Functional Programming (1/2)

Martin Kellogg

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation
- common paradigms include:
    - **imperative**: change state, assignments

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation
- common paradigms include:
  - **imperative**: change state, assignments
  - **structured**: if/block/routine control flow

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation
- common paradigms include:
  - **imperative**: change state, assignments
  - **structured**: if/block/routine control flow
  - **object-oriented**: message passing (=dyn. dispatch), inheritance

Programming language paradigms

Definition: a language ***paradigm*** is a way to classify programming languages, usually by their style of structuring programs

- usually based on some kind of mathematical foundation
- common paradigms include:
  - **imperative**: change state, assignments
  - **structured**: if/block/routine control flow
  - **object-oriented**: message passing (=dyn. dispatch), inheritance
  - **functional**: functions are **first-class citizens** that can be passed around or called recursively. We can avoid changing state by passing copies.

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **???**

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what’s a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm
- models **actual computers** very well:
  - commands = ?
  - array that is destructively updated = ?

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm
- models **actual computers** very well:
  - commands = instructions to the processor
  - array that is destructively updated = ?

Imperative programming

Definition: in the ***imperative*** paradigm, programs are sequences of commands that destructively update one or more arrays

- key mathematical formalism: **Turing machines**
  - review: what's a Turing machine (on the whiteboard)?
- this is the single **most-common** programming paradigm
- models **actual computers** very well:
  - commands = instructions to the processor
  - array that is destructively updated = registers/memory/disk

Imperative programming: example

Consider the following C program:

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

Imperative programming: example

Consider the following C program:

semicolons separate commands, program is a list of commands

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

Imperative programming: example

Consider the following C program:

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

destructive updates of memory cells

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **?**

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**
    - in the lambda calculus, **everything is a function**

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**
  - in the lambda calculus, **everything is a function**
  - lambda calculus is **as powerful** as Turing machines
    - "as powerful" = anything you can compute with a Turing machine can also be computed with the lambda calculus

Functional programming

Definition: in the ***functional*** paradigm, programs are compositions of mathematical expressions (especially functions)

- key mathematical formalism: **lambda calculus**
  - in the lambda calculus, **everything is a function**
  - lambda calculus is **as powerful** as Turing machines
    - “as powerful” = anything you can compute with a Turing machine can also be computed with the lambda calculus
- functional programming **models math** well
  - it is easier to formally reason about functional programs

Functional programming: characteristics

- Computation = **evaluating** (math) functions

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions
- Avoid sequential commands

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions
- Avoid sequential commands
- Important Features of functional languages:
    - **Higher-order, first-class** functions
    - Closures and **recursion**
    - **Lists** and list processing

Functional programming: characteristics

- Computation = **evaluating** (math) functions
- Avoid “global state” and “mutable data”
- Get stuff done = apply (**higher-order**) functions
- Avoid sequential commands
- Important Features of functional languages:
  - **Higher-order, first-class** fr
  - Closures and **recursion**
  - **Lists** and list processing

Let’s look at how imperative and functional languages **manage state** in a bit more detail

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.
    - e.g., after executing `*x = y` (in a C program), the memory cell that `x` points to now holds the value `y`. Its old value is gone.

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.
    - e.g., after executing `*x = y` (in a C program), the memory cell that `x` points to now holds the value `y`. Its old value is gone.
- **Functional** programs yield **new similar states** over time.

State management: functional vs imperative

Definition: The ***state*** of a program is all of the current variable and heap values

- **Imperative** programs **destructively update** the state.
  - e.g., after executing `*x = y` (in a C program), the memory cell that `x` points to now holds the value `y`. Its old value is gone.
- **Functional** programs yield **new similar states** over time.
  - `let x = y in …` , however, only changes `x`'s value **within** the scope of the …

