# CSCI 334:
# Principles of Programming Languages

## Lecture 11: ML and F#

Instructor: Dan Barowy

Williams

## Announcements

## Announcements

Midterm exam grades emailed

## Announcements

Midterm exam grades emailed

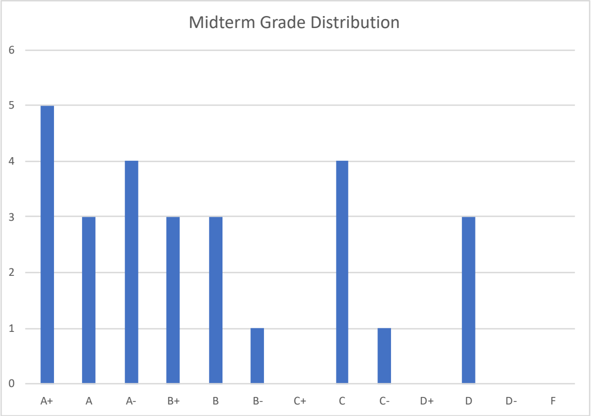
Need to meet with me 1 on 1 to get graded exam back

## Announcements

Midterm exam grades emailed

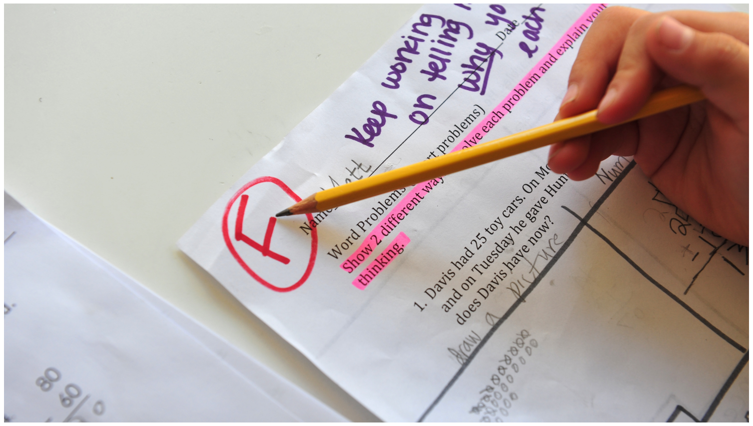
Need to meet with me 1 on 1 to get graded exam back

Exam grade distribuion

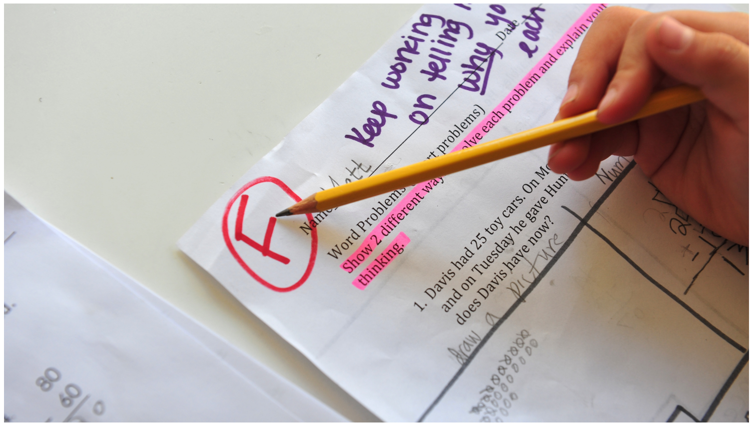


| | | | | | |
|---|---|---|---|---|---|
| average | 8.07407407 | 22.2962963 | 27.037037 | 28.1481481 | 85.5555556 |
| median | 8 | 26 | 27 | 30 | 88 |

