

- No **instance** for (Num String) arising from the literal '5'
- In the expression: 5 :: String  
In an equation for 'it': it = 5 :: String

## Reading Types (2/5)

The *function arrow* or *function operator* `->` denotes a function that takes an argument of the type on the left and returns a value of the type on the right.

```
-- addFive ``has type`` function that takes
--   an Int and returns an Int
addFive :: Int -> Int
addFive x = x + 5

-- mkAdder ``has type`` function that takes
--   an Int and returns a (function that takes
--   an Int and returns an Int)
mkAdder :: Int -> (Int -> Int)
mkAdder x = (\y -> y + x)
addFive = (mkAdder 5)
```

## Reading Types (3/5)

More practice reading types.

```
import Data.Char (toLower)
-- toLower ``has type`` function that takes a Char
--   and returns a Char
toLower :: Char -> Char

-- lowerString ``has type`` function that takes
--   a list of characters and returns a list of
--   characters
lowerString :: [Char] -> [Char]
lowerString = map toLower

Prelude> map toLower "ABCDE"
"abcde"
```

```
Prelude> lowerString "ABCDE"
"abcde"
```

## Reading Types (4/5)

You can use `:t` in GHCi to ask about an expression's type.

```
Prelude> :t "Hello, World!"
"Hello, World!" :: [Char]
-- String and [Char] are the same type
```

```
Prelude> let addFive x = x + 5
Prelude> :t addFive
addFive :: Num a => a -> a
Prelude> addFive 1.2
6.2
```

## Reading Types (5/5)

Locking down the type:

```
Prelude> :t (addFive :: Int -> Int)
(addFive :: Int -> Int) :: Int -> Int

Prelude> (addFive :: Int -> Int) 1.2 -- BOOM!
<interactive>:3:25: error:
    • No instance for (Fractional Int) arising
      from the literal '1.2'
    • In the first argument of
      'addFive :: Int -> Int', namely '1.2'
   In the expression:
      (addFive :: Int -> Int) 1.2
   In an equation for 'it':
      it = (addFive :: Int -> Int) 1.2
```

## Currying (1/3)

Currying is the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.

```
-- Uncurried
add (x, y) = x + y
add (5, 2)
```

Curried forms are preferred because they allow for partial application:

```
-- Curried
add x y = x + y
addFive = add 5
addFive 2
```

## Currying (2/3)

Curried 1, 2, and 3 are equivalent.

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

-- Curried 1
add :: Int -> Int -> Int
add x y = x + y

-- Curried 2
add :: Int -> (Int -> Int)
add x = (\y -> x + y)

-- Curried 3
add :: (Int -> (Int -> Int))
add = (\x -> (\y -> x + y))
```

## Currying (3/3)

The arrow operator is right-associative. Each value we apply returns a value conforming to “the rest” of the type signature.

```
a -> a -> a -> a
a -> a -> (a -> a)
a -> (a -> (a -> a))
```

Function application is left-associative due to currying. It “cancels out” because we apply values one at a time.

```
f a b c
(f a) b c
```

```
((f a) b) c
```

## Reading the Type of map (1/3)

```
Prelude> import Data.Char (toLower)
```

```
Prelude Data.Char> :t map
map :: (a -> b) -> [a] -> [b]
```

```
-- `a' and `b' are known as type variables,
-- because they can represent any type. They
-- are uncapitalized to distinguish them
-- from specific types such as `Int'.
```

```
Prelude Data.Char> :t toLower
toLower :: Char -> Char
```

```
Prelude Data.Char> :t map toLower
map toLower :: [Char] -> [Char]
```

## Reading the Type of map (2/3)

Hindley–Milner type inference!

1. Pass `toLower` to `map`.

```
map :: (a -> b) -> [a] -> [b]
toLower :: (Char -> Char)
```

2. See if there could be a fit.

```
(a -> b) -> [a] -> [b]
(Char -> Char)
```

3. There it is! Substitute `let a = Char, b = Char` in

```
(Char -> Char) -> [Char] -> [Char]
(Char -> Char)
```

4. “Subtract,” and then our result is

```
[Char] -> [Char]
map toLower :: [Char] -> [Char]
```

## Reading the Type of map (3/3)

We've just done partial function application!

```
map :: (a -> b) -> [a] -> [b]
lowerString = (map toLower :: [Char] -> [Char])
```

lowerString is the resulting function of a partially-applied map.

map has a and b so we can change the type of the list.

```
Prelude> :t even
even :: Int -> Bool
Prelude> :t [1,2,3]
[1,2,3] :: [Int]

Prelude> map even [1,2,3]
[False,True,False]
Prelude> :t [False,True,False]
[False,True,False] :: [Bool]
```

## Reading Typeclasses (1/5)

Num is a typeclass that defines some common numeric operations. Think interfaces in Java or Go, except that typeclasses apply to types.