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z;
  z
end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin



end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

NOT the same as a semi-colon: commands vs expressions

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in




end
```

Aside: commands vs expressions

Aside: commands vs expressions

Definition: An ***expression*** is a syntactic entity in a programming language that may be evaluated to determine its value.

- e.g., the expression “5 + 3” can be evaluated to “8”

Aside: commands vs expressions

Definition: An ***expression*** is a syntactic entity in a programming language that may be evaluated to determine its value.

- e.g., the expression “5 + 3” can be evaluated to “8”

Definition: A ***command*** is a syntactic entity in a programming language which causes some computation (or ***side-effect***) to occur, but which does not itself evaluate to a value

- e.g., a call to printf prints something to the terminal, but doesn’t actually evaluate to anything

Aside: commands vs expressions

Definition: An ***expression*** is a syntac

language that may be evaluated to d

- e.g., the expression "5 + 3" can be

We'll come back to this later in the course, when we discuss **operational semantics**

Definition: A ***command*** is a syntactic entity in a programming language which causes some computation (or ***side-effect***) to occur, but which does not itself evaluate to a value

- e.g., a call to printf prints something to the terminal, but doesn't actually evaluate to anything

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

NOT the same as a semi-colon: commands vs expressions

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in



end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in


end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

even the operators are type-safe (in OCaml)

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in


end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;

end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

commands still exist, but limited to inherently "imperative" operations (I/O, saving to disk, etc.)

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;

end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;
  z
end
```

Basic functional programming

Let's **translate** this C program into OCaml

```
double avg(int x, int y) {
  double z = (double)(x + y);
  z = z / 2;
  printf("Answer: %g\n", z);
  return z;
}
```

no "return" statement, because everything is an expression

```
let avg (x:int) (y:int) : float = begin
  let z = float_of_int (x + y) in
  let z = z /. 2.0 in
  printf "Answer: %g\n" z ;
  z
end
```

Tuples and Pairs

Definition: a ***tuple*** (or ***pair***) is the Cartesian product of two types

Tuples and Pairs

Definition: a ***tuple*** (or ***pair***) is the Cartesian product of two types

- you actually already know this concept ("from 10th grade")
- e.g., what is a "**point**" in your 10th-grade math class?

Tuples and Pairs

Definition: a ***tuple*** (or ***pair***) is the Cartesian product of two types

- you actually already know this concept ("from 10th grade")
- e.g., what is a "**point**" in your 10th-grade math class?

```
let x = (22, 58) in
let y, z = x in
printf "1st element: %d" y;
...
```

Tuples and Pairs

Definition: a ***tuple*** (or ***pair***) is the Cartesian product of two types

- you actually already know this concept ("from 10th grade")
- e.g., what is a "**point**" in your 10th-grade math class?

tuple creation

```
let x = (22, 58) in
let y, z = x in
printf "1st element: %d" y;
...
```

Tuples and Pairs

Definition: a ***tuple*** (or ***pair***) is the Cartesian product of two types

- you actually already know this concept ("from 10th grade")
- e.g., what is a "**point**" in your 10th-grade math class?

```
let x = (22, 58) in
let y, z = x in
printf "1st element: %d" y;
...
```

tuple field extraction

Tuples and Pairs

Definition: a ***tuple*** (or ***pair***) is the Cartesian product of two types

- you actually already know this concept ("from 10th grade")
- e.g., what is a "**point**" in your 10th-grade math class?

```
let x = (22, 58) in
let y, z = x in
printf "1st element: %d" y;
...
```

point example:

```
let add_points p1 p2 =
  let x1, y1 = p1 in
  let x2, y2 = p2 in
  (x1 + x2, y1 + y2)
```

Lists are Your Friends

[Concept] [OCaml Syntax]

Lists are Your Friends

Concept	OCaml Syntax
• Empty list	`[ ]`

Lists are Your Friends

Concept	OCaml Syntax
• Empty list	`[ ]`
• Singleton	`[ element ]`

Lists are Your Friends

Concept	**OCaml Syntax**
• Empty list	`[ ]`
• Singleton	`[ element ]`
• Longer list	`[ e1 ; e2 ; e3 ]`

Lists are Your Friends

Concept	OCaml Syntax	
Empty list	`[ ]`	
Singleton	`[ element ]`	
Longer list	`[ e1 ; e2 ; e3 ]`	
Cons	`x::[y;z]`	= [x;y;z]

Aside: “cons”, “car”, and “cdr”

