CSE 120/130

Introduction to

Programming Languages and Techniques

Fall 1999

Handout 1

Instructors:	Peter Buneman	562 Moore	MW 12–1
	Lyle Ungar	560 Moore	TW 1–2
	Seth Kulick (CSE130)	126 Moore	M 2–3 and W 2:30–3:30

Additional Personnel and Information

Teaching Assistants:	Matthew Beitler
	Aaron Bloomfield
	Vladimer Gapeyev
	Alwyn Goodloe
	Gang Huang
	Pankaj Kakker
	Xiaoyi Ma
	Jonathan Vitale
	John Wen
	Dianna Xu
Administrative Assistant:	Christine Metz, 556 Moore
Webpage:	`http://www.seas.upenn.edu/~cse120`
Newsgroup:	`upenn.cis.cse120`
Course Email:	`cse120@seas.upenn.edu`

Basic Administrative Stuff

This course is required of all majors in Computer Science.

There is a lab associated with the course, CSE130. If you are enrolled in this course you must also be enrolled in a section of 130.

You will receive 1.5 credits for the course and lab (same grade for both).

Grading

The required work for the course will consist of the following:

- Written homeworks and programming exercises, in conjunction with the lab
- Examinations (2 midterms and a final exam)

Final course grades will be computed as follows:

- The final exam counts for 35% of the course grade
- The average grade of the homeworks counts for 25% of the grade
- Each midterm counts for 20% of the grade.
- However: you must have a passing grade on the homework portion to pass the course

Late Policy for Homeworks: Late assignments will be penalized 25%, and will not be accepted after 10am of the day following the due date.

The following activities are not graded but will have a strong bearing on your performance on the homeworks, programming exercises, and exams:

- Attending and participating in lectures and labs
- Reading the handouts, course newsgroup, and on-line materials

Exams

1. **First mid-term:** Wednesday, October 11, in class.
2. **Second mid-term:** Wednesday, November 8, in class.
3. **Final:** Monday, December 18, 4-6PM

Additional administrative information will be provided during the semester. Keep an eye on the course web page and the newsgroup.

Discussion vs. cheating

You are free — in fact, encouraged — to discuss the material in this course, including assignments, with the instructors, TAs, and your fellow students.

However, the work you hand in must be your own. Copying assignments (or cheating of any other nature) carries serious penalties, in accordance with University procedures.

Advertisement

”Connections – Computing Resources at Engineering”
Orientation for Engineering Students

The ENIAC Museum - Moore Building

Friday	Sept. 8	2 pm
Tuesday	Sept. 12	12 noon
Thursday	Sept. 14	4 pm

Textbooks

There is no textbook for the first part of the course, however we are developing a short "primer" as an introduction to the programming language and topics for this part of the course. Textbooks for the second part of the course will be announced shortly.

1. Arnold and Gosling *The Java Programming Language* Second (1998) or Third (2000) Edition, Addison-Wesley, (required).
2. Savitch *Java: An Introduction to Computer Science and Programming*, 1998/9

What this course is like.

You do *not* need to have any programming experience to take this course. In fact, the first language we use will be new to almost everybody, and is sufficiently different from "standard" programming languages (C, C++, Java) that people who know one of these languages are not at any great advantage.

However, some computer experience helps. Knowing about files, editors, directories etc. is useful in the first stages of the course. It does not take long to learn this, but you need to do it almost on day one of the course. The labs will help you get started in the computing environment provided for this course.

The course is introductory but fast-paced. Don't get left behind!

What this course is about

CSE120 and CSE121 are rather different from most introductions to Computer Science (which start off rather slowly with low-level programming).

It has two purposes:

- **General Introduction to Computer Science**. Computer Science is much, much more than writing programs. It is one of the great intellectual advances of this century.
- **Basic Techniques of Programming**. Recursion and iteration, fundamental data structures (lists, trees, etc.), value-oriented and object-oriented programming styles, types, evaluation, algorithmic complexity, etc.

CSE 121 continues both of these topics in greater depth.

Some “Big Ideas” in Computer Science

Computer Science has generated remarkable ideas in its half-century history. We will touch on several in CSE120.

Computability
When Eniac was built 55 years ago very few people understood what it meant to compute something. Even less understood was the idea that some problems cannot be “computed” at all.