```
Prelude> let addFive x = x + 5
Prelude> :t addFive
addFive :: Num a => a -> a
Prelude> addFive 1.2
6.2
```

## Reading Typeclasses (2/5)

List function examples with type variables:

```
fst :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

zip :: [a] -> [b] -> [(a,b)]

id :: a -> a
```

## Reading Typeclasses (3/5)

Abbreviated examples:

```
sum :: Num a => [a] -> a
sum = foldl (+) 0

-- Num - Numeric Types
(+) :: Num a => a -> a -> a
-- Eq - Equality Types
(==) :: Eq a => a -> a -> Bool
-- Ord - Ordered Types
(<) :: Ord a => a -> a -> Bool
```

## Reading Typeclasses (4/5)

Nearly-complete definition from the GHC source code:

```
-- | Basic numeric class.
class Num a where
    (+), (-), (*)      :: a -> a -> a
    -- | Unary negation.
    negate            :: a -> a
    -- | Absolute value.
    abs               :: a -> a
    -- | Sign of a number.
    signum            :: a -> a
    -- | Conversion from an 'Integer'.
    fromInteger       :: Integer -> a

    x - y              = x + negate y
    negate x           = 0 - x
```

## Reading Typeclasses (5/5)

Typeclasses are useful because they allow us to define functions that can work across a variety of types sharing common properties.

```
class Show a where
    show      :: a -> String
    -- ...more function requirements

instance Show Int where
    show = showSignedInt' -- Some low-level function
```

```

prefixedShow :: (Show a) => String -> a -> String
prefixedShow prefix x = prefix ++ ": " ++ show x

prefixedShow "MyValue" 42      -- "MyValue: 42"
prefixedShow "MyPet" "cat"    -- "MyPet: \"cat\""
prefixedShow "MyPoint" (4,5)  -- "MyPoint: (4,5)"

```

## Infix Functions (1/3)

Functions can be put in between arguments like this:

```

Prelude> 5 + 5
10

```

They are of type `a -> b -> c`.

```

Prelude> :t (+)
(+) :: Num a => a -> a -> a

```

When surrounded by parentheses, they are considered in *prefix notation*, meaning that they are called like any other non-infix functions.

```

Prelude> (+) 5 5
10

```

## Infix Functions (2/3)

Likewise, non-infix functions matching type `a -> b -> c` may be used as infix functions.

```

Prelude> mod 12 5
2
Prelude> 12 `mod` 5
2

```

This is all just syntactic sugar.

## Infix Functions (3/3)

In other languages such as C, infix functions are built in. In Haskell, we can define our own.



```

infixr 0 ~ -- right associative
-- a ~ b ~ c becomes a ~ (b ~ c)

infixl 0 ~ -- left associative
-- a ~ b ~ c becomes (a ~ b) ~ c

infix 0 ~ -- non-associative
-- a ~ b ~ c becomes (a ~ b) ~ c OR a ~ (b ~ c)

```

They have associativity and precedence (0 the weakest, 9 the strongest).

## Anonymous Functions

Anonymous functions help us not think about names. They can be defined as follows:

```

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

```

```

addFive :: Int -> Int
addFive = \x -> add x 5

```

They are expressions, and can be used as inputs to functions that take functions as arguments or assigned to variables.

```

Prelude> map (\x -> x + 5) [1,2,3]
[6,7,8]
Prelude> let double = (\x -> x * 2) in double 5
10

```

## Evaluation Order

When in doubt, use parentheses.

```

Prelude> 5 - 3 * 2
-1
Prelude> (5 - 3) * 2
4

```

You can then look up the associativity and precedence per operator to reason about what the execution would be.

## Symbols Related to Evaluation Order

You will encounter two symbols that help dictate order of evaluation, `$` is known as *application*, and `.` is known as *function composition*.

- `($)` calls the function which is its left-hand argument on the value which is its right-hand argument.

```
map addFive (map addFive [1,2,3])
-- Same as
map addFive $ map addFive [1,2,3]
```

- `(.)` composes the function which is its left-hand argument on the function which is its right-hand argument.

```
addFive (double 5)
-- Same as
(addFive . double) 5
```

(Source: ellisbben, StackOverflow)

## PITFALL: Mismatched Arguments (1/3)

Watch out for what gets passed in as arguments!