- ***cons*** is a fundamental operation from Lisp, the first practical functional programming language (invented in the 1950s)
    - and 2nd higher-order language that’s still in use, after Fortran

Aside: “cons”, “car”, and “cdr”

- ***cons*** is a fundamental operation from Lisp, the first practical functional programming language (invented in the 1950s)
  - and 2nd higher-order language that’s still in use, after Fortran
- It’s named “cons” because it ***cons***tructs memory objects which hold two values or pointers to two values

Aside: “cons”, “car”, and “cdr”

- ***cons*** is a fundamental operation from Lisp, the first practical functional programming language (invented in the 1950s)
  - and 2nd higher-order language that’s still in use, after Fortran
- It’s named “cons” because it ***cons***tructs memory objects which hold two values or pointers to two values
  - e.g., `cons 2 3` in Lisp would create the pair `(2, 3)`

Aside: “cons”, “car”, and “cdr”

- ***cons*** is a fundamental operation from Lisp, the first practical functional programming language (invented in the 1950s)
  - and 2nd higher-order language that’s still in use, after Fortran
- It’s named “cons” because it ***cons***tructs memory objects which hold two values or pointers to two values
  - e.g., `cons 2 3` in Lisp would create the pair `(2, 3)`
  - it’s used as shorthand for similar operations in modern FP

Aside: “cons”, “car”, and “cdr”

- ***cons*** is a fundamental operation from Lisp, the first practical functional programming language (invented in the 1950s)
  - and 2nd higher-order language that’s still in use, after Fortran
- It’s named “cons” because it ***cons***tructs memory objects which hold two values or pointers to two values
  - e.g., `cons 2 3` in Lisp would create the pair `(2, 3)`
  - it’s used as shorthand for similar operations in modern FP
- you might also here “**car**” and “**cdr**” to refer to the first (resp. second) elements of a cons-pair (also historical Lisp terminology)

Lists are Your Friends

Concept	OCaml Syntax	
• Empty list	`[ ]`	
• Singleton	`[ element ]`	
• Longer list	`[ e1 ; e2 ; e3 ]`	
• Cons	`x::[y;z]`	= [x;y;z]

Lists are Your Friends

Concept	OCaml Syntax	
• Empty list	`[ ]`	
• Singleton	`[ element ]`	
• Longer list	`[ e1 ; e2 ; e3 ]`	
• Cons	`x::[y;z]`	= [x;y;z]
• Append	`[w;x]@[y;z]`	= [w;x;y;z]

Lists are Your Friends

Concept	OCaml Syntax	
• Empty list	`[ ]`	
• Singleton	`[ element ]`	
• Longer list	`[ e1 ; e2 ; e3 ]`	
• Cons	`x::[y;z]`	= [x;y;z]
• Append	`[w;x]@[y;z]`	= [w;x;y;z]

All lists must be ***homogenous*** (i.e., all elements must **have same type**)

Functional examples

Functional examples

- Simple function set (built out of lists):

```
let rec add_elem (s, e) =
    if s = [] then [e]
    else if List.hd s = e then s
    else List.hd s :: add_elem(List.tl s, e)
```

Functional examples

- Simple function set (built out of lists):

```
let rec add_elem (s, e) =
    if s = [] then [e]
    else if List.hd s = e then s
    else List.hd s :: add_elem(List.tl s, e)
```

- Same function using pattern matching instead:

```
let rec add_elem (s, e) = match s with
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem(tl, e)
```

Equivalent Imperative (C) Code

```
List* add_elem(List *s, item e) {
  if (s == NULL) {
    return list(e, NULL);
  } else if (s->hd == e) {
    return s;
  } else if (s->tl == NULL) {
    s->tl = list(e, NULL);
    return s;
  } else {
    return add_elem(s->tl, e);
  }
}
```

Equivalent Imperative (C) Code

```
List* add_elem(List *s, item e) {
  if (s == NULL) {
    return list(e, NULL);
  } else if (s->hd == e) {
    return s;
  } else if (s->tl == NULL) {
    s->tl = list(e, NULL);
    return s;
  } else {
    return add_elem(s->tl, e);
  }
}
```

More cases to handle!

