



CPSC 312

Functional and Logic Programming

October 27, 2015



Words of wisdom

"A language that doesn't affect the way you think about programming is not worth knowing."

Alan Perlis

GHC: Glasgow Haskell Compiler

Download either the full platform from: <https://www.haskell.org/platform/>

Or the compiler alone: <https://www.haskell.org/ghc/>

What about Text Editor you ask?

You may find that Eclipse has a plugin called [EclipseFP](#) that works beautifully with Haskell code; but don't be fooled by its shiny syntax highlighting. When it works, it's amazing indeed, but when it stops working (which, oh, yes, it will), it's a hell no one would get you out of. USE AT YOUR OWN RISK.

My two cents: stick with Sublime. Follow these instructions to install SublimeHaskell: <https://github.com/SublimeHaskell/SublimeHaskell>

and be content!

What is functional programming?

In the functional programming (FP) paradigm, computation is the evaluation of mathematical functions. (Note: math does not imply numbers only.)

What is functional programming?

In the functional programming (FP) paradigm, computation is the evaluation of mathematical functions. (Note: math does not imply numbers only.)

A function defines a correspondence between elements of an input *domain* and elements of an output *range*. (Remember those words from high school math?)

What is functional programming?

Here are some defining characteristics of FP:

What is functional programming?

Here are some defining characteristics of FP:

- FP is stateless -- or more accurately, any state that a functional program has does not change

What is state?

"*State* is any data the program stores that can possibly be changed by more than one piece of code. It is dangerous because if the code's behavior is dependent on a piece of state, it is impossible to analyze what it might do without taking into account all the possible values of that state, as well as every other part of the program that might modify that state. This problem is exponentially magnified in parallel programs, where it is not always easy to tell even what order code will execute in. It becomes nearly impossible to predict what a given state might be."

Practical Clojure, VanderHart and Sierra, p. 5. Springer-Verlag (Apress), 2010.

What is functional programming?

Here are some defining characteristics of FP:

- FP is stateless -- or more accurately, any state that a functional program has does not change
- Consequently, there are no side effects. This means that nothing that a function does will persist after the function evaluation has ended. Nothing a function does changes the state.

What is a side effect?

"*Side effects* are anything a function does when it is executed, besides just returning a value. If it changes program state, writes to a hard disk, or performs any kind of IO, it has executed a side effect. Of course, side effects are necessary. But they also make a function much more difficult to understand and to reuse in different contexts."

Practical Clojure, VanderHart and Sierra, p. 5. Springer-Verlag (Apress), 2010.

[Note that in some circles, a function that generates side effects isn't called a function. It's a procedure. In those circles, the name 'function' is reserved only for procedures with no side effects.]

What's the big deal with side effects?

Why do we care about

“...side effects are necessary. But they also make a function much more difficult to understand and to reuse in different contexts.”

or

“It is dangerous because if the code's behavior is dependent on a piece of state, it is impossible to analyze what it might do without taking into account all the possible values of that state, as well as every other part of the program that might modify that state.”

What's the big deal with side effects?

Why do we care about

“...side effects are necessary. But they also make a function much more difficult to understand and to reuse in different contexts.”

or

“It is dangerous because if the code's behavior is dependent on a piece of state, it is impossible to analyze what it might do without taking into account all the possible values of that state, as well as every other part of the program that might modify that state.”

It's not a big deal with the little programs you write for your CS classes. But your future employer isn't going to hire you to write little programs...

What is functional programming?

We care about anything that will help us manage the complexity of real (i.e., big) programs. Reducing the reliance on side effects may be one of those things.

Now back to “What is functional programming?”...

What is functional programming?

Here are some defining characteristics of FP:

- FP is stateless -- or more accurately, any state that a functional program has does not change
- Consequently, there are no side effects. This means that nothing that a function does will persist after the function evaluation has ended. Nothing a function does changes the state.
- Every function call with the same parameters returns the same result. Every time. This is a result of a mathematical and functional programming principle called referential transparency.

