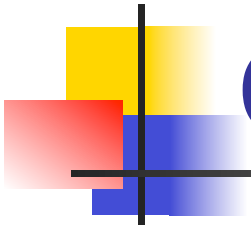
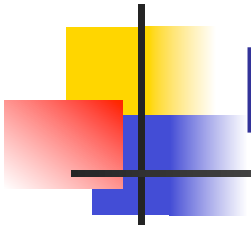


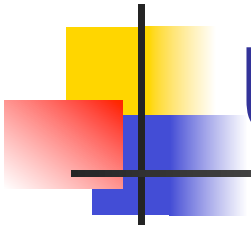
CS 11 Ocaml track: lecture 3

- Today:
  - A (large) variety of odds and ends
  - Imperative programming in Ocaml



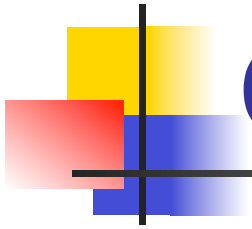
Equality/inequality operators

- Two inequality operators: **<>** and **!=**
- Two equality operators: **=** and **==**
- Usually want to use **=** and **<>**
- **=** means "structurally equal"
- **<>** means "structurally unequal"
- **==** means "the same exact object"
- **!=** means "not the same exact object"



Unit type

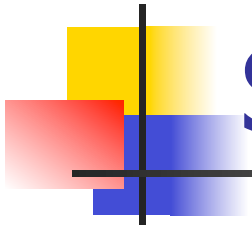
- The `unit` type is a type with only one member: `()`
  - not a tuple with only one element!
  - tuples must have at least two elements
- Seems useless, but
  - all Ocaml functions must return a value
  - return `()` when value is irrelevant
  - *i.e.* when function called for side effects



Option types

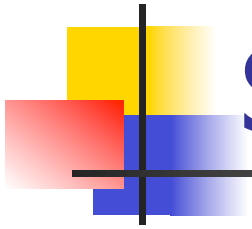
```
type 'a option =
   | None
   | Some of 'a
```

- Built in to Ocaml
- Used for functions that can return a value but can also "fail" (return `None`)
- Alternative to raising exception on failure



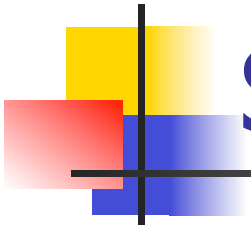
String accessing/mutating (1)

- Strings are not immutable
- Can treat as an array of chars
- To access a particular char:
  - `s.[i]`
- To mutate a particular char:
  - `s.[i] <- 'a'`



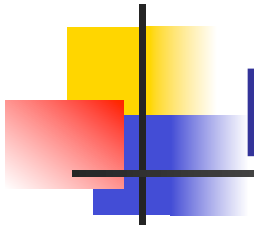
String accessing/mutating (2)

```
# let s = "some string" ;;
val s : string = "some string"
# s.[0] ;;  (* note weird syntax *)
- : char = 's'
# s.[0] <- 't' ;;
- : unit = ()
# s ;;
- : string = "tome string"
```



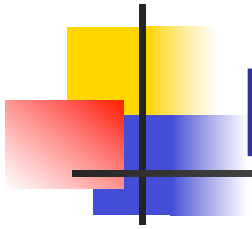
String accessing/mutating (3)

- String mutation is a misfeature!
  - only in the language because of historical reasons
  - most new languages have immutable strings
- Ocaml is starting to move towards immutable strings by default
  - with a "bytes" type for when you want a mutable string-like type
- The `-safe-string` option turns off the ability to mutate strings



printf and friends (1)