Functional advantages

Functional advantages

- Tractable program semantics
  - Procedures are functions (simplifies reasoning)
  - Formulate and prove assertions about code more easily
  - More readable (if you like math)

Functional advantages

- Tractable program semantics
  - Procedures are functions (simplifies reasoning)
  - Formulate and prove assertions about code more easily
  - More readable (if you like math)
- ***Referential transparency***
  - Replace any expression by its value without changing the result

Functional advantages

- Tractable program semantics
  - Procedures are functions (simplifies reasoning)
  - Formulate and prove assertions about code more easily
  - More readable (if you like math)
- ***Referential transparency***
  - Replace any expression by its value without changing the result
- “No” side-effects
  - Fewer errors

Functional disadvantages

Functional disadvantages

- Efficiency
  - Copying takes time

Functional disadvantages

- Efficiency
  - Copying takes time

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Functional disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Functional disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation
- Unfamiliar (to you!)
  - New programming style

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Functional disadvantages

- Efficiency
  - Copying takes time
- Compiler implementation
  - Frequent memory allocation
- Unfamiliar (to you!)
  - New programming style
- Not appropriate for every program
  - Some programs are inherently stateful (e.g., operating systems)

Language	Speed	Space
C (gcc)	1.0	1.1
C++ (g++)	1.0	1.6
OCaml	1.5	2.9
Java (JDK -server)	1.7	9.1
Lisp	1.7	11
C# (mono)	2.4	5.6
Python	6.5	3.9
Ruby	16	5.0

17 small benchmarks

Trivia Break: Computer Science History

This American computer scientist and mathematician was born in Washington, DC, in 1903. While a professor at Princeton, he advised Alan Turing's doctoral dissertation. He is known for inventing the lambda calculus, though he made many other contributions to mathematics, computer science, and philosophy.

Trivia Break: Cuisine

This dish is a sauce or gravy seasoned with spices, mainly derived from the interchange of Indian cuisine with European cuisine following the Columbian Exchange. Many types of this dish exist in different international cuisines. For example, in Southeast Asia, it often contains a spice paste and coconut milk. In India, the spices are fried in oil or ghee to create a paste. In Britain, this dish is regarded as national dish; some types were adopted from India, but others—such as Chicken Tikka Masala—were wholly invented in Britain in the 20th century.

ML’s innovative features

- Type system
  - Strongly typed
  - Type inference
  - Abstraction
- Modules
- Patterns
- Polymorphism
- Higher-order functions
- Concise formal semantics

ML's innovative features

- Type system
  - Strongly typed
  - Type inference
  - Abstraction
- Modules
- Patterns
- Polymorphism
- Higher-order functions
- Concise formal semantics

> *There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.*
>
> \- **Robin Milner**, 1997

Types

Definition: A ***type*** is a conservative over-approximation of the set of values an expression could possibly take on at run-time.

Types

Definition: A ***type*** is a conservative over-approximation of the set of values an expression could possibly take on at run-time.

- If `x+3` has type `Int`, then `x+3` could evaluate to 7 or -2 or 5102 at run-time, but not "Hello" or 1.2

Types

Definition: A ***type*** is a conservative over-approximation of the set of values an expression could possibly take on at run-time.

- If `x+3` has type `Int`, then `x+3` could evaluate to 7 or -2 or 5102 at run-time, but not "Hello" or 1.2
- To say that expression E has type T, we write:

  $$E : T$$

Types

Definition: A ***type*** is a conservative over-approximation of the set of values an expression could possibly take on at run-time.

- If `x+3` has type `Int`, then `x+3` could evaluate to 7 or -2 or 5102 at run-time, but not “Hello” or 1.2
- To say that expression E has type T, we write:

  E : T

- Types help us find bugs early
  - Requiring types to match up can rule out bad programs without even having to test them!

Aside: Why catch bugs earlier?

Aside: Why catch bugs earlier?

- An IBM report gives an average defect repair cost of (2008$):
  - $25 during coding
  - $100 at build time
  - $450 during testing/QA
  - $16,000 post-release

[L. Williamson. IBM Rational software analyzer: Beyond source code. 2008.]