Compare to imperative programming

In the imperative or procedural programming paradigm (C, C++, Java), computation is about state and sequences of instructions that change the state.

In functional programming, there is no sense of global program state. Programs are not sequences of instructions in the functional programming world.

In functional programming, a function is a single expression, which is executed by evaluating the expression. A program is a composition of functions (i.e., one function calls another, which calls another....).

Referential transparency*

In the world of mathematics, any (sub)expression can always be replaced by its value without changing the outcome of the overall computation.

For example, in: $(2ax + b)(2ax + c)$

We'd never evaluate $2ax$ twice.

If we determine once that $2ax$ is 12 (e.g., $a = 3$ and $x = 2$), it's not going to change if we evaluate $2ax$ a second time.

This is called referential transparency: any reference to a (sub)expression can be replaced by its value regardless of the surrounding context.

Math functions are always referentially transparent.

Referential transparency

Because of referential transparency, the result of evaluating a mathematical function depends only on the values passed as arguments (or actual parameters) to the function. The interface to the function -- the connection between the function and its surrounding context -- is visually obvious. There is no hidden interface.

This allows us to derive new expressions from given expressions, transform expressions into more useful forms, and prove things about expressions.

Referential transparency

While math functions are referentially transparent, procedures in programming languages aren't necessarily so. The result of a procedure can depend on non-local variables or local state variables, and successive calls with the same arguments can produce different results. For example...

If you can remember your math classes, you know that math would be fairly unpredictable and incomprehensible if you couldn't count on functions returning the same value consistently given the same inputs:

$$2 + 2 = 4$$

Referential transparency

If you can remember your math classes, you know that math would be fairly unpredictable and incomprehensible if you couldn't count on functions returning the same value consistently given the same inputs:

$$2 + 2 = 4$$

$$2 + 2 = 3.14159$$

Referential transparency

If you can remember your math classes, you know that math would be fairly unpredictable and incomprehensible if you couldn't count on functions returning the same value consistently given the same inputs:

$$2 + 2 = 4$$

$$2 + 2 = 3.14159$$

$$2 + 2 = -37$$

Referential transparency

If you can remember your math classes, you know that math would be fairly unpredictable and incomprehensible if you couldn't count on functions returning the same value consistently given the same inputs:

$$2 + 2 = 4$$

$$2 + 2 = 3.14159$$

$$2 + 2 = -37$$

This would be disturbing, to say the least.

Referential transparency

If you can remember your math classes, you know that math would be fairly unpredictable and incomprehensible if you couldn't count on functions returning the same value consistently given the same inputs:

$$2 + 2 = 4$$

$$2 + 2 = 3.14159$$

$$2 + 2 = -37$$

This would be disturbing, to say the least. (Just ask Glenn.)

What is $2 + 2$?



Referential transparency

If you can remember your math classes, you know that math would be fairly unpredictable and incomprehensible if you couldn't count on functions returning the same value consistently given the same inputs:

$$2 + 2 = 4$$

$$2 + 2 = 3.14159$$

$$2 + 2 = -37$$

You'd have failed all your math exams!

Yet, in the imperative programming paradigm that we're so familiar with, this sort of behaviour is commonplace and even encouraged (e.g., the static variable in Java)...

Referential transparency

```
public class StateDemo
{
    public static double foovar = 0;

    public static void main(String[] args)
    {
        System.out.println(foo(1)/foo(1));
        System.out.println(foo(1)/foo(1));
        System.out.println(foo(1)/foo(1));
    }

    public static double foo(int x)
    {
        foovar++;
        return x + foovar;
    }
}
```

Referential transparency

```
public class StateDemo
{
    public static double foovar = 0;

    public static void main(String[] args)
    {
        System.out.println(foo(1)/foo(1));
        System.out.println(foo(1)/foo(1));
        System.out.println(foo(1)/foo(1));
    }

    public static double foo(int x)
    {
        foovar++;
        return x + foovar;
    }
}
```

In a sane world, the output would be

1
1
1

(unless `foo(1)` returns 0, of course)