## Why I dislike grades



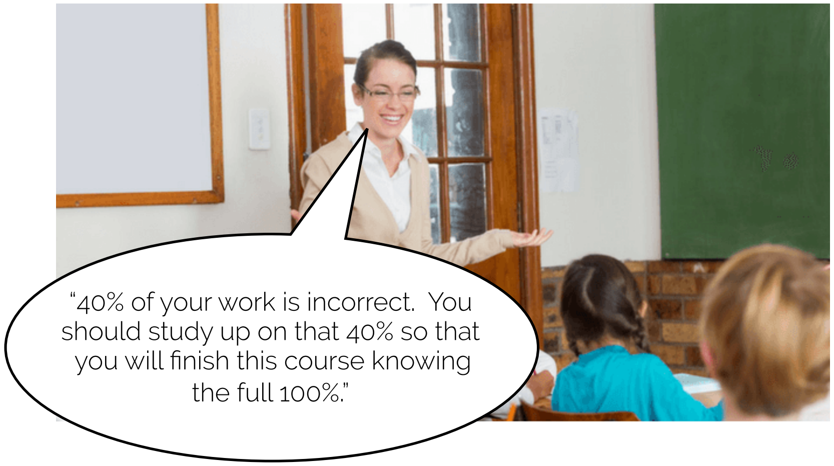
## Why I dislike grades



Let's say you get a 60% on your exam.

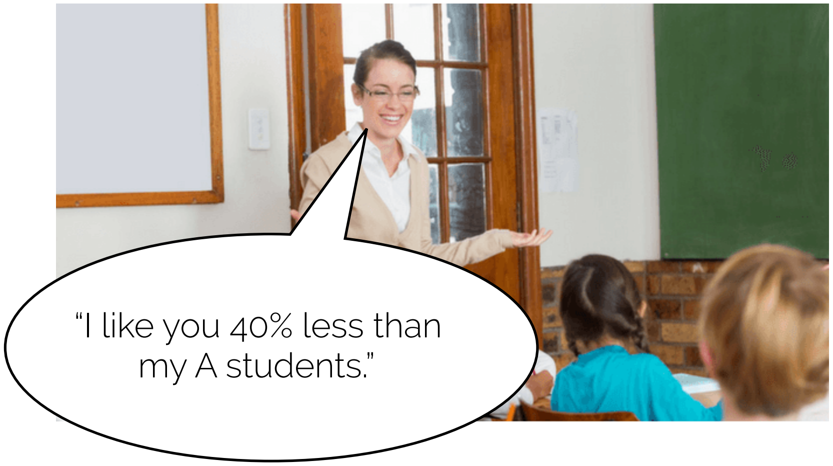
## Why I dislike grades

What **your grade means**.


"40% of your work is incorrect.  You should study up on that 40% so that you will finish this course knowing the full 100%."

## Why I dislike grades

What **your grade does not mean**.



## How you feel

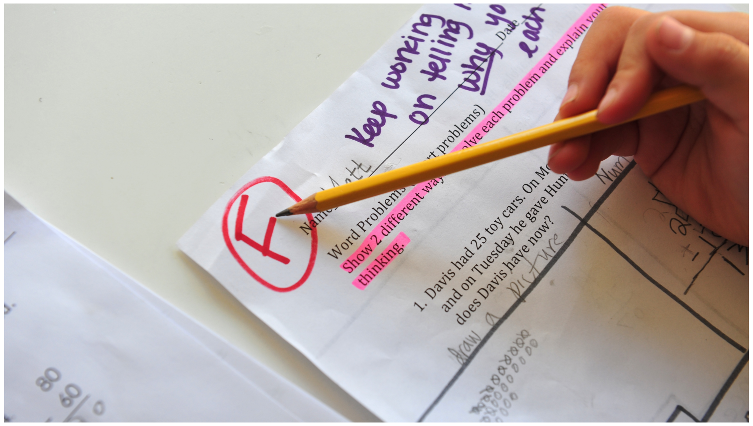
Surprised; Embarassed

## Why I dislike grades

Your grade has almost no bearing on whether I like you or not.
(It is sometimes even inversely correlated.)

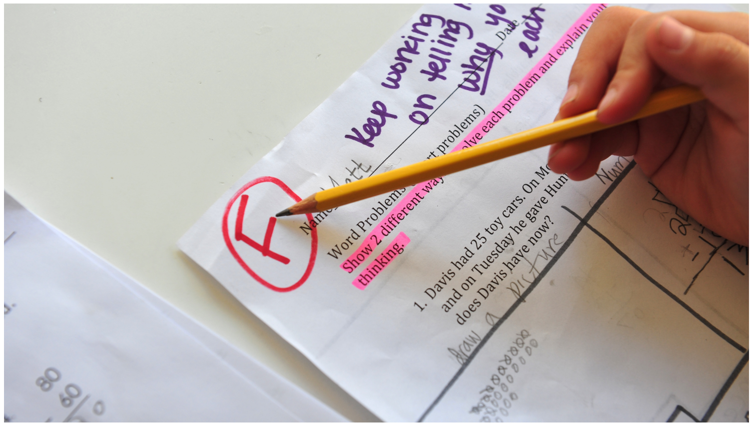
The same goes for most faculty.