ML Type System

- Type Inference

ML Type System

- Type Inference

```
let rec add_elem (s, e) = match s with
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem(tl, e)
```

ML Type System

- Type Inference

```
let rec add_elem (s, e) = match s with
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem(tl, e)
    val add_elem : α list * α -> α list
```

ML Type System

- Type Inference

```
let rec add_elem (s, e) = match s with
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem(tl, e)
    val add_elem : α list * α -> α list
```

- α means “works for any type (your choice)”
  - “α list” means “List<T>” or “List<α>”

ML Type System

- Type Inference

```
let rec add_elem (s, e) = match s with
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem(tl, e)
    val add_elem : α list * α -> α list
```

- α means “works for any type (your choice)”
  - “α list” means “List<T>” or “List<α>”
- ML infers (all!) types: inconsistent types are errors

ML Type System

- Type Inference

```
let rec add_elem (s, e) = match s with
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem(tl, e)
    val add_elem : α list * α -> α list
```

- α means “works for any type (your choice)”
  - “α list” means “List<T>” or “List<α>”
- ML infers (all!) types: inconsistent types are errors
- Optional type declarations `( exp : type )`
  - Clarify ambiguous cases, documentation

ML Type System

- Type Inference

```
let rec add_elem (s, e) = match
    | [] -> [e]
    | hd :: tl when e = hd -> s
    | hd :: tl -> hd :: add_elem
    val add_elem : α list * α ->
```

- α means “works for any type (your
  - “α list” means “List<T>” or “List<α>
- ML infers (all!) types: inconsistent types are errors
- Optional type declarations (exp : type)
  - Clarify ambiguous cases, documentation

You might be tempted to ask “How does ML infer types?” Unfortunately, this is a complex topic. Ask in OH if you’re curious, or take a PhD-level seminar from me or Iulian Neamtiu.

Pattern Matching

- Simplifies code (eliminates ifs, accessors)

Pattern Matching

- Simplifies code (eliminates ifs, accessors)

```
type btree = (* binary tree of strings *)
  | Node of btree * string * btree
  | Leaf of string
```

Pattern Matching

- Simplifies code (eliminates ifs, accessors)

```
type btree = (* binary tree of strings *)
   | Node of btree * string * btree
   | Leaf of string
let rec height tree = match tree with
   | Leaf _ -> 1
   | Node(x,_,y) -> 1 + max (height x) (height y)
```

Pattern Matching

- Simplifies code (eliminates ifs, accessors)

```
type btree = (* binary tree of strings *)
   | Node of btree * string * btree
   | Leaf of string
let rec height tree = match tree with
   | Leaf _ -> 1
   | Node(x,_,y) -> 1 + max (height x) (height y)
let rec mem tree elt = match tree with
   | Leaf str -> str = elt
   | Node(x,str,y) -> str = elt || mem x elt || mem y elt
```

Pattern Matching Mistakes

- What if I forget a case? E.g.,

```
let rec is_odd x = match x with
  | 0 -> false
  | 2 -> false
  | x when x > 2 -> is_odd (x-2)
```

Pattern Matching Mistakes

- What if I forget a case? E.g.,

```
let rec is_odd x = match x with
   | 0 -> false
   | 2 -> false
   | x when x > 2 -> is_odd (x-2)
```

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 1
```

Polymorphism

- Functions and type inference are ***polymorphic***

Polymorphism

- Functions and type inference are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

Polymorphism

- Functions and type inference are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl
```

Polymorphism

- Functions and type inference are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int
```

Polymorphism

- Functions and type inference are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int
```

Recall that α means “any one type”

Polymorphism

- Functions and type inference are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl

val length : α list -> int

length [1;2;3] = 3
```

Recall that α means “any one type”

Polymorphism

- Functions and type inference are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl
```

Recall that α means “any one type”

```
val length : α list -> int
```

```
length [1;2;3] = 3
length [“algol”; ”smalltalk”; ”ml”] = 3
```

Polymorphism

- Functions and type inference are ***polymorphic***
  - "Polymorphic" means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl
```

Recall that α means "any one type"