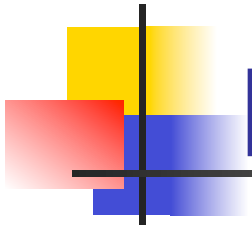
```
# Printf.printf "hello, world!\n" ;;
hello, world!
- : unit = ()
# open Printf ;;
# printf "hello, world!\n" ;;
hello, world!
- : unit = ()
```



printf and friends (2)

```
# printf "s = %s\tf = %f\ti = %d\n"
  "foo" 3.2 1 ;;
s = foo f = 3.200000    i = 1
- : unit = ()
```

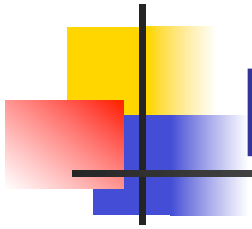
- `printf` has a weird type
  - not really well-typed
  - compiler "knows" about it and makes it work



printf and friends (3)

```
# fprintf stderr "Oops! An error occurred!\n" ;;
- : unit = ()
# stderr ;;
- : out_channel = <abstr>
```

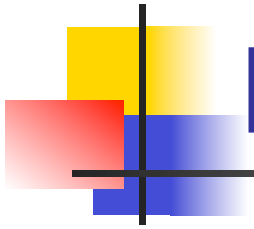
- Predefined I/O "channels":
- **stdin : in_channel**
- **stdout : out_channel**
- **stderr : out_channel**



printf and friends (4)

```
# sprintf "%d + %d = %d\n" 2 2 4 ;;
- : string = "2 + 2 = 4\n"
```

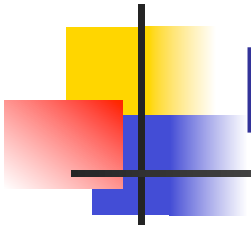
- **`sprintf`** is "printing to a string"
- Very useful!



File I/O (1)

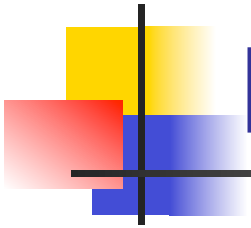
- Files come in two flavors: input and output

```
# open_in ;;
- : string -> in_channel = <fun>
# open_out ;;
- : string -> out_channel = <fun>
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```



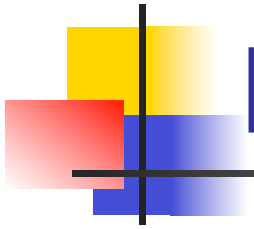
File I/O (2)

- Files come in two flavors: input and output
- `let infile = open_in "foo"`
  - tries to open file named `"foo"` for input only
  - binds file object to `infile`
- `close_in infile`
  - closes the input file



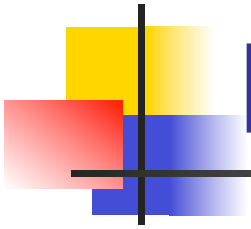
File I/O (3)

- Files come in two flavors: input and output
- `let outfile = open_out "bar"`
  - tries to open file named `"bar"` for output only
  - binds file object to `outfile`
- `close_out outfile`
  - closes the output file



File I/O (4)

- **`flush stdout`**
  - forces an output file (here, **`stdout`**) to write its buffers to the disk
- **`input_line stdin`**
  - gets a line of input from an input file (here, **`stdin`**) and returns a string

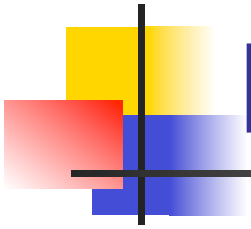


begin/end and sequencing (1)

- With side effects, often want multiple statements inside a function:

```
let print_and_square x =
    Printf.printf "%d\n" x ;
    x * x
```

- *Single* semicolon used to separate statements that execute one after another

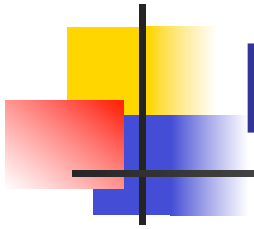


begin/end and sequencing (2)

- Sometimes want to say "these sequences should be treated as a single expression"
- Use **begin**/**end** for this:

```
begin
   Printf.printf "%d\n" x;
   x * x
end
```

- Can often leave out **begin**/**end**

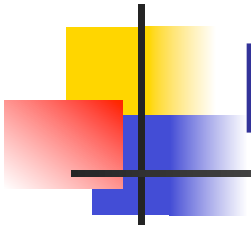


begin/end and sequencing (3)

- Sometimes can just use parentheses:

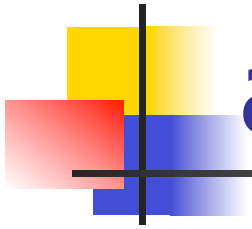
```
(Printf.printf "%d\n" x ;
  x * x)
```

- I advise against this
- Can make code hard to read



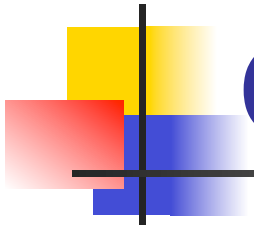
begin/end and sequencing (4)

- Very often, when you get weird error messages it's because you should have put in a **`begin`**/**`end`** somewhere
- Commonly found in nested **`match`** expressions (Ocaml grammar is highly ambiguous!)
- When in doubt, add explicit **`begin`**/**`end`** statements everywhere you use sequencing



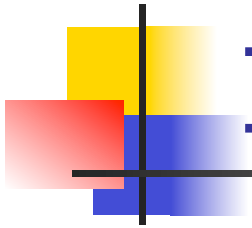
assert

- Ocaml has an **`assert`** statement like most imperative languages
- Not a function!
- Takes one "argument", a boolean
- If it's false, raises **`Assert_failure`** exception
- Turn off assertions with **`-noassert`** compiler option



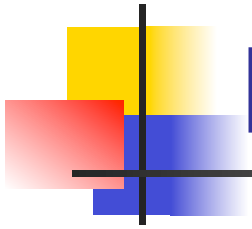
On to...

- Imperative programming!
- We've already done imperative programming
- **`printf`** is a function called for side-effects only
- **`begin`**/**`end`** and sequencing only useful for side effecting operations
- Now want to cover the "core" of imperative programming



Imperative programming

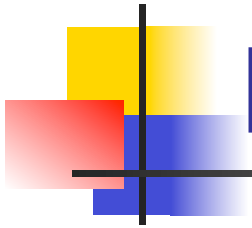
- Imperative data types:
  - references
  - records with mutable fields
  - mutable arrays
- Imperative statements:
  - `for` loop
  - `while` loop
- Breaking out of loops



References (1)

- A reference type is like a "box" that holds a single value:

```
# let x = ref 0 ;;
val x : int ref = {contents = 0}
# !x ;;
- : int = 0
```

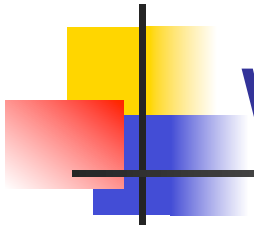


References (2)

- The `!` operator fetches the value from the reference "box"
- The `:=` operator assigns a new value to the reference

```
# x := 10 ;;
- : unit = ()
# x  ;;
- : int ref = {contents = 10}
```

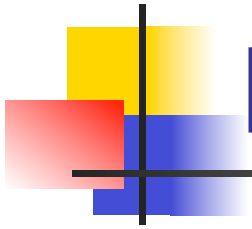
- LHS of `:=` *must* be a reference, not a value!



while loop

- **while** loop is basically like C/C++/Java while loop:

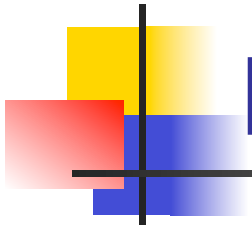
```
while <condition> do
    <stmt1>;
    <stmt2>;
    ...
    <stmtn>
done
```



Example

