

CS 421 — Programming Languages and Compilers

Welcome!

In today's class:

- Course intro
 - What CS 421 is about
 - Class structure
- Brief discussion of programming language history
- Intro to the OCaml programming language

CS 421 — Class 1, 1/17/12 — 1

Course staff

Professor:	Sam Kamin	11–12TR
	kamin	4237 SC
TAs:	Kyle Blocher	t.b.a.
	blocher1	
	Dwight Guth	2–3MF
	dguth2	1218 SC
	Rajesh Karmani	3–4M, 11–12R
	rkumar8	0207 SC

- Web: www.cs.illinois.edu/class/cs421
- Piazza: piazza.com

CS 421 — Class 1, 1/17/12 — 2

What is 421 about?

Design and implementation of programming languages

- First quarter: Lexing and parsing
- Second quarter: Definition and implementation of Java-like languages
- Third quarter: Definition, implementation, and use of functional languages
- Fourth quarter: Various topics: lambda calculus; program verification; parallel programming languages

CS 421 — Class 1, 1/17/12 — 3

Why learn about lexing and parsing?

Learn basic algorithms for dealing with structured textual data, applying theory from CS 373:

- DFAs and regular expressions for “lexing”
- Top-down (aka recursive descent) parsing
- Bottom-up (aka LR, aka shift/reduce) parsing

CS 421 — Class 1, 1/17/12 — 4

Why learn about definition and implementation of Java-like languages?

- Understand standard languages more precisely
- Complete picture of how programs go from keyboard to execution
- Understand translation from high-level language to machine language — help understand inefficiencies
- Learn to build language processors (esp. compilers)
- Learn basic concepts for handling structured text data: abstract syntax; syntax-directed translation

CS 421 — Class 1, 1/17/12 — 5

Why learn about definition, implementation, and use of functional languages?

- Functional programming is the major alternative programming paradigm to imperative/object-oriented programming.
- Functional programming language features are increasingly appearing in mainstream languages
- Functional programming concepts increasingly used in non-functional languages; easiest to learn them in a functional programming language

CS 421 — Class 1, 1/17/12 — 6

Other topics...

- Lambda calculus: The “ur” language — simplest and most powerful functional language
- Program verification: Learning how to reason about programs
- Parallel languages: Coming into use to deal with ubiquitous parallelism

CS 421 — Class 1, 1/17/12 — 7

How this class operates

- Lectures 9:30–10:45 TR in 1404 SC
- Usually weekly assignments, due Tuesday at 9:30am
- Programming mostly in OCaml, a functional language
- Each class’s slides put on web day before class; subset of the slides — the ones containing exercises — printed and distributed in class
- Two midterms and final (dates t.b.a.)
- Course policies — grade calculations, collaboration/cheating policy, lateness — given on web

CS 421 — Class 1, 1/17/12 — 8

Classroom pedagogy

Trying out some new things this semester...

- In-class exercises (ungraded)
- Discussion encouraged when doing exercises
- TAs will be in class to help (if possible)
- Tablet PCs will be used to help me see what you're doing, possibly share with class

Brief overview of programming languages

- First high-level language (FORTRAN) developed c. 1958. Emphasis was on efficiency.
- As computers got faster, and memories grew, languages became less and less about efficiency, more and more about programmer productivity.
- Modern language design involves trade-offs among various issues: efficiency; (short-term) programmer convenience; (long-term) program maintainability; portability; security; parallelizability.

Capsule history of PLs

	<u>Conventional</u>	<u>O-o</u>	<u>Scripting</u>	<u>Functional</u>
1957/8	Fortran			LISP
1960's	Algol	Simula 67		
1970's	C	Smalltalk		ML
1980's		C++		
1990's		Java	Perl	Haskell OCaml
early 2000's		C#	Python Ruby	

Recent languages: Scala, Clojure, F#, Lua, Go, ...

Others?

Programming language features

	Traditional, "static"	Static o-o	Scripting, "dynamic"	Mixed
Examples	C, Fortran	C++	Python, Ruby	Java, OCaml
Objects?				
Automatic mem. mgt.?				
Static types?				
Tagged values?				

Down to details...