## The purpose of a class


F
Keep working on telling why each
Word Problems
Show 2 different ways
thinking.
1. Davis had 25 toy cars. On Mo
and on Tuesday he gave Hun
does Davis have now?

To turn a weakness into a strength.

## The purpose of a class

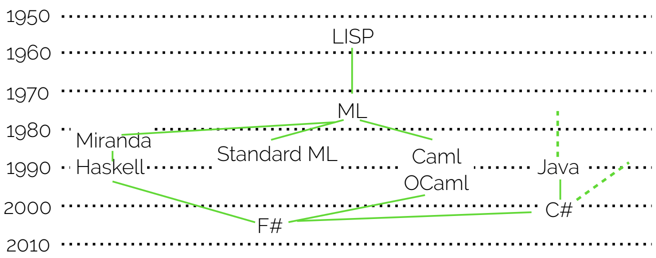

F
Keep working on telling why each
Word Problems
Show 2 different ways
thinking.
1. Davis had 25 toy cars. On Mo
and on Tuesday he gave Hun
does Davis have now?

A grade is just one way to identify a weakness.

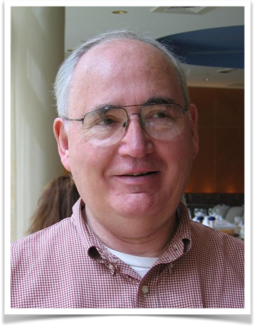
## Why I dislike grades

“It is our choices, Harry, that show what we truly are, far more than our abilities.”

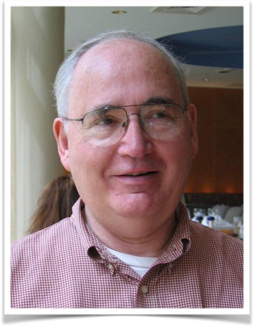
## ML



## ML

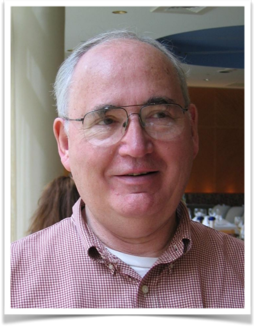


## ML



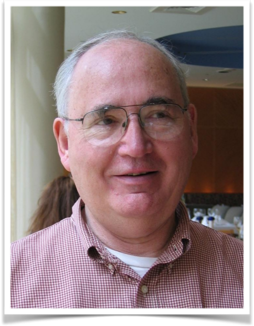
- Dana Scott

## ML



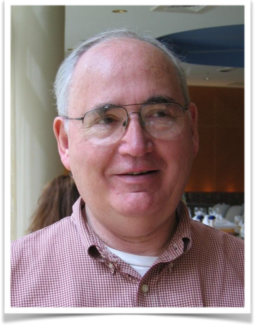
- Dana Scott
- Logic of Computable Functions (LCF)

## ML



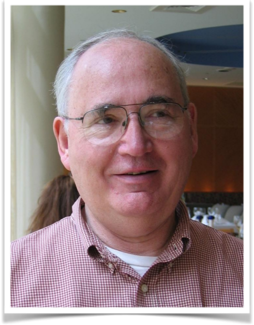
- Dana Scott
- Logic of Computable Functions (LCF)
  - Automated proofs!

## ML



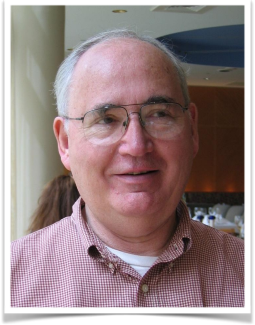
- Dana Scott
- Logic of Computable Functions (LCF)
  - Automated proofs!
  - Theorem proving is essentially a "search problem".