Referential transparency

```
public class StateDemo
{
    public static double foovar = 0;

    public static void main(String[] args)
    {
        System.out.println(foo(1)/foo(1));
        System.out.println(foo(1)/foo(1));
        System.out.println(foo(1)/foo(1));
    }

    public static double foo(int x)
    {
        foovar++;
        return x + foovar;
    }
}
```

In the imperative world of Java, the output is

```
0.6666666666666666
0.8
0.8571428571428571
```

Referential transparency

In the previous example, the value returned by a procedure depends not only on the values passed to it as parameters, but also on a not-so-obvious static or non-local variable.

What the procedure returns is governed not just by the parameters being passed, but when the procedure is called and what's going on elsewhere in the program at that time!

Referential transparency

In the previous example, the value returned by a procedure depends not only on the values passed to it as parameters, but also on a not-so-obvious static or non-local variable.

What the procedure returns is governed not just by the parameters being passed, but when the procedure is called and what's going on elsewhere in the program at that time!

By contrast, referential transparency means that the result returned by a procedure is dependent only upon the values passed to the procedure as parameters. You never see a procedure whose result depends not only on explicit parameters but also "hidden" influences like the value of some variable which may change between one procedure call and the next.

Referential transparency

It's not difficult to see that referentially-transparent programs are easier to work with (e.g., make correct, debug, prove correct) than those that are referentially-opaque.

Referential transparency

It's not difficult to see that referentially-transparent programs are easier to work with (e.g., make correct, debug, prove correct) than those that are referentially-opaque.

The bad news is that to make referentially-transparent programs, you have to learn how to write programs that don't rely on side effects. And that means you'll have to give up your beloved assignment statements that you grew addicted to in CPSC 210, CPSC 221, EECE 210, or EECE 309.

Referential transparency

Giving up assignment statements might not be easy for you. Assignment statements are essential to imperative and OO programming. Recall that those approaches are based on the von Neumann computing architecture, which relies on moving things from one memory location to another. That's what assignment statements do. If you give up assignment, you can't move things in and out of memory locations. How can you write programs if you can't do assignment?

In ten words or less, that's functional programming: you give up side effects to gain referential transparency.

Referential transparency

One more thing about referential transparency: when your expressions, procedures, or functions are referentially transparent and there are no side effects, order of evaluation doesn't matter. Consequently, a functional program can be decomposed relatively easily into independent subcomponents to be evaluated in parallel on multiple processors. “you can have concurrency or you can have side effects, but you can't have both.”

Imperative programs, however, have lots of dependencies by design. Converting an imperative program into subcomponents to be evaluated independently is quite a chore.

How will we survive?

Aren't side effects necessary? Can we really write all programs *without* assignment statements?

How will we survive?

Aren't side effects necessary? Can we really write all programs *without* assignment statements?

The earliest FP language is a theoretical entity called lambda calculus -- a formal system for talking about computation with simple functions and without state. It was put forward by Alonzo Church in the 1930s, who proved that anything computable could be computed with lambda calculus.

Lambda calculus is the evolutionary ancestor of current functional programming languages like Scheme, Lisp, Haskell, ML, and Miranda.

How will we survive?

Aren't side effects necessary? Can we really write all programs *without* assignment statements?

If Alonzo says that anything computable can be computed with lambda calculus, then it can be computed with a functional programming language.

How will we survive?

Aren't side effects necessary? Can we really write all programs *without* assignment statements?

If Alonzo says that anything computable can be computed with lambda calculus, then it can be computed with a functional programming language.

Alonzo says you'll be ok. Relax.

How will we survive?

Aren't side effects necessary? Can we really write all programs *without* assignment statements?

If Alonzo says that anything computable can be computed with lambda calculus, then it can be computed with a functional programming language.

Alonzo says you'll be ok. Relax.

Do you need to understand lambda calculus to be a successful functional programmer? No. And as far as I can tell, none of my teaching predecessors in CPSC 312 have tried to teach lambda calculus in here, and we'll continue in that tradition.