- MP0 — “due” Thursday 9:30am
 - Install OCaml, write simple programs (approx. 1 hr. work)
 - Not graded, but uses handin
- MP1 — due Tuesday, 1/24, 9:30am
 - Recursive functions on lists in OCaml
- First month of class is on lexing, parsing, type-checking, and abstract syntax
 - Programming assignments in OCaml
 - Next two classes: intro to OCaml

OCaml

OCaml is a popular functional programming language, which means

- Calculations performed mainly by writing recursive functions on lists and trees
- Automatic memory management
- No *assignable* variables — variables assigned once, never re-assigned; no while loops
- Higher-order functions (but not until later in semester)

Also, OCaml makes it very convenient to define and manipulate *abstract syntax trees*, which are of crucial importance to us in this class.

Interactive use of OCaml

Like Python and Ruby, OCaml can be used by entering expressions and function definitions interactively, in a “read-eval-print” loop:

```
~$ ocaml
# let pi = 3.14159;;
val pi : float = 3.14159
# let circum r = 2.0 *. pi *. r;; (* Use *. for floating pt mult *)
val circum : float -> float = <fun>
# circum 8.0;;
- : float = 50.26544
# let fac x = if x=0 then 1 else x * fac (x-1);;
Error: Unbound value fac
# let rec fac x = if x=0 then 1 else x * fac (x-1);;
val fac : int -> int = <fun>
# fac 4;;
- : int = 24
```

Interactive use of OCaml (cont.)

```
# let rec binom n m = if m>n then 0
                      else if m=0 or m=n then 1
                      else binom (n-1) m + binom (n-1) (m-1);;
val binom : int -> int -> int = <fun>
# binom 4 2;;
- : int = 6
# #use "defs.ml";; (* defs.ml contains defn of fib *)
val fib : int -> int = <fun>
# fib 4;;
- : int = 5
```

- Function application syntax: juxtaposition
- Variable and function names must begin with lower-case letter or underscore; can contain digits or apostrophe.
- No “return” statement: function body is an expression, and its value is returned.

Interactive use of OCaml (cont.)

- `if-then-else` is a **conditional expression**, not a statement, like C's conditional expression (*condition ? expr : expr*).
- `#use` must be used with care: if you load a file, then use its definitions to define some other functions, and then *reload* the original file, the functions you defined in the meantime will still use the old definitions. Simplest thing is to terminate the OCaml session and restart.
- Although no types are given, OCaml is a *statically typed* language, like C or Java, unlike Python or JavaScript:

```
# circum 4;;
Error: This expression has type int but an expression was expected of type
      float
```

CS 421 — Class 1, 1/17/12 — 17

Types in OCaml

- We will not use the class facility of OCaml, but instead will just use the built-in types the language provides, which are very powerful:
 - Primitive types: `int`, `float`, `bool`, `char`, `string`
 - Structured types:
 - Homogeneous lists: τ list, where τ is any type
 - Heterogeneous tuples: $\tau_1 * \tau_2 * \dots * \tau_n$
 - Functions: $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n$
- (There is another way to define new types — different from using classes — that we will use heavily. Will discuss that next week.)

CS 421 — Class 1, 1/17/12 — 18

Primitive types

- Primitive types — `int`, `float`, `bool`, `char`, `string` — are more or less what you would expect:
 - `int`: 3, 74, -853
 - `float`: 3.0, .012
 - `bool`: `true`, `false`
 - `char`: `'a'`, `'\\n'`
 - `string`: `"sam I am"`
- ... with some quirks:
 - Arithmetic operations: usual for ints; add period for floats (`+. , *. , etc.`)

CS 421 — Class 1, 1/17/12 — 19

Primitive types (cont.)

- Comparison operations: `=`, `<`, `>`, `<>`, `<=`, `>=`
- Boolean operations: `or` or `||`, `&` or `&&`, `not`
- String operations: `^` (concatenation); `s.[i]` (subscript); use `String` module ("open String") for many other operations
- No automatic type conversions; use `string_of_int`, `int_of_string`, `float_of_int`, etc.