## ML



- Dana Scott
- Logic of Computable Functions (LCF)
  - Automated proofs!
  - Theorem proving is essentially a "search problem".
  - It is (essentially) NP-Complete

## ML



- Dana Scott
- Logic of Computable Functions (LCF)
  - Automated proofs!
  - Theorem proving is essentially a "search problem".
  - It is (essentially) NP-Complete
  - But works "in practice" with the right "tactics"

## ML

## ML

- Robin Milner

## ML

- Robin Milner
- How to program tactics?

## ML

- Robin Milner
- How to program tactics?
- A “meta-language” is needed

## ML

- Robin Milner
- How to program tactics?
- A “meta-language” is needed
- ML is born (1973)

## F#

## F#

- Don Syme

## F#

- Don Syme
- ML is “more fun” than Java or C#.

## F#

- Don Syme
- ML is “more fun” than Java or C#.
- Can we use ML instead?

# F#

- Don Syme
- ML is “more fun” than Java or C#.
- Can we use ML instead?
- F# is born (2010).

# ML Features: static types

# ML Features: static types

- Core: LISP + “static types”

# ML Features: static types

- Core: LISP + “static types”
  - types are checked *before program runs*

## ML Features: static types

- Core: LISP + “static types”
  - types are checked *before program runs*
- Static types guarantee correctness of programs

## ML Features: static types

- Core: LISP + “static types”
  - types are checked *before program runs*
- Static types guarantee correctness of programs
  - Why does this not violate halting problem?

## ML Features: static types

- Core: LISP + “static types”
  - types are checked *before program runs*
- Static types guarantee correctness of programs
  - Why does this not violate halting problem?
  - All “well-typed” programs do not fail at runtime

## ML Features: parametric polymorphism

## ML Features: parametric polymorphism

```
let swapInt(x: int, y: int): int*int = (y,x)
```

## ML Features: parametric polymorphism

```
let swapInt(x: int, y: int): int*int = (y,x)
let swapReal(x: real, y: real): real*real = (y,x)
```

## ML Features: parametric polymorphism

```
let swapInt(x: int, y: int): int*int = (y,x)
let swapReal(x: real, y: real): real*real = (y,x)
let swapString(x: string, y: string): string*string = (y,x)
```

## ML Features: parametric polymorphism

```
let swapInt(x: int, y: int): int*int = (y,x)
let swapReal(x: real, y: real): real*real = (y,x)
let swapString(x: string, y: string): string*string = (y,x)
```

- “abstract types” allow programmers to write generic programs; reveal underlying idea without boilerplate

# ML Features: parametric polymorphism

```
let swapInt(x: int, y: int): int*int = (y,x)
let swapReal(x: real, y: real): real*real = (y,x)
let swapString(x: string, y: string): string*string = (y,x)
```

- "abstract types" allow programmers to write generic programs; reveal underlying idea without boilerplate

```
let swap(x: 'a, y: 'b): 'b * 'a = (y,x)
```

# ML Features: type inference

# ML Features: type inference

```
let swap(x: 'a, y: 'b): 'b * 'a = (y,x)
```

# ML Features: type inference

```
let swap(x: 'a, y: 'b): 'b * 'a = (y,x)
```

- writing types is hard (and sometimes ugly!)

## ML Features: type inference

```
let swap(x: 'a, y: 'b): 'b * 'a = (y,x)
```

- writing types is hard (and sometimes ugly!)

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

## ML Features: exceptions

## ML Features: exceptions

- Milner: it's hard to write well-typed programs

## ML Features: exceptions

- Milner: it's hard to write well-typed programs
  - mechanism to allow programs to signal error

## ML Features: exceptions

- Milner: it's hard to write well-typed programs
  - mechanism to allow programs to signal error
  - *and correct for them at runtime*

## ML Features: exceptions

- Milner: it's hard to write well-typed programs
  - mechanism to allow programs to signal error
  - *and correct for them at runtime*

```
let foo() =
 exception DivByZero of string
 if x = 0 then raise DivByZero(“no zeros!”)
```

## ML Features: exceptions

- Milner: it's hard to write well-typed programs
  - mechanism to allow programs to signal error
  - *and correct for them at runtime*

```
let foo() =
 exception DivByZero of string
 if x = 0 then raise DivByZero(“no zeros!”)

…
```