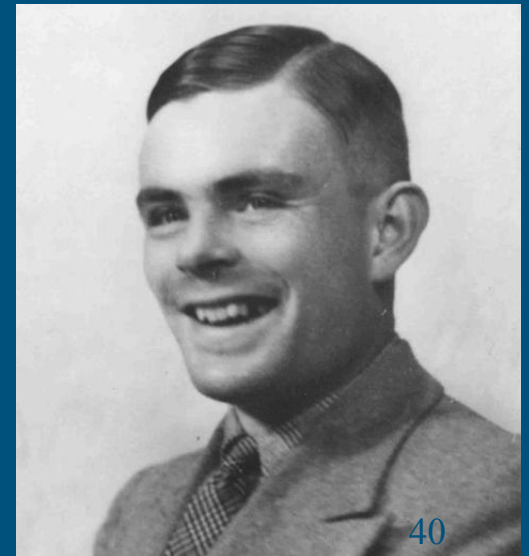
How will we survive?

But if lambda calculus can compute anything that's computable, why didn't we all grow up in a world of functional computing based on Alonzo Church's ideas?



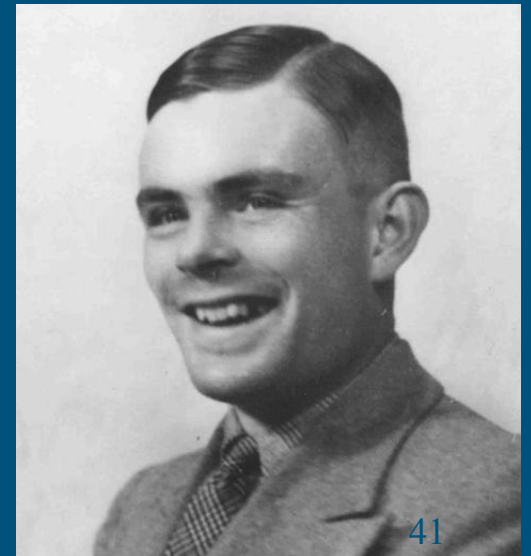
How will we survive?

But if lambda calculus can compute anything that's computable, why didn't we all grow up in a world of functional computing based on Alonzo Church's ideas? Because his work was later overshadowed by his grad student's work.



How will we survive?

But if lambda calculus can compute anything that's computable, why didn't we all grow up in a world of functional computing based on Alonzo Church's ideas? Because his work was later overshadowed by his grad student's work. Perhaps you've heard of Alan Turing and his Turing Machine?