```
val length : α list -> int
```

```
length [1;2;3] = 3
length [“algol”; ”smalltalk”; ”ml”] = 3
length [1 ; “algol” ] = ?
```

Polymorphism

- Functions and type inference are ***polymorphic***
  - “Polymorphic” means they operate on more than one type

```
let rec length x = match x with
  | [] -> 0
  | hd :: tl -> 1 + length tl
```

Recall that α means “any one type”

```
File "list-example.ml", line 1, characters 25-26:
1 | let myList = [ "algol" ; 1 ] in
                             ^
Error: This expression has type int but an expression was expected of type
         string
```

```
length [1 ; “algol” ] = ?
```

Higher-order functions

- Functions are first-class values

Higher-order functions

- Functions are first-class values
    - Can be used whenever a value is expected (i.e., as an *expression*)

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around

Higher-order functions

- Functions are first-class values
    - Can be used whenever a value is expected (i.e., as an *expression*)
    - Notably, can be passed around
    - Closure captures the environment

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
```

f is itself a function!

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> β list
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> β list
let offset = 10 in
let myfun x = x + offset in
val myfun : ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> β list
let offset = 10 in
let myfun x = x + offset in
val myfun : int -> int
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> β list
let offset = 10 in
let myfun x = x + offset in
val myfun : int -> int
map myfun [1;8;22] = ?
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> β list
let offset = 10 in
let myfun x = x + offset in
val myfun : int -> int
map myfun [1;8;22] = [11;18;32]
```

Higher-order functions

- Functions are first-class values
  - Can be used whenever a value is expected (i.e., as an *expression*)
  - Notably, can be passed around
  - Closure captures the environment

```
let rec map f lst = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
val map : (α -> β) -> α list -> β lis
let offset = 10 in
let myfun x = x + offset in
val myfun : int -> int
map myfun [1;8;22] = [11;18;32]
```

Extremely powerful programming technique:

- general iterators
- implement abstraction

The Story of Fold

- We’ve seen **length** and **map**
- We can also imagine:

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14

The Story of Fold

- We’ve seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40
  - **and** [true; true; false] = false

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40
  - **and** [true; true; false] = false
  - **or** [true; true; false] = true

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40
  - **and** [true; true; false] = false
  - **or** [true; true; false] = true
  - **filter** (fun x -> x > 4) [1; 5; 8] = [5; 8]

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:

sum	[1; 5; 8]	= 14
product	[1; 5; 8]	= 40
and	[true; true; false]	= false
or	[true; true; false]	= true
filter	(fun x -> x > 4) [1; 5; 8]	= [5; 8]
reverse	[1; 5; 8]	= [8; 5; 1]

The Story of Fold

- We've seen **length** and **map**
- We can also imagine:
  - **sum** [1; 5; 8] = 14
  - **product** [1; 5; 8] = 40
  - **and** [true; true; false] = false
  - **or** [true; true; false] = true
  - **filter** (fun x -> x > 4) [1; 5; 8] = [5; 8]
  - **reverse** [1; 5; 8] = [8; 5; 1]
  - **mem** 5 [1; 5; 8] = true

The Story of Fold

How can we **build** all of these?

- We've seen **length** and **map**
- We can also imagine:
    - **sum** [1; 5; 8] = 14
    - **product** [1; 5; 8] = 40
    - **and** [true; true; false] = false
    - **or** [true; true; false] = true
    - **filter** (fun x -> x > 4) [1; 5; 8] = [5; 8]
    - **reverse** [1; 5; 8] = [8; 5; 1]
    - **mem** 5 [1; 5; 8] = true

The Story of Fold

- The *fold* operator comes from recursion theory (Kleene, 1952):

```
let rec fold f acc lst = match lst with
    | [] -> acc
    | hd :: tl -> fold f (f acc hd) tl
```

The Story of Fold

- The *fold* operator comes from recursion theory (Kleene, 1952):

```
let rec fold f acc lst = match lst with
    | [] -> acc
    | hd :: tl -> fold f (f acc hd) tl

val fold : ?
```

The Story of Fold

- The *fold* operator comes from recursion theory (Kleene, 1952):