CS 421 — Class 1, 1/17/12 — 20

Tuples

- Create “struct” by putting expressions in parentheses separated by commas:
 - `(3, 5.0): int * float`
 - `(3, "abc", true): int * string * bool`
- Exercise — fill in types:
 - `('a', 'b') :`
 - `('a', "a", 'a') :`
 - `(5, ("a", 'a')) :`
- Exercise — create a value of the given type:
 - `_____ : int * (int * float) * string`

Tuples (cont.)

- Use functions `fst` and `snd` to get elements of a pair. *Only works for pairs.* (We'll see how to deal with bigger tuples in next class.)
- Exercise: Write a function to add the elements of an `int * int` pair:
 - `let addelts p =`

Lists

- Create a list by putting expressions, all of same type, in square brackets separated by semicolons:
 - `[3]: int list`
 - `[true; false; true]: bool list`
 - `[['a']; ['b'; 'c'; 'd']; []]: (char list) list`
 - `[3; 4.0]: type error`
 - `[3; [4]]: type error`

Lists (cont.)

- Exercise — fill in types (or flag error):
 - `['a'; 'b'] :`
 - `['a'; 'b'; "c"] :`
 - `[4; int_of_string "34"] :`
 - `[[4]; [5]; []] :`
 - `[(1, 2); (3, 4)] :`
 - `[(1, 2); (3, 4, 5)] :`
 - `[(1, [3]); (4, [5; 6])] :`

Lists (cont.)

- Exercise — create a value of the given type (other than the empty list):

- `int list`
- `(int list) list`
- `(int * string) list`
- `(string list) list`
- `(int * string list) list`
- `((int * string list) list) list`

Lists (cont.)

- Use infix `@` to concatenate lists, infix `::` to “cons” to start of list; load `List` module (“`import List;;`”) to get functions `hd`, `tl`, `nth`, `length`, and others:

```
# open List;;
# let lis1 = [1; 2; 3];;
val lis1 : int list = [1; 2; 3]
# hd lis1;;
- : int = 1
# tl lis1;;
- : int list = [2; 3]
# 4 :: lis1;; (* N.B. non-destructive add *)
- : int list = [4; 1; 2; 3]
# length lis1;;
- : int = 3
# nth lis1 2;;
- : int = 3
```

Lists (cont.)

- Exercise: Write a function to compute the sum of the first two elements of an `int list`: `addfirsttwo [5; 3; 2; 6] = 8`. You can assume the list is of length at least 2:

- `let addfirsttwo lis =`

- Exercise: Write a function to compute the sum of the *lengths* of the first two elements of an `(int list) list`: `addfirsttwolengths [[5; 3]; [2]; [6; 2; 5; 3]] = 3`. You can assume the list is of length at least 2:

- `let addfirsttwolengths lis =`

Polymorphic functions

- OCaml detects the types of functions, so that variable and function types don’t have to be declared.
- However, for some functions, there is no way to determine a *specific* type. An example is this function that takes a pair and returns it in reversed order:

```
let revpair p = (snd p, fst p)
```

- You can write `revpair(3,4.0)` — so `revpair` has type `int * float → float * int` — but you can also write `revpair("abc", true)` — so `revpair` has type `string * bool → bool * string`. In fact, for any types τ and τ' ,

$$\text{revpair} : \tau * \tau' \rightarrow \tau' * \tau.$$

Polymorphic functions (cont.)

- OCaml realizes this and assigns `revpair` a *polymorphic* type — basically, a type with “type variables” in it. Specifically, here is how OCaml responds to `revpair`:

```
# let revpair p = (snd p, fst p);;  
val revpair : 'a * 'b -> 'b * 'a = <fun>
```

Read this type as “ $\alpha * \beta \rightarrow \beta * \alpha$.”

- Similarly, a function may operate on lists of any type. Thus, `length` has type:

$\alpha \text{ list} \rightarrow \text{int}$.

Polymorphic functions (cont.)

Ex: Write the polymorphic types of the following functions. (You can write either `'a`, `'b`, etc. or α , β , as you prefer.)

```
let mktriple p = (fst p, snd p, 3)
```

```
let pair_with_int lis = (hd lis, 4)
```

```
let double_first lis = [hd lis; hd lis]
```

```
let pair2list p = [fst p; snd p]
```

```
addfirsttwolengths (defined above)
```