Efficiency
Almost every interaction with a computer involves a non-trivial program. How does Alta Vista or Yahoo search 100 million web pages in a fraction of a second to find those that contain the pages that contain your search term?

Some problems are believed to have *no* efficient solution. Ironically, this fact can be used to positive effect, for example in cryptography: all known approaches to “cracking” a strong cryptosystem take too long on even the fastest of computers.

Representation
How we represent a problem or domain is crucial to our ability to understand and to program with it.

Databases were revolutionized with the invention of relational databases (and they may undergo another revolution following the recent development of XML).

Abstraction
Transistors “abstract away” from the details of electrophysics. Microprocessor architectures abstract away from the details of transistors. Operating systems abstract away from the details of microprocessor architectures. Application libraries abstract away from the details of operating systems. User programs abstract away from the details of application libraries...

Abstraction is a constant activity in programming. The program you write to sort a sequence of integers should work equally well to sort a sequence of bank account transactions. How do we abstract from the specific to the general?

Programming Languages Matter

A **programming language** is a precise notation for expressing instructions to a computer.

A programming language is *also* a medium for communicating ideas between human beings.

A well-designed programming language makes both these tasks easier, leading to programs that are more robust, faster, and easier to understand.

Programming Languages Don't Matter

A programming language is a "power tool": a fair amount of detailed knowledge is usually required to use it well.

However, it's important not to become too attached to any particular language. Imagine a carpenter that knew how to use a jigsaw, but not a circular saw. Or a composer that could only write for alto clarinet...

Languages come and go, while the fundamental *concepts* of programming remain relatively stable.

ML and Java

We will work with two languages in this course:

ML is a powerful and elegant language, emphasizing a "value-oriented" style of computation. It is only beginning to catch on outside of academia: you are unlikely to use it in your summer job next year. However, it provides an ideal setting for discussing many fundamental ideas in computer science, allowing us to get straight to the meat of the subject at the beginning of the course.

Java is a mainstream "object-oriented" language. It owes much of its current visibility to its use in distributing tiny applications ("applets") across the World Wide Web, but it is gaining popularity in many other areas of software development.

Getting Started – Syntax and Semantics

Like natural languages, computer languages have both *syntax* (rules governing which strings of text are legal programs) and *semantics* (rules specifying what those programs do or mean).

We are quite used to dealing with the fact that different pieces of text (syntax) can have the same meaning (semantics):

- 34
- thirty four
- trente-quatre
- XXXIV
- 16 + 18
- XVI + XVIII (yes, the Romans wrote this)
- Ox22 (hexadecimal)

Computing with Expressions

Every program in Objective Caml is an *expression*. The "meaning" of the program is the value of the expression.

The simplest OCaml programs express calculations with numbers and other basic forms of data, using a syntax resembling familiar mathematical notation:

```
# 16 + 18;;
- : int = 34

# 2*8 + 3*6;;
- : int = 34
```

The basic mode of interacting with OCaml is typing in a series of expressions; OCaml *evaluates* them as they are typed and displays the results. In the interaction above, lines beginning with `#` are inputs and lines beginning with `-` are the system's responses Note that inputs are always terminated by a double semicolon.

The annotation `int` in the responses is the *type* (i.e. integer) of the results.

In this course, we concentrate on how we express computations and not on how particular computers perform, say, addition. How computers perform addition, and more generally how they are organized as engineered machines/devices is the topic of an important field, Computer Architecture, studied extensively in other courses. One of the goals of the present course is to introduce programming in a machine-independent fashion, as much as possible. As software became more complex, being able to understand computation in terms more abstract than those of the computer architecture became essential.

The idea of using mathematical notation to specify computation was an essential step in programming languages, going back to FORTRAN (FORmula TRANslator) 1954.

Expressions have a basic uniform structure: functions applied to arguments. As in usual mathematical practice, ML uses *infix* notation for operations such as addition and multiplication. For other sorts of functions (e.g. $\min$, $\log$), it uses a *prefix* notation similar to the standard notation from mathematics.

$\sqrt{2}$	`sqrt 2`
$6!$	`fact 6`
$3+4$	`3+4`
$3+4\cdot 5$	`3+4*5`
$\min(3,4)$	`min 3 4`

(In algebra, the dot for multiplication is often omitted. This is not allowed in OCaml.)