```
let rec fold f acc lst = match lst with
    | [] -> acc
    | hd :: tl -> fold f (f acc hd) tl

val fold : (α -> β -> α) -> α -> β list -> α
```

The Story of Fold

- The *fold* operator comes from recursion theory (Kleene, 1952):

```
let rec fold f acc lst = match lst with
    | [] -> acc
    | hd :: tl -> fold f (f acc hd) tl

val fold : (α -> β -> α) -> α -> β list -> α
                f               acc    lst     (fold f acc lst)
```

The Story of Fold

- The *fold* operator comes from recursion theory (Kleene, 1952):

```
let rec fold f acc lst = match lst with
    | [] -> acc
    | hd :: tl -> fold f (f acc hd) tl
```

Note: acc type and return type are the same!

```
val fold : (α -> β -> α) -> α -> β list -> α
                f             acc    lst     (fold f acc lst)
```

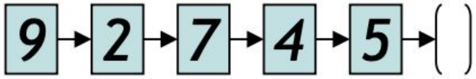
The Story of Fold

- The *fold* operator comes from recursion theory (Kleene, 1952):

```
let rec fold f acc lst = match lst with
    | [] -> acc
    | hd :: tl -> fold f (f acc hd) tl

val fold : (α -> β -> α) -> α -> β list -> α
                 f              acc     lst     (fold f acc lst)
```

- on the whiteboard, this example (f is +):



Let’s build things out of fold

- **length** lst = fold (fun acc elt -> ???) ? lst

Let’s build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst

Let's build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst
- **sum** lst = fold (fun acc elt -> ???) ? lst

Let’s build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst
- **sum** lst = fold (fun acc elt -> acc + elt) 0 lst

Let’s build things out of fold

- **length** lst = <u>fold</u> (fun acc elt -> acc + 1) 0 lst
- **sum** lst = <u>fold</u> (fun acc elt -> acc + elt) 0 lst
- **product** lst = <u>fold</u> (fun acc elt -> ???) ? lst

Let’s build things out of fold

- **length** lst = <u>fold</u> (fun acc elt -> acc + 1) 0 lst
- **sum** lst = <u>fold</u> (fun acc elt -> acc + elt) 0 lst
- **product** lst = <u>fold</u> (fun acc elt -> acc * elt) 1 lst

Let’s build things out of fold

- **length** lst = <u>fold</u> (fun acc elt -> acc + 1) 0 lst
- **sum** lst = <u>fold</u> (fun acc elt -> acc + elt) 0 lst
- **product** lst = <u>fold</u> (fun acc elt -> acc * elt) 1 lst
- **and** lst = <u>fold</u> (fun acc elt -> ???) ? lst

Let’s build things out of fold

- **length** lst = <u>fold</u> (fun acc elt -> acc + 1) 0 lst
- **sum** lst = <u>fold</u> (fun acc elt -> acc + elt) 0 lst
- **product** lst = <u>fold</u> (fun acc elt -> acc * elt) 1 lst
- **and** lst = <u>fold</u> (fun acc elt -> acc & elt) true lst

Let’s build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst
- **sum** lst = fold (fun acc elt -> acc + elt) 0 lst
- **product** lst = fold (fun acc elt -> acc * elt) 1 lst
- **and** lst = fold (fun acc elt -> acc & elt) true lst

- think you can do **or** on your own?

Let's build things out of fold

- **length** lst = fold (fun acc elt -> acc + 1) 0 lst
- **sum** lst = fold (fun acc elt -> acc + elt) 0 lst
- **product** lst = fold (fun acc elt -> acc * elt) 1 lst
- **and** lst = fold (fun acc elt -> acc & elt) true lst

- think you can do **or** on your own?
  - what about **reverse**?

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> ???) ? lst

Let's build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst

Let's build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> ???) ? lst

Let's build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst

Let’s build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: (acc : α list) (e : α)
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = <u>fold</u> (fun acc elt -> ???) ? lst

Let's build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst
  - note types: (acc : α list) (e : α)
- **filter** keep_it lst = fold (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = fold (fun acc elt -> acc || wanted = elt) false lst

Let’s build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst
  - note types: (acc : α list) (e : α)
- **filter** keep_it lst = fold (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = fold (fun acc elt -> acc || wanted = elt) false lst
  - note types: (acc : bool) (e : α)

Let's build things out of fold, part 2

- **reverse** lst = fold (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = fold (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = fold (fun acc elt -> acc || wanted = elt) false lst
  - note types: **(acc : bool) (e : α)**
- Could we do **map**?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]

Let's build things out of fold, part 2

- **reverse** lst = <u>fold</u> (fun acc elt -> acc @ [e]) [] lst
  - note types: **(acc : α list) (e : α)**
- **filter** keep_it lst = <u>fold</u> (fun acc elt -> if keep_it elt
  then elt :: acc
  else acc) [] lst
- **filter** wanted lst = <u>fold</u> (fun acc elt -> acc || wanted = elt) false lst
  - note types: **(acc : bool) (e : α)**
- Could we do **map**?
  - Recall: map (fun x -> x +10) [1;2] = [11;12]
  - Let's do it together…

Let's build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt ->          ???        ) ? lst
```