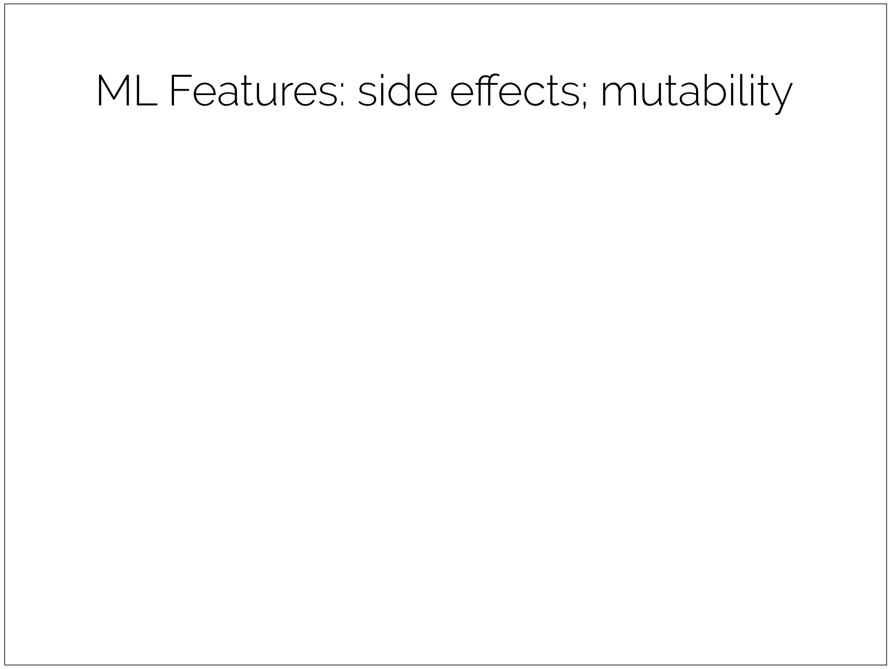
## ML Features: exceptions

- Milner: it's hard to write well-typed programs
  - mechanism to allow programs to signal error
  - *and correct for them at runtime*

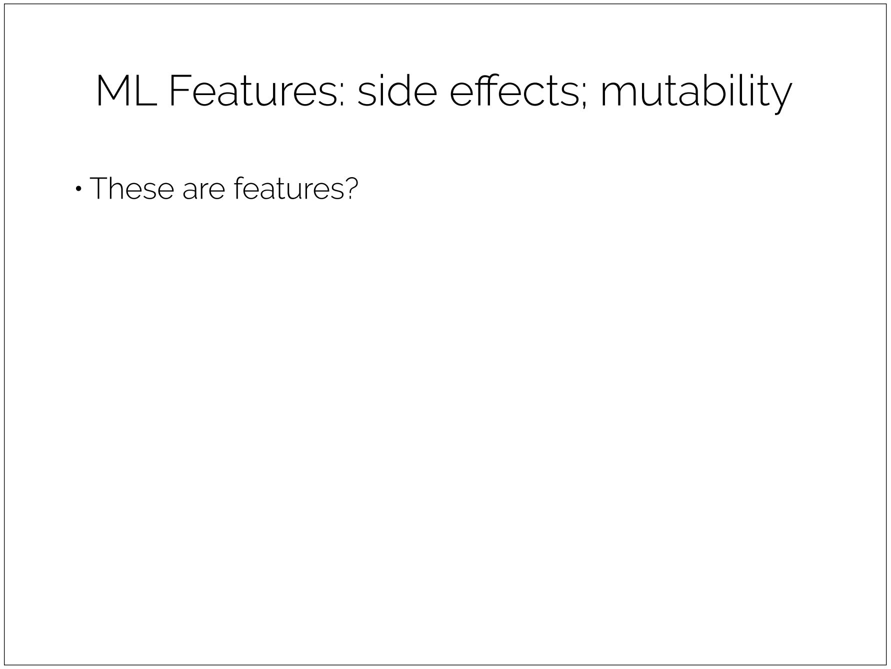
```
let foo() =
 exception DivByZero of string
 if x = 0 then raise DivByZero(“no zeros!”)

…

try
  foo()
with
| DivByZero msg -> do something else
```