```
Prelude> map addFive map addFive [1,2,3]
```

```
<interactive>:3:1: error:
```

- Couldn't match expected type `'(Integer -> Integer) -> [Integer] -> t'` with actual type `'[Integer]'`
- The function `'map'` is applied to four arguments, but its type `'(Integer -> Integer) -> [Integer] -> [Integer]'` has only two  
In the expression:  
    map addFive map addFive [1, 2, 3]  
In an equation for `'it'`:  
    it = map addFive map addFive [1, 2, 3]
- Relevant bindings include  
    it :: t (bound at <interactive>:3:1)

## PITFALL: Mismatch Arguments (2/3)

```
-- ...continued
<interactive>:3:13: error:
• Couldn't match expected type
  '[Integer]'
  with actual type
  '(a0 -> b0) -> [a0] -> [b0]'
• Probable cause: 'map' is applied to too few arguments
  In the second argument of 'map', namely 'map'
  In the expression:
    map addFive map addFive [1, 2, 3]
  In an equation for 'it':
    it = map addFive map addFive [1, 2, 3]
```

## PITFALL: Mismatched Arguments (3/3)

```
map addFive map addFive [1,2,3]
|-----^-----^-----^
  1         2   3         4
```

Instead, do:

```
map addFive (map addFive [1,2,3])
|-----^-----^
  1         2
```

Or

```
map addFive $ map addFive [1,2,3]
|-----^-----^
  1         2
```

## Pattern Matching

Pattern matching is an extremely useful technique that is a staple in functional programming languages. It allows us to “search” for the data we want from an expression. We will cover:

- Pattern matching constants, lists, and tuples
- Using pattern matching in
  - Function Definitions (we’ve already been using them!)
  - Guards
  - **where**
  - **let**

```
- case...of
```

Pattern Matching in Haskell is top to bottom, left to right.

## Pattern Matching: In JavaScript

Here is an example of code with and without destructuring syntax in ES6:

```
const user = {name: "Jones", age: 50,
              email: "jones@jones.com"}
const name = user.name
const age  = user.age
const email = user.email
```

With pattern matching:

```
const user = {name: "Jones", age: 50,
              email: "jones@jones.com"}
const {name, age, email} = user
```

While destructuring isn't pattern matching, it has a similar intent, and there are even proposals to add true pattern matching to the language (see references).

## Pattern Matching: Matching Constants (1/2)

Somewhere in `test.hs`:

```
guess :: Int -> Bool
guess 5 = True

(guess 5)  -- True
(guess 10) -- BOOM!
-- *** Exception: Non-exhaustive
--           patterns in function guess
```

GHC can warn us about this before it happens!

```
$ ghc -Wall test.hs
test.hs:83:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'guess':
        Patterns not matched:
            p where p is not one of {5}
```

## Pattern Matching: Matching Constants (2/2)

Fixed, in `test.hs`:

```
guess :: Int -> Bool
guess 5 = True
guess _ = False
```

- Underscore (`_`) means “throw this value out, we’re not going to use it.”
- Patterns are tried sequentially until there’s a match.
- If no match could be found, then an exception is thrown. Therefore, it’s good practice to have catch-all patterns at the end.

## Pattern Matching: Matching Lists

Lists can be matched in a way that grants you their first few elements, which is very useful in recursive functions. The form is `(x:xs)`.

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = (f x):(map f xs)
map f []     = []    -- Cover all the cases
```

If necessary, we can also specify a handle for the whole list:

```
addListLengthToFirstElement :: [Int] -> Int
addListLengthToFirstElement lst@(x:_) = length lst + x
addListLengthToFirstElement [] = error "empty list"
```

## PITFALL: Matching Lists

Be wary of these syntax fails that can lead to confusing compiler errors:

```
sum [x:xs] = x + sum xs -- NO
```

```
sum x:xs = x + sum xs   -- NO
```

```
sum (x:xs) = x + sum xs -- ok
```

## Pattern Matching: Matching Tuples

We can pull the values right out of tuples using pattern matching.

```

dist :: (Double, Double) -> (Double, Double) -> Double
dist (x, y) (x', y') =
    sqrt $ (x - x') ** 2 + (y - y') ** 2
Prelude> dist (0.0, 0.0) (1.0, 1.0)
1.4142135623730951

```

Even with nesting:

```

showIdAndFullName :: (Int, (String, String)) -> String
showIdAndFullName (id', (firstName, lastName)) =
    show id' ++ ": " ++ firstName ++ " " ++ lastName
Prelude> showIdAndFullName (5, ("Rebecca", "Jones"))
"5: Rebecca Jones"

```

## Pattern Not-Matching: Guards

Guards are ways to test whether some properties are true or false. They don't actually do pattern matching. However, they can be used with the `where` clause to do pattern matching. They are evaluated sequentially.

```

scovilleComment :: Int -> String
scovilleComment score
    | score < 0 = "Huh?"
    | score <= 300 = "What spice?"
    | score <= 700 = "Eh...mild." -- Frank's
    | score <= 3500 = "A bit of kick!" -- Tabasco
    | score <= 10000 = "Mmm, spicy." -- Orange Kush
    | score <= 1500000 = "WOW THAT'S HOT" -- Pyro Diablo
    | otherwise = "OMG"

```

## PITFALL: Guards

Do not put an equal sign after the function name and variables:

```

-- NO
guess x =
    | x == 5 = "You got it!"
    | otherwise = "Wrong!"

-- ok
guess x
    | x == 5 = "You got it!"
    | otherwise = "Wrong!"

