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**

Course staff

Professor:	Sam Kamin kamin@illinois.edu
TAs:	Susannah Johnson sjohnsn2@illinois.edu
	Ted Pacyga tpacyg2@illinois.edu
	Rohan Sharma sharma27@illinois.edu

- **Web: courses.engr.illinois.edu/cs421/sp2013/**
- **Piazza: piazza.com/illinois/spring2013/cs421**

What you will learn this semester

- How to implement programming languages
  - Writing lexical analyzers and parsers
  - Translating programs to machine language
  - Implementing run-time systems
- How to write programs in a functional programming language
- How to formally define languages (including the definitions of type rules and of program execution)
- Key differences between statically-typed languages (e.g. C, Java) and dynamically-typed languages (Python, JavaScript)
- Plus a few other things...

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**

Why learn about definition and implementation of conventional 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

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, and especially in scripting languages; easiest to learn them in a functional programming language

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**

Classroom pedagogy

- **In-class exercises (ungraded) - These are smaller versions of what you'll be doing on homeworks and exams.**
- **Discussion encouraged when doing exercises**
- **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

	Conventional	O-o	Scripting	Functional
1957/8	Fortran			LISP
1960's	Algol	Simula 67		
1970's	C	Smalltalk		ML
1980's		C++, Obj. C		
1990's		Java	Perl	Haskell
			JavaScript	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, JS	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, 1/22, 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: justaposition**
- **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.**
- **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
```

Types in OCaml

- **OCaml provides powerful built-in types:**
  - **Primitive types: int, float, bool, char, string**
  - **Structured types:**
    - **Homogeneous lists:** τ **list, where** τ **is any type**
    - **Heterogeneous tuples:** τ_1 `*` τ_2 `*` $\cdots$ `*` τ_n
    - **Functions:** τ_1 `->` τ_2 `->` $\cdots$ `->` τ_n
- **The class facility allows the definition of new types, as in Java. However, we will not use classes. There is another way to define new types — using the `type` facility — that we will use heavily. We will discuss that next week.**

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 one quirk: arithmetic operations for floats use a period (`+.`, `*.`, etc.)**

Primitive types (cont.)

- Comparison operations: =, <, >, <>, <=, >=
- Boolean operations: or or ||, & or &&, not
- String operations: ^ (concatenation); s.[i] (subscript)
- No automatic type conversions; use string_of_int, int_of_string, float_of_int, etc.
- Various "modules" provide additional operations; these are loaded by entering open *module-name*;;.
  - String and Str modules have operations on strings.
  - Pervasives is a module that is always loaded — you don't need to open it.
  - Index of modules is linked at the bottom of the online manual — see "resources" tab in course web page.

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 =`

```
addelts (3,4);;  (* returns 7 *)
```

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 `@` to concatenate lists, `::` to "cons" to start of list; load List module 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 cons *)
- : int list = [4; 1; 2; 3]
# [4] @ lis1;;   (* N.B. non-destructive append *)
- : 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.**
- **But consider:**

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

- **It is legal to write**
  - `revpair (3, 4.0)`
  - `revpair ("abc", true)`
  - ***etc.***
- **In fact, for any types τ and τ',**

$$\texttt{revpair:}\ \tau\ \texttt{*}\ \tau' \rightarrow \tau'\ \texttt{*}\ \tau$$

Polymorphic functions (cont.)

- **OCaml realizes this and assigns `revpair` a *polymorphic* type — a type with "type variables" in it:**

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

Read this as "`revpair` has type α * $\beta \rightarrow \beta$ * α."

- **Similarly, a function may operate on lists of any type. Thus, `length` has type α list $\rightarrow$ 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_of_first p = (fst p, fst p)
```

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

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

```
addfirsttwolengths (defined above)
```

Wrap-up

- **Today we discussed:**
  - **What CS421 is about**
  - **Basic OCaml programming and use of the top level**
- **We discussed it because:**
  - **It's good to know why you're learning this stuff**
  - **We'll be using OCaml for MPs this semester. And one goal of the class is for you to learn functional programming.**
- **In the next class, we will:**
  - **Talk more about OCaml — esp. writing functions on lists.**
- **What to do now:**
  - ***MP0* — not graded, but using OCaml a little will give you a much better feel for it than listening to me talk about it.**