```
let factorial n =
    let result = ref 1 in
    let i = ref n in
        while !i > 1 do
            result := !result * !i;
            i := !i - 1
        done;
    !result
```

- *Very* easy to accidentally omit `!` operators

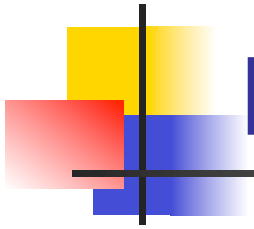


Records with mutable fields (1)

- References are just a special case of records with mutable fields
- Recall record type declaration:

```
type point = { x: int; y: int }
```

- This declares `point` as an *immutable* type
  - `x` and `y` fields can't change after point created
  - not always what you want



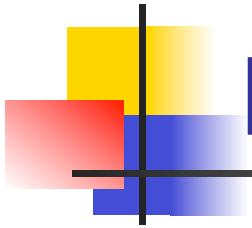
Records with mutable fields (2)

- To get mutable fields:

```
type point = { mutable x: int;
               mutable y: int }
```

- Now can change x, y fields:

```
let p = { x = 10; y = 20 } ;;
val p : point = {x = 10; y = 20}
# p.x <- 1000 ;;
- : unit = ()
# p ;;
- : point = {x = 1000; y = 20}
```



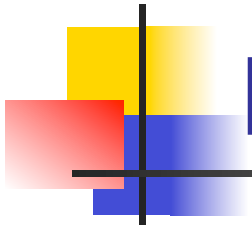
Records with mutable fields (3)

- To get only some mutable fields:

```
type point = { x: int; mutable y: int }
```

- Now can change only change `y` field:

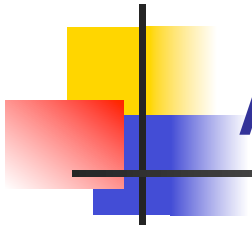
```
# let p = { x = 10; y = 20 } ;;
val p : point = {x = 10; y = 20}
# p.x <- 1000 ;;
The record field label x is not mutable
```



Records with mutable fields (4)

- The `<-` record mutation operator is not a true operator
  - Just built-in syntax
- The `!` and `:=` reference operators *are* true operators:

```
# (!) ;;
- : 'a ref -> 'a = <fun>
# (:=) ;;
- : 'a ref -> 'a -> unit = <fun>
```



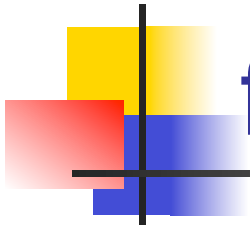
Arrays

- Recall: literal arrays:

```
# let arr = [| 10; 20; 30; 40; 50 |] ;;
```

- Arrays are <u>always</u> mutable:

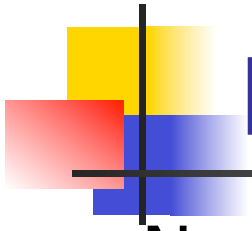
```
# arr.(0) ;;
- : int = 10
# arr.(0) <- 1000 ;;  (* same syntax as records *)
- : unit = ()
# arr ;;
- : int array = [| 1000; 20; 30; 40; 50 |]
```



for loops

```
# for i = 1 to 10 do
    Printf.printf "%d " i
done;
Printf.printf "\n";;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

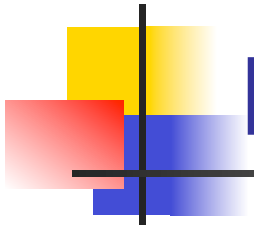
- Index variable `i` assigned values from 1 to 10, inclusive
- Don't need to use `!i` syntax to refer to `i`'s value



Breaking out of loops

- No **`break`** statement like in C/C++/Java
- Instead, raise an **`Exit`** exception and catch it:

```
# try
  for i = 1 to 10 do
    if i = 5 then raise Exit  (* like a "break" *)
    else Printf.printf "%d " i
  done
with Exit -> Printf.printf "\n";;
1 2 3 4
- : unit = ()
```



Next time

- Modules
- Functors