```

```
-- ok
guess x | x == 5    = "You got it!"
        | otherwise = "Wrong!"
```

## Pattern Matching: where

`where` can be used to help things read nicer and avoid repetition.

```
totalCompensation :: Int -> Int -> Int -> Int -> Int
totalCompensation base yearsEmployed rank performance =
    base + bonus
    where rankMultiplier = (yearsEmployed / rank) `mod` 2
          bonus = rankMultiplier * performance * 10000
```

We can see right away that `totalCompensation` simply returns `base + bonus`, but if we want more detail, then we can dig in to the `where` clause. They can also be used to pattern match:

```
showPoint :: (Double, Double) -> String
showPoint p = "In " ++ show p ++ " " ++ x' ++ "," ++ y'
    where (x, y) = p
          x' = "x=" ++ show x
          y' = "y=" ++ show y
Prelude> showPoint (1,2)
"In (1,2) x=1, y=2"
```

## Pattern Matching: let

`let` can be used to pattern match in the form of:

```
let <bindings> in <expression>
```

`let` itself is an expression and therefore produces a value, which is an important difference from `where`.

```
quadratic :: Double -> Double -> Double -> Int -> Double
quadratic a b c sign =
    let sign' = if sign >= 0 then 1.0 else -1.0
        rootTerm = sqrt $ (b**2) - 4*a*c
        numerator = (- b) + (sign') * rootTerm
        denominator = 2 * a
    in
        numerator / denominator
```

## Pattern Matching: `case...of`

Case expressions can be thought as guards that do pattern matching instead of property testing. Like `let`, they are expressions.

```
case expression of pattern -> result
                    pattern -> result
                    pattern -> result
                    ...
```

(Source: Learn You a Haskell)

```
showPointType :: (Double, Double) -> String
showPointType p = "Point Type: " ++ case p of
    (0, 0)    -> "Origin"
    (_, 0)    -> "Y-Intercept"
    (0, _)    -> "X-Intercept"
    otherwise -> "Normal Point"
Prelude> showPointType (0, 0)
"Point Type: Origin"
```

## `if...then...else`

`if...then...else` constructs in Haskell are better thought of as *if expressions* than *if statements*. They evaluate to a value depending on what comes after `if`. They cannot be used to pattern match.

```
addIfEven :: Int -> Int -> Int
addIfEven x y = x + if even y then y else 0
```

## PITFALL: Sometimes spacing matters

```
zeroIfEven :: Int -> Int

zeroIfEven x = -- NO
    if even x then
        0
    else
        x

zeroIfEven x = -- ok
    if even x
-- ^ Code Block 1
    then 0
```



```
    else x
--    ^ Code Block 2
```

## Sum Types (1/7)

If we have three types of people: employees, managers, and clients, how do we cleanly represent them? Let's say we're using Go.

```
type PersonType string

const (
    EmployeePersonType PersonType = "Employee"
    ManagerPersonType  PersonType = "Manager"
    ClientPersonType   PersonType = "Client"
)
```

## Sum Types (2/7)

```
type PersonType string
```

The primary benefit is that the compiler gives us some type checks for PersonType.

```
func AmIOkay(pt PersonType) bool {
    switch pt {
    case ManagerPersonType:
        fallthrough
    case ClientPersonType:
        return true
    case EmployeePersonType:
        return false
    default:
        panic(fmt.Sprintf(
            "Unknown PersonType: %s", pt))
    }
}
```

## Sum Types (3/7)

```
type PersonType string
```

Disadvantages:

- Strings are expensive to pass around
- Strings are OVERKILL.  $c^{strlen}$  possibilities!
- Someone can just decide to invent a new type via cast: `PersonType("Daughter")`
- No exhaustiveness checks

We could consider using enum-like `iota`, but the last 3 points would still exist.

## Sum Types (4/7)

```
type PersonType string
```

What if we forget one of the checks?

```
-         case EmployeePersonType:
-             return false
```

Runtime error!

```
$ go run test.go
```

```
panic: Unknown PersonType: Employee
```

```
goroutine 1 [running]:
panic(0x48c5c0, 0xc42000a320)
```

## Sum Types (5/7)

Enter sum types!

```
-- Person is a sum type with zero-argument value
-- constructors of Employee, Manager, and Client.
data PersonType = Employee | Manager | Client

greet :: PersonType -> String
greet pt = case pt of
    Employee -> "G'day, employee!"
    Manager  -> "Hello, manager."

main :: IO ()
main = do
    putStrLn $ greet Client
```

## Sum Types (6/7)

Run it:

```
$ ghc -Wall test.hs
test.hs:33:12: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In a case alternative: Patterns not matched: Client

$ ./test
test: test.hs:(33,12)-(35,32):
    Non-exhaustive patterns in case
```

## Sum Types (7/7)

Fixed:

```
greet :: PersonType -> String
greet pt = case pt of
    Employee -> "G'day, employee!"
    Manager -> "Hello, manager."
    Client -> "Hey!"
```

```
$ ghc -Wall test.hs
$ ./test
Hey!
```

## Product Types (1/2)

Sum types can be combined with product types to store additional data:

```
data ApptTime = NotScheduled | WalkIn | At UTCTime
--
--           ^
--           |
--           | This product type combines At
--           | and UTCTime to form a new type.
--           | They're "glued" together.
```

```
apptStatus :: ApptTime -> String
apptStatus apptTime = case apptTime of
    NotScheduled -> "Not currently scheduled."
    WalkIn -> "Walk in during business hours."
    At time -> "Arrive by " ++ show time
```

The data can be accessed using pattern matching, as seen with `time`.