Computational model face-off

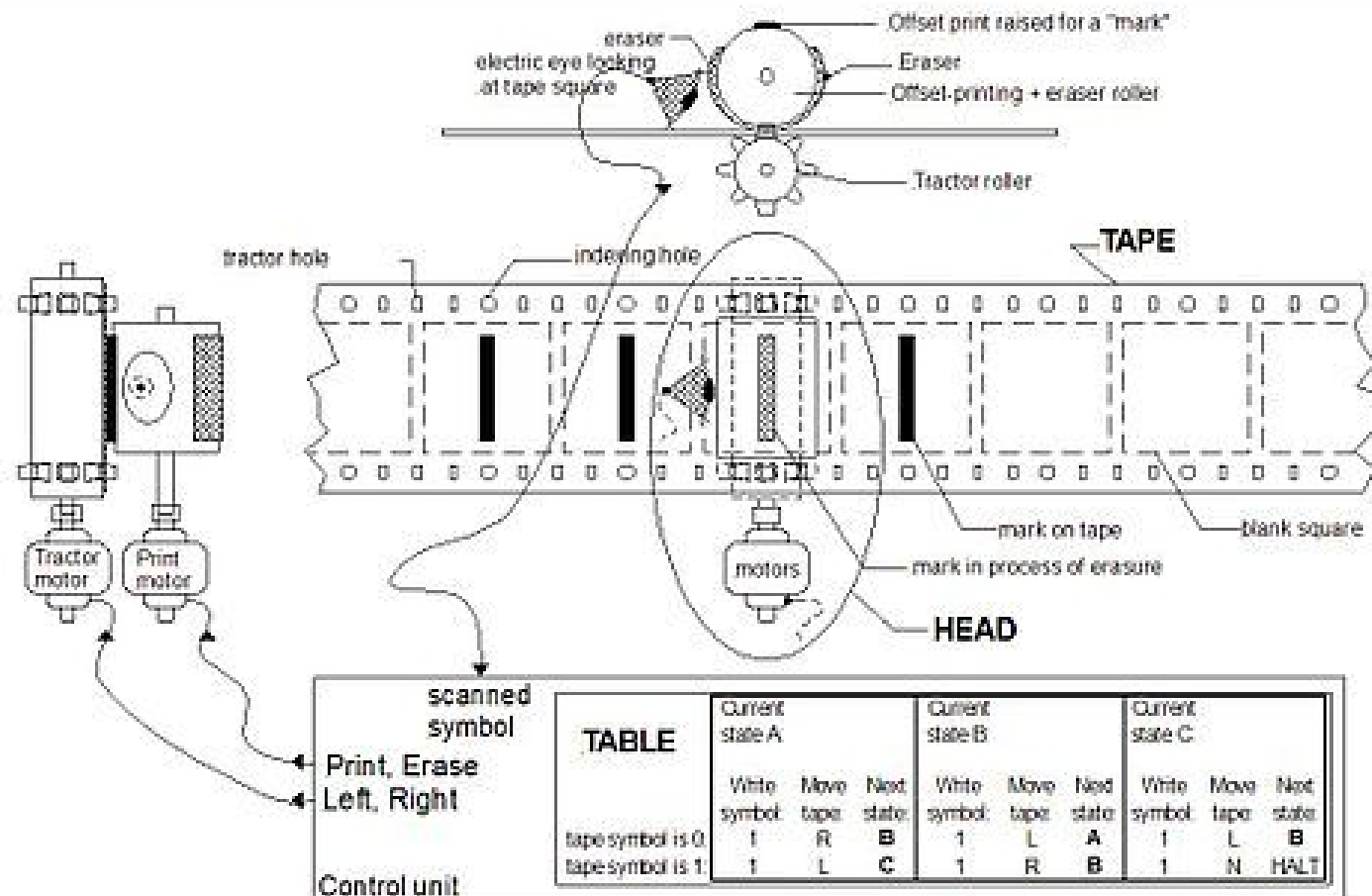
Without going into any detail, let it suffice to say that lambda calculus is a formal system for expressing computation with anonymous (nameless) functions with only single arguments.

Here's a taste of what lambda calculus looks like:

$$Y := \lambda g. (\lambda x. g (x x)) (\lambda x. g (x x))$$

This particular expression is called the Y-combinator. It's how you get recursion when you don't have function names.

Computational model face-off



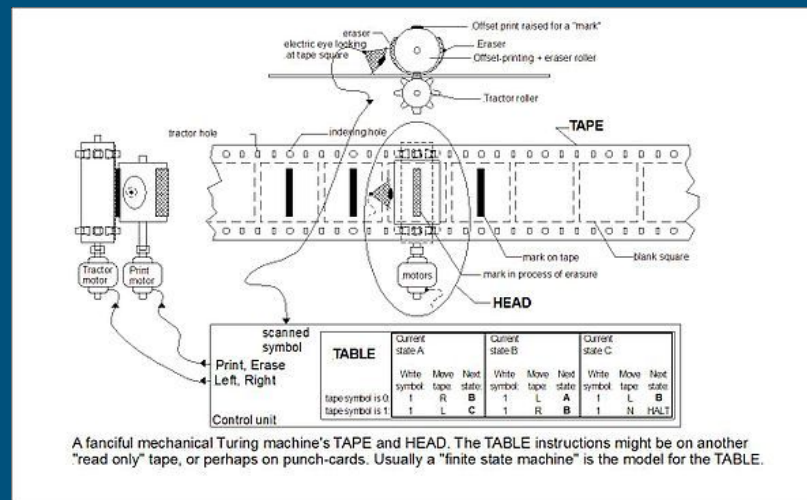
A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

Computational model face-off

So let's say you're an engineer, approaching the task of building one of the first electronic computers. Which of two computationally-equivalent models are you going to try to implement?