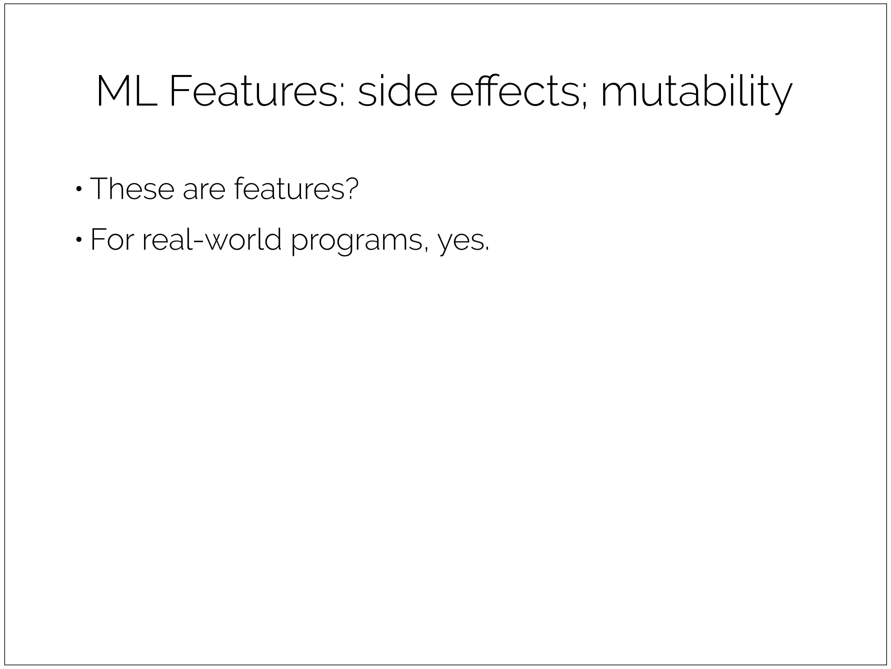


## ML Features: side effects; mutability



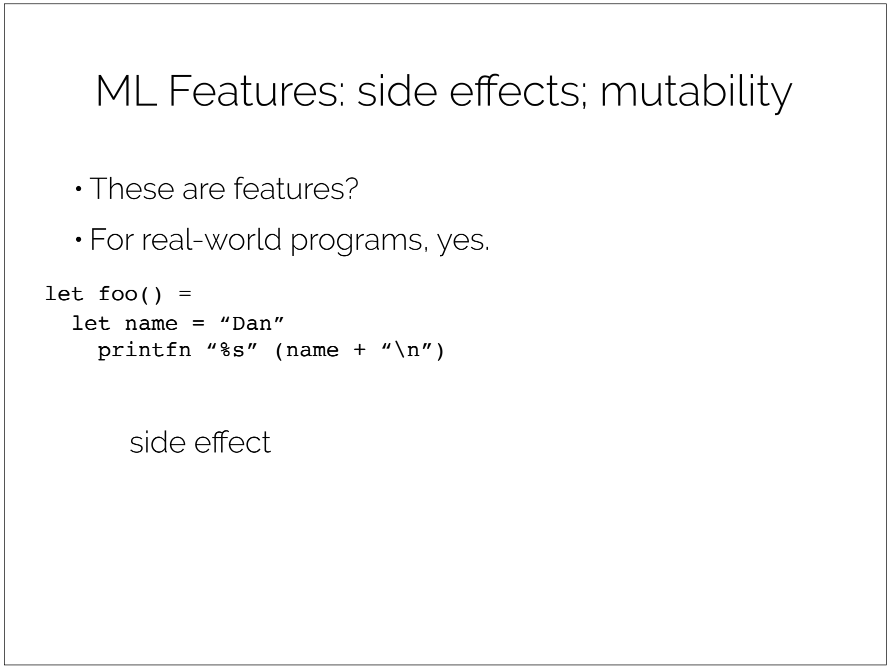
## ML Features: side effects; mutability

- These are features?



## ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.



## ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.

```
let foo() =
  let name = “Dan”
    printfn “%s” (name + “\n”)
```

side effect

## ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.

```
let foo() =
  let name = “Dan”
    printfn “%s” (name + “\n”)
```

```
let mutable x = 3
x <- 4
```

side effect

mutability

## ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.

```
let foo() =
  let name = “Dan”
    printfn “%s” (name + “\n”)
```

```
let mutable x = 3
x <- 4
```

side effect

mutability

- Both are often essential for speed

## ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.

```
let foo() =
  let name = “Dan”
    printfn “%s” (name + “\n”)
```

```
let mutable x = 3
x <- 4
```

side effect

mutability

- Both are often essential for speed
- But can be largely avoided in many programs for safety

## ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.

```
let foo() =
  let name = “Dan”
    printfn “%s” (name + “\n”)
```

```
let mutable x = 3
x <- 4
```

side effect

mutability

- Both are often essential for speed
- But can be largely avoided in many programs for safety
- Do not use these in this class unless instructed.

## Running F#

- Type `fsharpi` on Unix machines
- `#quit;;` to quit
- Enter expression or declarations to evaluate:

```
> 3 + 5;;
val it : int = 8
>it * 2;;
val it : int = 16
> let six = 3 + 3;;
val six : int = 6;;
```

## Defining Functions

No type info given- compiler infers it

- Example

```
> let succ x = x + 1;;
val succ : x:int -> int
> succ 12;;
val it : int = 13
> 17 * (succ 3);;
val it : int = 68
```

- Or:

```
> let succ = fun x -> x + 1;;
val succ : int -> int
```

## Recursion

- Most functions written using recursion and `if.. then.. else` (and patterns):

```
> let rec fact n =
    if n = 0 then 1 else n * fact (n-1);;
```

- `if..then..else` is an expression:

```
> if 3<4 then "moo" else “cow";;
val it : string = "moo"
```

  - types of both branches must match

## Local Declarations

```
> let cylinderVolume diameter height =
    let radius = diameter / 2.0
    let square y = y * y
    3.14 * square radius * height
  ;;
val cylinderVolume : float -> float -> float

> cylinderVolume 6.0 6.0;;
val it : float = 169.56
```

## Built-in Data Types

- unit
  - only value is `()`
- bool
  - `true, false`
  - `not, and, or`
- int
  - `..., -2, -1, 0, 1, 2, ...`
  - `+,-,*,/,%,abs`
  - `=,<,<=,<> etc.`

## Built-in Data Types

- float / double
  - `3.17, 2.2,` …
  - `+, -, *, /`
  - `=, <>, <, <=,` etc.
  - no implicit conversions from int to float:

    `2 + 3.3`

    is bad
  - Original ML had no equality for float (test that `-0.001 < x-y < 0.001`, etc.)
- strings
  - `"moo"`
  - `"moo" + “cow"`

## Overloaded Operators

- +,-,etc. defined on both int and float
- Which variant inferred depends on operands:

```
> let succ x = x + 1
val succ : int -> int

> let double x = x * 2.0
val double float -> float

> let double x = x + x
val double : int -> int
```

## Type Declarations

- Can add types when type inference does not work

```
- fun double (x:float) = x + x;
val double : float -> float

- fun double (x:float) : float = x + x;
val double : float -> float
```

## Compound Types

- Tuples, Records, Lists
- Tuples

```
(14, "moo", true): int * string * bool
```

- Functions can take tuple argument

```
> let rec power (exp,base) =
    if exp = 0 then 1
      else base * power(exp-1,base);;
val power: int -> int -> int
- power(3,2);;
```

## Curried Functions (named after Haskell Curry)

- Previous power

```
> let rec power (exp,base) =
      if exp = 0 then 1
         else base * power(exp-1,base);
val power: int * int -> int
```

- Curried power function

```
> let rec cpower exp base =
       if exp = 0 then 1
          else base * cpower (exp-1) base;;

val cpower: int -> (int -> int)
```

## Curried Functions (named after Haskell Curry)

## Curried Functions (named after Haskell Curry)

- Why is this useful?

```
> let cpower exp base =
       if exp = 0 then 1
          else base * cpower (exp-1) base;
val cpower : int -> (int -> int)
```

## Curried Functions (named after Haskell Curry)

- Why is this useful?

```
> let cpower exp base =
      if exp = 0 then 1
         else base * cpower (exp-1) base;
val cpower : int -> (int -> int)
```

- Can define

```
let square = cpower 2
val square : int -> int
- square 3;;
val it : int = 9
```