## Product Types (2/2)

Running our latest example:

```
-- ...

main :: IO ()
main = do
    currentTime <- getCurrentTime
    putStrLn $ apptStatus (At currentTime)

$ ghc -Wall test.hs
$ ./test
Arrive by 2017-05-08 10:04:09.234585398 UTC
```

## Algebraic Data Types

Sum and product types together make up the *algebraic data types* (ADTs) in Haskell. These are not to be confused with *abstract data types*. There are also *generic algebraic data types* (GADTs), but those are out of scope for this presentation.

```
-- Sum Types are combined with '|',
--   behaving like an "OR" operator
data MySumType = A | B
-- Product Types are combined with ' ',
--   behaving like an "AND" operator
data MyProductType = C D
```

## Namespaces and Constructors (1/12)

Haskell has two namespaces:

- One for *values*.
- One for *types*.

Constructors:

- A data/value constructor is a “function” that takes 0 or more values and gives you back a new value.
- A type constructor is a “function” that takes 0 or more types and gives you back a new type.

(Source: kqr, StackOverflow)

## Namespaces and Constructors (2/12)

The `data` operator constructs a new type.

```
data Color = Red | Green | Blue
Red :: Color
Green :: Color
Blue :: Color

data Color = RGB Int Int Int
RGB :: Int -> Int -> Int -> Color
```

For all practical purposes you can just think of data constructors as constants belonging to a type.

## Namespaces and Constructors (3/12)

### Same-named constructors

```
data Person = Person String Int
--      ^      ^
--      |      |
--      |      | Here we declared and defined a
--      |      | value constructor named ``Person''.
--      |      | To construct the value, we need a String value
--      |      | followed by an Int value. It takes two value
--      |      | arguments, so it is ``binary''.
--      |
--      | Here we declared and defined a type constructor
--      | named ``Person''. It takes no type arguments and is
--      | therefore considered ``nullary''.
```

## Namespaces and Constructors (4/12)

```
data Point a = Point a a
--      ^      ^
--      |      |
--      |      | Created a binary value constructor named
--      |      | ``Point'' which takes two value arguments
--      |      | of type a.
--      |
--      | Created a type constructor named ``Point''
--      | that takes one type argument such as Int or
--      | Double. It is ``unary''.
```

```
Prelude> :t Point (3.0 :: Double) (4.0 :: Double)
--           ^ value constructor
Point (3.0 :: Double) (4.0 :: Double) :: Point Double
--           ^ type constructor ^
```

## PITFALL: Constructor Naming

```
-- NO
data Person = String Int
--      ^      ^
--      |      Created a value constructor named
--      |      ``String''. Possible because it is
--      |      currently available in the value
--      |      namespace (although not in the type
--      |      namespace). Probably not what was intended.
--      |
--      Created a nullary type constructor ``Person''

-- ok
data Person = Person String Int
```

## Namespaces and Constructors (5/12)

Pattern matching is the primary way we access the data wrapped in value constructors.

```
data Person = Person String Int
showNameAndAge :: Person -> String
showNameAndAge (Person name age) =
    name ++ " age=" ++ show age
```

```
Prelude> showNameAndAge (Person "Fred" 2)
"Fred age=2"
```

## Namespace and Constructors (6/12)

Next example:

```
-- Cats have a certain number of lives
data Creature = Dog String | Cat String Int
showCreature :: Creature -> String
showCreature c = case c of
```

```

(Dog name)      -> name
(Cat name lives) -> name ++ " lives=" ++ show lives

Prelude> showCreature (Dog "Fido")
"Fido"
Prelude> showCreature (Cat "Mittens" 9)
"Mittens lives-left=9"

```

## Namespaces and Constructors (7/12)

Another pattern matching example for constructed values:

```

-- Function from the test-taking belt of New Jersey
data Grade = A | B | C | D | F
data TestResult = Complete Grade | Missed
showTestResult :: TestResult -> String
showTestResult Missed = "The test was missed."
showTestResult (Complete g) = case g of
    A      -> "Top score!"
    B      -> "Horrible score, but you'll live!"
    otherwise -> "Why even bother?"

Prelude> showTestResult (Complete A)
"Top score!"