This? $Y := \lambda g.(\lambda x.g (x x)) (\lambda x.g (x x))$

or this?



Computational model face-off

Did anyone build a lambda calculus machine? In software, yes. In the late 1950s, John McCarthy based the LISP programming language on lambda calculus, and the rest, as they say, is programming language history.



Beyond referential transparency

Functional programming isn't just about referential transparency or the potential for programming in a world of concurrency. FP tends to look at a solution to a problem at a higher, more abstract level than imperative programming. It's more about describing what to do to solve the problem and much less about the low-level details of how the computation gets done.

This in turn can lead to functional programs that are more concise and more clear than their imperative counterparts. For example, consider the algorithm for Quicksort:

Beyond referential transparency

Quicksort*:

The steps are:

1. Pick an element, called a **pivot**, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

And here's quicksort implemented in Java:

Beyond referential transparency

```
int partition(int arr[], int left, int right)
{
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    };

    return i;
}
```

```
void quickSort(int arr[], int left, int right) {
    int index = partition(arr, left, right);
    if (left < index - 1)
        quickSort(arr, left, index - 1);
    if (index < right)
        quickSort(arr, index, right);
}
```


Beyond referential transparency

...and here's quicksort in Haskell

Beyond referential transparency

...and here's quicksort in Haskell

Are you ready?

Beyond referential transparency

...and here's quicksort in Haskell

```
qsort :: Ord a => [a] -> [a]
```

```
qsort [] = []
```

```
qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort greater)
```

```
  where
```

```
    lesser = filter (< p) xs
```

```
    greater = filter (>= p) xs
```

Beyond referential transparency

...and here's quicksort in Haskell

```
qsort :: Ord a => [a] -> [a]
```

```
qsort [] = []
```

```
qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort greater)
```

```
  where
```

```
    lesser = filter (< p) xs
```

```
    greater = filter (>= p) xs
```

You can already tell that the Haskell version is more concise. In several weeks, you'll say that the Haskell version is more understandable too.

Beyond referential transparency

```
void qsort(int a[], int lo, int hi) {
{
    int h, l, p, t;

    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];
        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        } while (l < h);

        t = a[l];
        a[l] = a[hi];
        a[hi] = t;

        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}
```

```
qsort :: Ord a => [a] -> [a]
```

```
qsort [] = []
qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort
greater)
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

See all those assignment statements to the left? That's what we mean when we say that the imperative programming paradigm is all about following commands to move data among memory locations.

Compare to the Haskell code above: No memory locations, no assignment.

Beyond referential transparency

But (and this is a big but) functional programs can be less efficient in both time and space than their imperative counterparts. The Haskell quicksort runs slower and uses more memory than the C quicksort. The C quicksort program sorts the array in place and does no additional memory allocation, and it runs more quickly by comparison.

Beyond referential transparency

But (and this is a big but) functional programs can be less efficient in both time and space than their imperative counterparts. The Haskell quicksort runs slower and uses more memory than the C quicksort. The C quicksort program sorts the array in place and does no additional memory allocation, and it runs more quickly by comparison.

Sometimes performance is the big issue, so FP might not be the best way to go in those cases. But in many cases, programmer efficiency is more important than program efficiency, and that's when FP comes in real handy.

Questions?

The Haskell programming language

Haskell is a pure functional programming language named after noted logician Haskell Curry.



The Haskell programming language

Haskell is a pure functional programming language named after noted logician Haskell Curry.

Haskell is "pure" because it doesn't ever allow side effects. Scheme and Lisp, for example, are not pure because, although they're based on lambda calculus too, they allow you to go over to the dark side if you want to (i.e., they let you change state in powerful and dangerous ways).

The Haskell programming language

Haskell is a pure functional programming language named after noted logician Haskell Curry. We'll hear more about him later in the course.

Haskell is "pure" because it doesn't ever allow side effects. Scheme and Lisp, for example, are not pure because, although they're based on lambda calculus too, they allow you to go over to the dark side if you want to (i.e., they let you change state in powerful and dangerous ways).

Haskell is statically typed (data type declarations are required and checked at compile time) and strongly typed (operations won't succeed on operands of the wrong type -- it won't allow potentially harmful type conversions).

Anatomy of a Haskell function

`square (x) = x2`

in math is the equivalent of this in Haskell:

```
square :: Int -> Int
square n = n ^ 2
```

Anatomy of a Haskell function

This is the type declaration for the function

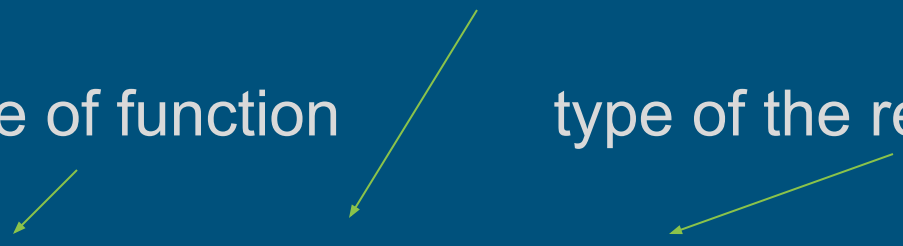
```
square :: Int -> Int  
square n = n ^ 2
```

Anatomy of a Haskell function

type of the formal parameter

name of function

type of the result to be returned



```
square :: Int -> Int  
square n = n ^ 2
```

Anatomy of a Haskell function

```
square :: Int -> Int  
square n = n ^ 2
```

This is the actual function definition

Anatomy of a Haskell function

```
square :: Int -> Int  
square n = n ^ 2
```

formal parameter

function name

result (function body defined
in terms of formal parameters)

Anatomy of a Haskell function

What's a function definition with no parameters?

Anatomy of a Haskell function

What's a function definition with no parameters?

A constant:

```
mypi :: Float  
mypi = 3.14159
```

Haskell scripts

A program in Haskell is called a script. A script consists of one or more function definitions along with associated comments.

Names of files containing Haskell scripts should end with either the `.hs` or `.lhs` suffix. Use the `.hs` prefix unless you write Haskell code in the "literate style". More about this later.

Interacting with GHCi

```
$ ghci
GHCi, version 7.10.2: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load test
[1 of 1] Compiling Main                ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main> square 5
25
*Main> square 1024
1048576
*Main> :quit
Leaving GHCi.
```

Basic data types

The Boolean data type in Haskell is called `Bool`

The Boolean values are `True` and `False`

The logical operators are

`&&` `and`

`||` `or`

`not` `not`

Basic data types

The integer data type in Haskell is called `Int`

The maximum value for the `Int` type is 2147483647

The arithmetic operators include

`+` addition

`-` subtraction

`*` multiplication

`/` division

`^` power

`div` whole number division (prefix)

`mod` remainder from whole number division
(prefix)

Basic data types

The integer data type in Haskell is called `Int`

The maximum value for the `Int` type is 2147483647

The relational operators include

- `>` greater than
- `>=` greater than or equal to
- `==` equal to
- `/=` not equal to
- `<=` less than or equal to
- `<` less than

Basic data types

The integer data type in Haskell is called `Int`

The maximum value for the `Int` type is 2147483647

Arbitrarily large integers need the `Integer` data type

Basic data types

There's also

Char character

Float floating point

Double floating point with more precision

Read the textbook for more.

Basic data types

There's also

`Char` `character`

`Float` `floating point`

`Double` `floating point with more precision`

Read the textbook for more

Yes, there's a `String` type too. Later.

Basic data types

There's no automatic conversion from `Int` to `Float` as there is in Java, for example. (That's a result of strong typing.) Use `fromIntegral` to convert from `Int` to `Float`.

