General outline
- Introduction
- Functional programming overview
- Type inference
- Running example
- Seminal approach

Functional programming review: Currying
- Let plus a b = a + b;
  plus : int -> int -> int
- Let pf = plus 5; What??
  (plus 5) : int -> int
  If you give it a number, it will add 5 to it
  This is a new function!
- Called Currying, or partial application

Functional programming review: Lists
- An empty list is written []
- Add one element:
  newElem :: oldList
  3 :: [4; 5; 6] == [3; 4; 5; 6]
- Modify all elements: map
  List.map (fun x -> 2*x) [3; 4; 5] = [6; 8; 10]
  List.map : ('a -> 'b) -> 'a list -> 'b list

Functional programming review: Pairs and tuples
- A pair of integers (5, 6) : int * int
- A triple (5, 'x', false) : int * char * bool
- Combining lists into pairs:
  List.combine [1;2;3] [4;5;6] =
  [[1,4]; [2,5]; [3,6]]
  List.combine : 'a list -> 'b list -> ('a * 'b) list

Type inference
- Why do I have to write types out all the time?
  C++:
  list<pair<int, char>>
  *picList = new
  list<pair<int, char>>();
  picList->push(new
  pair<int, char>(42, 'x'))

- Why can't the compiler figure it out for me?
  ML:
  let icList = ref [];
  icList := (42,'x') ::
  !icList;
Type inference

It can!

... it takes a little bit of work to do

Example: Curried functions

```
# let map2 f aList bList = List.map (fun (a, b) -> f a b) (List.combine aList bList);;
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
# map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6];;

This expression has type int but is here used with type 'a -> 'b
```

How are messages generated?

Existing compilers:

```
class AddExpr extends Expr {
  Expr arg1;
  Expr arg2;
  ResolvedType typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
    ResolvedType arg1_type = arg1.typecheck(st);
    ResolvedType arg2_type = arg2.typecheck(st);
    arg1_type.checkIsInt();
    arg2_type.checkIsInt();
    return ResolvedType.intType();
  }
}
```
How are messages generated?

**SEMINAL:**

![Diagram](https://via.placeholder.com/150)

Our approach, in one slide

- Treats type checker as oracle
- Makes no assumptions about type system
- Tries many variations on program, see which ones type-check
- “Variant type-checks” ≠ “Variant is right”
- Ranks successful suggestions, presents results to programmer
- Programmer knows which are right

Example: Curried functions

```
# let map2 f xs ys = List.map (fun (x, y) -> f x y) (List.combine xs ys);
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list

# map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6];;
val it : int list = [5;7;9]
```

Finding the changes, part 0

Change

```
let map2 f aList bList = ... ;;
map2 (fun (x, y) -> x+y) [1;2;3] [4;5;6]
```

Into...

```
map2 ([x,y] -> x+y) [1;2;3] [4;5;6]
```

Finding the changes, part 1

Change

```
map2 (fun (x, y) -> x+y) [1;2;3] [4;5;6]
```

Info...

- ✗ (fun (x,y)->x+y) [1;2;3] [4;5;6]
- ✗ map2 (fun (x,y)->x+y) [1;2;3] [4;5;6]
- ✓ (fun (x,y)->x+y) [1;2;3] [4;5;6]
- ✓ map2 (fun (x,y)->x+y) [1;2;3] [4;5;6]
- ✗ map2 (fun (x,y)->x+y) [1;2;3] [4;5;6]
- ✗ map2 (fun (x,y)->x+y) [1;2;3] [4;5;6]

Finding the changes, part 2

Change

```
(fun (x, y) -> x + y)
```

Into...

```
(fun (y, x) -> x + y)
```

```
(fun (x, y) -> x + y)
```

```
(fun (x, y) -> x + y)
```

```
(fun (y, x) -> x + y)
```

Note: ❌ is a placeholder that always type-checks
Ranking the suggestions

- Replace `map2 (fun (x,y)->x+y) [1;2;3] [4;5;6]` with `map2`
- Replace `map2` with `map2`
- Replace `(fun (x,y)->x+y)` with `(fun (x,y)->x+y)`
- Replace `(fun (x,y)->x+y)` with `(fun x y -> x+y)`
- Prefer smaller changes over larger ones
- Prefer non-deleting changes over others

Tidying up

- Find type of replacement
  - Get this for free from TC
  - Maintain surrounding context
  - Help user locate the replacement

Suggestions:

```
fun (x, y) -> x + y
with
  fun x y -> x + y
of type
  int -> int -> int
within context
{map2 (fun x y -> x + y) [1; 2; 3] [4; 5; 6]}
```

Conclusions

- Searching for error messages works!
  - Yields useful messages as often as current compiler
  - Can be improved easily
- Good for programmers, good for compiler writers