```

## Namespaces and Constructors (8/12)

Here is an example of a recursive type definition:

```

data List a = Nil | Cons a (List a)

```

- `List` is a unary type constructor, meaning it takes one type argument. This can be anything from `Int`, to `Char`, to `[Char]`, etc.

```
List :: * -> *
```
- `Nil` is a nullary value constructor, taking no value arguments.

```
Nil :: List a
```
- `Cons` is a binary value constructor, taking two value arguments.

```
Cons :: a -> List a -> List a
```

## Namespaces and Constructors (9/12)

Here is an example of a recursive type definition:

```
data List a = Nil | Cons a (List a)
```

### Kinds

`List` can be viewed as an even higher-order function with the *kind* of `* -> *`, in the same fashion as `Int -> Int`.

*Kinds are higher than types.*

`*` represents a type argument, and in this case, the *type variable* `a`, which contains a type.

## Namespaces and Constructors (10/12)

You can ask GHCi to print out some kinds:

```
-- Int is a nullary type constructor
Prelude> :k Int
Int :: *

-- Int is a unary type constructor
Prelude> :k []
[] :: * -> *

-- We feed [] :: (* -> *) an Int :: *
Prelude> :k [Int]
[Int] :: *
```

## PITFALL: Value Constructor Argument Mismatch

Value constructors only understand nullary type constructors (of kind `*`). Watch out for argument mismatches. In this case, the RHS `List` does not get passed the type argument `a`. Instead, `List` gets passed to `Cons` as its second argument and `a` gets passed as its third:

```
-- NO
data List a = Nil | Cons a List a
```



```
--          ^ ^ ^
--          1 2  3
```

Instead, wrap the type constructor and its type argument in parentheses.

```
-- ok
data List a = Nil | Cons a (List a)
--          ^ ^
--          1 2
```

## Namespaces and Constructors (11/12)

Let's re-examine our recursive type definition:

```
data List a = Nil | Cons a (List a)
```

Value constructors explained (again):

- Nil is a nullary value constructor: it takes no values to make a new constructed value. This is just like a previous example of three of them:

```
data = Employee | Manager | Client
```

- Cons is a binary value constructor. It takes a first argument of type `a` and a second argument of type `List a`. This is okay because `List` is of kind `*`  $\rightarrow$  `*`, so when it is passed the type variable `a` in the expression `List a`, the resulting kind is `*`, just like `Int`.

## Namespaces and Constructors (12/12)

Our List in action:

```
(Cons 5 Nil) :: Num a => List a
(Cons 5 (Cons 4 Nil)) :: Num a => List a
(Cons 5 (Cons 4 (Cons 3 Nil))) :: Num a => List a

car :: List a -> a
car (Cons x _) = x
car Nil = error "car called on Nil List"

-- Compare to the GHC source code:
-- | Extract the first element of a list...
head :: [a] -> a
head (x:_) = x
head [] = badHead
```

## Records (1/2)

Records are a way to generate accessor functions for a constructed value.

Without records:

```
data Person = Person String Int deriving Show
Prelude> :t Person "Fred" 5
Person "Fred" 5 :: Person
Prelude> let (Person _ age) = (Person "Fred" 5) in age
5
```

With records:

```
data Person = Person { name :: String, age :: Int } deriving Show
Prelude> Person {name="Fred", age=5}
Person {name = "Fred", age = 5}
Prelude> age Person {name="Fred", age=5}
5
```

## Records (2/2)

Trouble in paradise. There is a serious namespace issue:

```
data Person = Person { name :: String, age :: Int }
data Company = Company { name :: String, revenue :: Int }
```

When we compile:

```
$ ghc -Wall ./test.hs
[1 of 1] Compiling Main                ( test.hs, test.o )
```

```
test.hs:93:26: error:
    Multiple declarations of ‘name’
    Declared at: test.hs:92:24
                 test.hs:93:26
```

Google “haskell record problem” to learn more about the workarounds.

## Typeclasses Revisited (1/6)

- Typeclasses were created to express “ad-hoc polymorphism,” AKA, overloaded functions.
- They turned out to be nicer in many ways than those in Java or C++.
- I stole some great examples from Philip Wadler’s talk at Microsoft. He implemented typeclasses for Haskell and generics for Java.