```
Main> (floor 5.6) + 6.7
ERROR - Unresolved overloading
*** Type      : (Fractional a, Integral a) => a
*** Expression : floor 5.6 + 6.7
```

```
Main> fromIntegral(floor 5.6) + 6.7
11.7
```

Identifiers

Function names and variable (parameter) names begin with lower case letters. Type names (like `Int`) begin with upper case letters.

Identifiers

Function names and variable (parameter) names begin with lower case letters. Type names (like `Int`) begin with upper case letters.

The same identifier can be used to name a function and a variable. For example, this

```
foobar :: Int -> Int
foobar foobar = foobar * foobar
```

```
Main> foobar 3
9
```

works just fine. Never ever ever do this.

Haskell comments

```
-- precedes a one-line comment
```

```
{-    this is a block of  
  comments    -}
```

Haskell comments

You could also program in the "literate style" where comments are the norm and have no special indicators. Instead, executable lines of code are preceded by `>`

Here's a comment in literate style
and below is the executable code

```
> square :: Int -> Int
> square n = n ^ 2
```

Names of literate Haskell scripts end with the `.lhs` suffix, not the `.hs` suffix.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise            = z
```


Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                 = y
  | otherwise             = z
```

If $x \geq y$ and $x \geq z$

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z    = x
  | y >= z              = y
  | otherwise          = z
```

If $x \geq y$ and $x \geq z$ then substitute the expression x for the expression $\text{max3 } x \ y \ z$.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```

Else if $y \geq z$

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```

Else if $y \geq z$ then substitute the expression y for the expression $\text{max3 } x \ y \ z$.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise            = z
```

Else

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise            = z
```

Else substitute the expression `z` for the expression `max3 x y z`.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```

The `otherwise` is not required but...

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```

The `otherwise` is not required but...

good programming style demands that you make explicit what you intend when no boolean expression is True

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise            = z
```

The `otherwise` is not required but...

good programming style demands that you make explicit
what you intend when no boolean expression is `True`

Haskell blows up at run time when no boolean expression
is `True`...oops!

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
```

```
*Main> max3 1 2 3
```

```
*** Exception: tests.hs:(6,1)-(8,28): Non-exhaustive patterns
in function max3
```

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```

Why "guard"? The guard directs flow of control. Think of it as protecting the expression which follows from being evaluated unless specific conditions are met (i.e., the boolean expression between the | and the =).

Layout

A Haskell program, or script, is a series of definitions. When does one definition end and the next one begin?

Layout

A Haskell program, or script, is a series of definitions. When does one definition end and the next one begin?

It's based on indentation. A definition ends by the first piece of text which lies at the same indentation as or to the left of the start of the definition.

```
max2 :: Int -> Int -> Int
max2 x y
  | x >= y    = x
  | otherwise = y
```

Haskell knows this definition
has ended because

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z    = x
  | y >= z              = y
  | otherwise          = z
```

this line starts at the same indent
as the first line of the previous
function definition

Layout

A Haskell program, or script, is a series of definitions. When does one definition end and the next one begin?

It's based on indentation. A definition ends by the first piece of text which lies at the same indentation as or to the left of the start of the definition.

```
max2 :: Int -> Int -> Int
max2 x y
    | x >= y      = x
    | otherwise  = y
```

These lines are type declarations
and not function definitions.
They could be way up at the top
of the script.

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z      = x
    | y >= z                = y
    | otherwise            = z
```

Layout

A Haskell program, or script, is a series of definitions. When does one definition end and the next one begin?

It's based on indentation. A definition ends by the first piece of text which lies at the same indentation as or to the left of the start of the definition.

```
max2 x y
  | x >= y      = x
  | otherwise  = y
```

These lines are type declarations
and not function definitions.
They could be way up at the top
of the script.

```
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```