Expressions are composed of sub-expressions. The values of the sub-expressions are used in computing the value of the expression.

Sub-expressions may be parenthesized to indicate the order in which operations are applied. When parentheses are omitted, the *precedence* of the operations determines the grouping of sub-expressions.

```
# (2 * 8) + (3 * 6);;
- : int = 34
# 2 * 8 + 3 * 6;;
- : int = 34
# 2 * (8 + 3) * 6;;
- : int = 132
# ((2 * 8) + 3) * 6;;
- : int = 114
```


Other kinds of expressions (preview)

String expressions (`^` is string concatenation):

```
# "cats" ^ " and " ^ "dogs";;
- : string = "cats and dogs"
```

List expressions (`List.rev` is list reverse and `@` is list concatenation):

```
#  List.rev [1; 2; 3];;
- : int list = [3; 2; 1]

# [1; 2; 3; 4; 5] @ (List.rev [9; 8; 7; 6]);;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Mixed expressions (the result is a boolean):

```
# 2200 > 7 * 314;;
- : bool = true
```

In fact, ML carries the idea of computing with expressions much further than languages like FORTRAN, Pascal, C/C++ or Java.

Giving things names

It is convenient to name values, both so that we can remember them

```
# let inchesPerMile = 12*3*1760;;
val inchesPerMile : int = 63360
```

and for efficiency:

```
# let a = 8 * 16 * 127;;
val a : int = 16256

# a * a + 2 * a;;
- : int = 264290048
```

(If we had performed the equivalent computation

```
# 8*16*127 * 8*16*127 + 2 * 8*16*127;;
```

we would have used eight multiplications rather than four.)

In general, a `let` declaration gives a name to the value that results from an expression. Such names are also called *variables* or *identifiers*.

Warning: as opposed to Pascal or C, the value associated with a variable in ML cannot be changed later through "assignment." We say that ML is (predominantly) a *functional* or *value-oriented* language while Pascal and C are *imperative* languages.

Functions

A calculation that is to be repeated many times on different data can be captured by a single function declaration:

```
# let cube (x:int) = x*x*x;;
val cube : int -> int = <fun>

# cube 9;;
- : int = 729
```

We call `x` the *parameter* of the function `cube`; the expression `x*x*x` is its *body*.

The expression `cube 9` is an *application* of `cube` to the *argument* `9`.

The *type* printed by OCaml, `int->int` (pronounced "`int` arrow `int`") indicates that `cube` is a function that should be applied to a single, integer argument and whose result is an integer.

Note that OCaml responds to a function declaration by printing just `<fun>` as the function's "value."

Here is a function with two parameters:

```
# let sumsq (x:int) (y:int)  = x*x + y*y;;
val sumsq : int -> int -> int = <fun>

# sumsq 3 4;;
- : int = 25
```

The type printed for `sumsq` is `int->int->int`, indicating that it should be applied to two integer arguments and yields an integer as its result.

Note that the syntax for invoking function declarations in OCaml is slightly different from languages in the C/C++/Java family: we write `cube 3` and `sumsq 3 4` rather than `cube(3)` and `sumsq(3,4)`.

Function Evaluation

It is easy to understand how applications of functions are evaluated. Take our example:

```
# let cube (x:int) = x*x*x;;
```

To evaluate `cube 9`, we substitute `9` for `x` in the body of `cube` to get `9*9*9`. This is then evaluated as usual to get the answer, `729`.

How do we evaluate `cube(cube(2))`?

Functions with Conditions

```
# let abs(x:int) = if x>=0 then x else -x;;
val abs : int -> int = <fun>

# abs (5-3);;
- : int = 2

# abs (3-4);;
- : int = 1
```

The meaning of this function is clear — it computes the absolute value of an integer.

The whole body of `abs` (i.e., `if x>=0 then x else -x`) is an expression. It has three sub-expressions: `x>=0`, `x`, and `-x`.

The second and third of these are integer expressions. The first is a *boolean* expression, whose value can either be `true` or `false`, depending on what argument we substitute for the parameter `x`.

The type boolean

There are only two values of type `boolean`, `true` and `false`. Comparison operations return boolean values:

```
# 1 = 2;;
- : bool = false

# 4 >= 3;;
- : bool = true

# not (2 = 2);;
- : bool = false
```

`not` is a unary operation on booleans.


```
# 2 = 3 || 4 > 3;;
- : bool = true

# 2 = 3 && 4 > 3;;
- : bool = false

# if 3 = 4 then 17 else 64;;
- : int = 64
```

The operator `||` means `or` and `&&` means `and`.

The conditional `if B then E1 else E2` is an *expression* and has a result which is either the result of `E1` or that of `E2`, depending on whether the result of `B` is `true` or `false`.

A conditional expression can be a sub-expression of another expression:

```
# if 3 = 4 then 17 else if 5 = 5 then 64 else 77;;
- : int = 64

# 2 * (if 5 = 5 then 64 else 77);;
- : int = 128
```


Another example

```
# let ord3 (x:int) (y:int) (z:int) = x < y && y < z;;
val ord3 : int -> int -> int -> bool = <fun>

# ord3 1 2 3;;
- : bool = true

# ord3 2 1 3;;
- : bool = false
```

Defining things inductively

In mathematics, we often define things inductively by giving a "base" case and an "inductive step". For example, the sum of all integers from 0 to n or the product of all integers from 1 to n (the *factorial*):

$$\begin{aligned} \text{sum}(0) &= 0 \\ \text{sum}(n) &= n + \text{sum}(n-1) \quad \text{if } n \geq 1 \end{aligned}$$

$$\begin{aligned} \text{fact}(1) &= 1 \\ \text{fact}(n) &= n * \text{fact}(n-1) \quad \text{if } n \geq 2 \end{aligned}$$

It is customary to extend the factorial to all non-negative integers by adopting the convention $\text{fact}(0) = 1$.

Recursive functions

We can translate inductive definitions directly into *recursive* functions. We use `rec` after the `let` to indicate that this is a recursive function — one that can "call" itself in its own body.

```
# let rec sum(n:int) = if n = 0 then 0 else n + sum(n-1);;
val sum : int -> int = <fun>

# sum(6);;
- : int = 21

# let rec fact(n:int) = if n = 0 then 1 else n * fact(n-1);;
val fact : int -> int = <fun>

# fact(6);;
- : int = 720
```

(Note that these definitions will only "work" for non-negative integers.)

Understanding Recursive Functions

We can use substitution to explain how a recursive function gets evaluated:

```
fact(3)
if 3 = 0 then 1 else 3 * fact(3-1)
3 * fact(2)
3 * ( if 2 = 0 then 1 else 2 * fact(2-1) )
3 * 2 * fact(1)
3 * 2 * ( if 1 = 0 then 1 else 1 * fact(1-1) )
3 * 2 * 1 * fact (0)
3 * 2 * 1 * ( if 0 = 0 then 1 else 1 * fact(0-1) )
3 * 2 * 1 * 1
6
```


We have intentionally glossed over some of the details of how and when things get evaluated (these will come later) but it is important to note that we can explain the computation through manipulation of syntax and that the meaning of the program can be explained without reference to any particular compiler or machine.

Not all recursive definitions "work"

Consider the mathematical definition:

$$\begin{aligned} \text{stupid}(n) &= 0 && \text{if } n = 0 \\ \text{stupid}(n) &= n + \text{stupid}(n+1) && \text{otherwise} \end{aligned}$$

and the corresponding program

```
# let rec stupid(n:int) = if n = 0 then 0 else n + stupid(n+1);;
val stupid : int -> int = <fun>

# stupid 10;;
Stack overflow during evaluation (looping recursion?).
```

A mathematician would say that the function is "ill-defined" for positive integers; a computer scientist would say that it would "not terminate" or "loop forever". In fact, OCaml doesn't even loop forever. It can handle a large but limited number of recursive calls. Certainly our evaluation-by-substitution process would not terminate.

Calculating n^m

```
# (* Computes n to the mth power. Assumes m >=0 *)
  let rec power (n:int) (m:int) =
  if m = 0 then 1    (* Test for m = 0 *)
  else n * power n (m-1);;
```

The text between `(*` and `*)` is a comment and is ignored by the compiler. However it will *not* be ignored by readers of your program. Good comments are essential.

The comment by the function header is an important and necessary comment. (Why?) The second comment is pointless.

Euclid's Algorithm

The Greatest Common Divisor (GCD) of two positive integers is the largest integer that divides both. For example, $\gcd(9, 12) = 3$ Here are three useful facts about gcd:

$$\begin{aligned} \gcd(x, x) &= x \\ \gcd(x, y) &= \gcd(y, x) \\ \gcd(x, y) &= \gcd(x - y, y) \quad \text{if } x > y \end{aligned}$$

We can translate this into an ML algorithm. Note the use of OCaml syntax for defining and applying two-argument functions.