## Typeclasses Revisited (2/6)

We want a generic function `max` so that we don't have to define it over and over again for different types. Let's write it naively in C++ using templates.

```
#include <iostream>

template<class T>
T max(T x, T y) {
    return x < y ? y : x;
}
```

## Typeclasses Revisited (3/6)

```
#include <iostream>

template<class T>
T max(T x, T y) {
    return x < y ? y : x;
}

void main() {
    // 2
    std::cout << max<int>(1, 2) << std::endl; // 2
    // 'b' (on most machines)
    std::cout << max<char>('a', 'b')) << std::endl;
    const char* a = "zzz"; const char* b = "aaa";
    // ???
    std::cout << max<const char*>(a, b)) << std::endl;
    // It's "aaa"
}
```

## Typeclasses Revisited (4/6)

Instead of C++ templates, let's try it using Haskell typeclasses.

```
class Ord a where
    (<) :: a -> a -> Bool

instance Ord Int where
    (<) = primitiveLessInt

instance Ord Char where
```

```

(<) = primitiveLessChar

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

```

## Typeclasses Revisited (5/6)

List of Char? No problem. Just add:

```

instance Ord String where
  [] < []      = False
  [] < blst    = True
  alst < []    = False
  (a:as) < (b:bs)
    | a < b    = True
    | b < a    = False
    | otherwise = as < bs

```

## Typeclasses Revisited (6/6)

But we can go deeper. String is just [Char]. Char already implements Ord. What if we abstracted this?

```

-- Instead of this tautology
Ord Char => [Char]
-- Let's say
Ord a => [a]

instance Ord a => Ord [a] where
  [] < []      = False
  [] < y:ys    = True
  x:xs < []    = False
  x:xs < y:ys | x < y    = True
               | y < x    = False
               | otherwise = xs < ys

```

Now we can handle comparisons with any level of list nesting.

## The Maybe Type

```

data Maybe a = Just a | Nothing

Just (5 :: Int) :: Maybe Int

```

```

Nothing :: Maybe a

addMaybe :: Int -> Maybe Int -> Int
addMaybe (x (Just y)) = x + y
addMaybe (x Nothing) = x

addMaybe 5 (Just 3)  -- 8
addMaybe 5 (Nothing) -- 5

```

## Functors Introduction (1/2)

Functors are essentially types that define `fmap`, as follows:

```

class Functor f where
    fmap :: (a -> b) -> f a -> f b

```

`fmap` takes a function that maps from `a` to `b`. It lifts the function so that it can operate on the *argument* of a value constructor `f`. That is, `fmap` gives us a function that can take `f a` and return `f b`!

## Functors Introduction (2/2)

Additionally, functors must also satisfy some rules known as the functor laws. Namely,

```

-- functor identity law
fmap id = id

-- functor composition law
fmap (f . g) = fmap f . fmap g

```

We don't have time to go over these in detail, but the motivation behind supporting these laws is that they're necessary to reduce complexity by defining specific but still "small" behavior.

## The Maybe Functor

The `Maybe` functor is defined as follows:

```

instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing

```

Usage example:

```

Prelude> fmap (+1) $ Just 5
Just 6
Prelude> fmap (+1) $ Nothing
Nothing
Prelude> fmap (\x -> x^x) $ Just 10
Just 10000000000
Prelude> fmap (\x -> x^x) $ Nothing
Nothing

```

## Interview Questions: Select Permutations (1/2)

```

-- Given a string and list of case-switching characters,
-- return all versions of the string that can be formed
-- by toggling the case-switching characters.
-- Inputs will be lowercase.
--
-- "test", ['s']      -> ["test", "teSt"]
-- "test", ['e', 's'] -> ["test", "teSt", "tEst", "tEst", ""]
import Data.Char (toLower, toUpper)

```

```

data Token = Normal Char | Switching Char deriving Show

```

```

tokenize :: String -> [Char] -> [Token]
tokenize str lst = map toToken str
  where toToken x = if elem x lst
                    then Switching x
                    else Normal x

```

## Interview Questions: Select Permutations (2/2)

```

generate :: String -> [Char] -> [String]
generate s switchList = generate' reversedTokens []
  where reversedTokens = reverse $ tokenize s switchList

generate' :: [Token] -> [Char] -> [String]
generate' [] current = [current]
generate' (r:rs) current = case r of
  Switching (ch) -> (generate' rs $ (toLower ch):current)
    ++ (generate' rs $ (toUpper ch):current)
  Normal (ch) -> (generate' rs $ ch:current)

main :: IO ()

```

```

main = do
  putStrLn $ show $ generate "test" ['s', 'e', 't']
-- Output:
-- ["test", "Test", "tEst", "TEst", "teSt", "TeSt", "tEST",
--  "TEST", "tesT", "TesT", "tEsT", "TEsT", "teST", "TeST",
--  "tEST", "TEST"]

```

## Interview Questions: Check Leading Ones (1/3)

```

-- Write a program that takes a byte array message
-- and integer count as inputs, and returns true if
-- there are at least that many leading ones in the
-- message, false otherwise.

-- Messages beginning with the two bytes 1 1 1 1 1 1 1 1
-- / 1 0 0 0 0 0 0 0 have nine leading ones.
--
-- Messages beginning with the byte 1 1 1 0 1 0 0 0
-- have three leading ones.

import Data.List
import Data.Bits
import Data.Word
import qualified Data.ByteString as BS