Let's build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt -> (myfun elt) :: acc) [] lst
```

Let’s build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt -> (myfun elt) :: acc) [] lst
```

- Types of:
  - myfun : **α -> β**
  - lst : **α list**
  - acc : **β list**
  - elt : **α**

Let's build things out of fold, part 3 (map)

```
let map myfun lst =
        fold (fun acc elt -> (myfun elt) :: acc) [] lst
```

- Types of:
  - myfun : **α -> β**
  - lst : **α list**
  - acc : **β list**
  - elt : **α**
- Could we do **sort**?

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- <u>sort</u> (fun a b -> ???) langs = **[“algol”; “c”; “fortran”]**

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- sort (fun a b -> a < b) langs = [“algol”; “c”; “fortran”]

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- sort (fun a b -> a < b) langs = [“algol”; “c”; “fortran”]
- sort (fun a b -> ???) langs = **[“fortran”; “c”; “algol”]**

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- sort (fun a b -> a < b) langs = [“algol”; “c”; “fortran”]
- sort (fun a b -> a > b) langs = [“fortran”; “c”; “algol”]

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- <u>sort</u> (fun a b -> a < b) langs = [“algol”; “c”; “fortran”]
- <u>sort</u> (fun a b -> a > b) langs = [“fortran”; “c”; “algol”]
- <u>sort</u> (fun a b -> ???) langs = **[“c”; “algol”; “fortran”]**

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- sort (fun a b -> a < b) langs = [“algol”; “c”; “fortran”]
- sort (fun a b -> a > b) langs = [“fortran”; “c”; “algol”]
- sort (fun a b -> strlen a < strlen b) langs = [“c”; “algol”; “fortran”]

Sorting examples

let **langs = [“fortran”; “algol”; “c”]** in

- sort (fun a b -> a < b) langs = [“algol”; “c”; “fortran”]
- sort (fun a b -> a > b) langs = [“fortran”; “c”; “algol”]
- sort (fun a b -> strlen a < strlen b) langs = [“c”; “algol”; “fortran”]

- Recall Java’s **Comparator** interface
  - in this functional style, our implementations are much simpler!

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
```

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

- How do we know what this means? We use ***referential transparency***! Basically, just substitute it in.

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

- How do we know what this means? We use ***referential transparency***! Basically, just substitute it in.

```
val addtwo : int -> int
addtwo 77 = 79
```

Partial Application and Currying

```
let myadd x y = x + y
val myadd : int -> int -> int
myadd 3 5 = 8
let addtwo = myadd 2
```

- How do we know what this means? We use ***referential transparency***! Basically, just substitute it in.

```
val addtwo : int -> int
addtwo 77 = 79
```

- called ***Currying***: "if you fix some arguments, you get a function of the remaining arguments"

Course Announcements

- Don’t forget: PA1c1 **due today**
  - and PA1c2 (1 more language!) **due Thursday**
  - and PA1 (full, all four languages!) **due next Monday**
- Cool Reference Manual is assigned reading for Wednesday, too ;)
  - I ***certainly*** wouldn’t consider giving another quiz…

Course Announcements