```
# (* Compute the GCD of two numbers *)
  let rec gcd (x:int) (y:int) =
    if x = y then y
    else if x > y then gcd (x-y) y
    else gcd x (y-x);;
```

It's easy to see that *if* the program terminates, it terminates with a correct answer. But how do we know it terminates for all arguments?

N.b.: This is not precisely the form in which Euclid presented this algorithm. (Euclid's version used the *modulo* operation, which OCaml provides but which we haven't seen yet.)

Choosing n things from m

How many ways can we choose a set of n things from a given set of m things. Call this number $C(n, m)$. Example: a selection of 10 pizza toppings is offered and you can choose three different toppings for the standard charge. The number of ways of doing this is $C(3, 10)$.

How do we compute $C(n, m)$? Consider one of the set of m objects – say anchovies – the number of choices that contain anchovies is $C(n-1, m-1)$. The number of choices that do not contain anchovies is $C(n, m-1)$ So we have the equation

$$C(n, m) = C(n-1, m-1) + C(n, m-1).$$

We also need to know that $C(m, m) = 1$ and that $C(0, m) = 1$.

Here is the inductive definition translated into OCaml. Again note the use of the OCaml syntax `choose n m` for the mathematical notation $C(n, m)$.

```
# (* choose n things from m *)
  let rec choose (n:int) (m:int) =
    if n = m || n = 0 then 1
    else choose (n-1) (m-1) + choose n (m-1);;

# choose 3 10;;
- : int = 120
```

As for Euclid's algorithm, an important part of understanding this program is seeing why it terminates.

Making Change

Here is another interesting use of recursion on integer arguments. Suppose you are a bank and therefore have an "infinite" supply of money, and you have to give a customer a certain sum. How many ways are there of doing this?

If you only have pennies, there is only one way of doing this: pay the whole sum in pennies.

```
# (* No. of ways of paying s in pennies *)
  let rec changeP (s:int) = 1;;
```

That wasn't too hard!

Making Change – continued

Now suppose the bank has both nickels and pennies. If `s` is less than 5 then we can only pay with pennies. If not, we can do one of two things:

- Pay in pennies; we already know how to do this.
- Pay with at least one nickel. The number of ways of doing this is the number of ways of making change (with nickels and pennies) for `s`-5.

```
# (* No. of ways of paying in pennies and nickels *)
  let rec changePN (a:int) =
    if a < 5 then changeP a
    else changeP a + changePN (a-5);;
```

Making Change – continued

Continuing the idea for dimes and quarters:

```
# (* ... pennies, nickels, dimes *)
  let rec changePND (a:int) =
    if a < 10 then changePN a
    else changePN a + changePND (a-10);;

# (* ... pennies, nickels, dimes, quarters *)
  let rec changePNDQ (a:int) =
    if a < 25 then changePND a
    else changePND a + changePNDQ (a-25);;
```


Finally:

```
# (* Pennies, nickels, dimes, quarters, dollars *)
  let rec changePNDQS (a:int) =
    if a < 100 then changePNDQ a
    else changePNDQ a + changePNDQS (a-100);;
```

Some tests:

```
# changePNDQS 5;;
- : int = 2

# changePNDQS 9;;
- : int = 2

# changePNDQS 10;;
- : int = 4
```

```
# changePNDQS 29;;
- : int = 13

# changePNDQS 30;;
- : int = 18

# changePNDQS 100;;
- : int = 243

# changePNDQS 499;;
- : int = 33995
```


Summary of concepts in this handout

- Syntax and semantics
- Integer expressions
- Giving names to values
- Boolean expressions
- Defining functions
- Recursive functions