```

## Interview Questions: Check Leading Ones (2/3)

```

-- Create a list: [(255,8), (254,7), (252,6), (248,5), ...]
byteAList :: [(Word8, Int)]
byteAList = map makeMaskPair [0..7]
  where makeMaskPair x = ((shift 0xFF x) .&. 0xFF, 8-x)

-- Count how many leading ones the Word8 b has
countByteLeadingOnes :: Word8 -> Int
countByteLeadingOnes b = case find match byteAList of
  Nothing -> 0
  Just (_, y) -> y
  where match (x, _) = x .&. b == x

```

## Interview Questions: Check Leading Ones (3/3)

```
-- Determines if the bytestring bs has
-- at least n leading ones
hasLeadingOnes :: Int -> BS.ByteString -> Bool
hasLeadingOnes n bs = hasLeadingOnes' n bs 0

hasLeadingOnes' :: Int -> BS.ByteString -> Int -> Bool
hasLeadingOnes' n bs count = case BS.uncons bs of
    Nothing -> count >= n
    Just (b, bs') -> hasLeadingOnes' n bs' newCount
        where currentCount = countByteLeadingOnes b
              newCount = currentCount + 1
```

## Where to go from here

- Haskell Book by Chris Allen <http://haskellbook.com/>
- Learn You a Haskell by Miran Lipovača, <http://learnyouahaskell.com>
- Gentle Introduction To Haskell, version 98 by Paul Hudak, John Peterson, and Joseph Fasel, <https://www.haskell.org/tutorial/>
- Real World Haskell by Bryan O'Sullivan, <http://book.realworldhaskell.org/>
- An emporium of resources on the Haskell wiki: [https://wiki.haskell.org/Learning\\_Haskell](https://wiki.haskell.org/Learning_Haskell)

## Video learning materials

- Functional Programming Fundamentals by Dr. Erik Meijer (Video Lectures) <https://channel9.msdn.com/Series/C9-Lectures-Erik-Meijer-Functional-Programming-Fundamentals/>
- Haskell Course Taught by Philip Wadler, <https://www.youtube.com/watch?v=AOl2y5uW0mA&list=PLtRG9GLtNcHBv4cuh2w1cz5VsgY6adoc3>
- Adventure with Types in Haskell by Simon Peyton Jones, <https://www.youtube.com/watch?v=6COvD8oynmI>
- Haskell is Not For Production and Other Tales by Katie Miller, <https://www.youtube.com/watch?v=mlTO510zO78>



## Resources to learn about monads (1/2)

There are so many monad tutorials online, so I'll only list a few that I felt really helped me. Your learning style might be different, so please check out many of them! However, here's some ground advice that could really help your venture:

- Know how to define typeclasses
- Be comfortable with type and value constructors and how they interact with functions and typeclasses
- Learn functors, applicative functors, and monoids first. LYAH will go over these beforehand.
- If you just need to *use* (as opposed to construct) a monad, you don't need to fully understand it.

## Resources to learn about monads (2/2)

- Read “What is a Monad?” asked by Peter Mortensen on StackOverflow for a variety of good answers, <https://stackoverflow.com/questions/44965/what-is-a-monad>
- Start with Learn You a Haskell by Miran Lipovača, <http://learnyouahaskell.com>
- My personal favorite explanation after somewhat grasping the concept was You Could Have Invented Monads! by sigfpe. The practice problems were immensely helpful in consolidating my knowledge, <http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html>
- There are so many monad tutorials that Haskell Wiki gave it a whole section called “Monad tutorials timeline” with dozens of them from over the years, [https://wiki.haskell.org/Monad\\_tutorials\\_timeline](https://wiki.haskell.org/Monad_tutorials_timeline)

## References (1/2)

These information sources were heavily leaned on for amazing examples, explanations, and more found in these slides.

- Learn You a Haskell, Miran Lipovača, <http://learnyouahaskell.com>
- Glasgow Haskell Compiler, <https://www.haskell.org/ghc/>
- “Haskell Type vs Data Constructor”, kqr on StackOverflow, <https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>

- “Haskell function composition (.) and function application (\$) idioms: correct use”, ellisbben on StackOverflow, <https://stackoverflow.com/questions/3030675/haskell-function-composition-and-function-application-idioms-correct-us>

## References (2/2)

(continued)

- “Faith, Evolution, and Programming Languages: from Haskell to Java”, Philip Wadler, <https://www.youtube.com/watch?v=NZeDRs6snm0>
- “How to Learn Haskell in Less Than 5 Years”, Chris Allen, <https://www.youtube.com/watch?v=Bg9ccYzMbxc>
- “Gentle Introduction To Haskell, version 98”, Paul Hudak, John Peterson, Joseph Fasel, <https://www.haskell.org/tutorial/>
- “Algebraic data type”, Haskell Wiki, [https://wiki.haskell.org/Algebraic\\_data\\_type](https://wiki.haskell.org/Algebraic_data_type)
- ECMAScript Pattern Matching Proposal, <https://github.com/tc39/proposal-pattern-matching>

**EOF**