DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CSL 101 ; Introduction to Computer Science
Semester II 2007-08
Brief Lecture Notes Instructor **Sandeep Sen**

1 The notion of computing

In our everyday life, we are involved in numerous activities that require numerical calculation, starting from our grocery bills, keeping cricket scores to handling budget. It is not unusual to find shopkeepers punching in numbers to produce bills for customers. In bigger stores, life is made simpler where there are check-out machines where one has to only punch in the item codes (rather than the item prices) for producing the bills, thus reducing the chances of error in punching the prices. This also provides additional flexibility to the shopowners who can modify the prices of items very easily by changing the "table of prices." The person at the check-out counter doesn't have to remember the prices and he/she can punch the same keys for the same items. But this requires for sophisticated computing machines than calculators.

1.1 An abstract model of computer

A machine that executes instructions faithfully and very quickly. There are differences between computers with regards to speed and instructions but the *final answer* will be same. The computers are all more or less equally capable in their ability to solve problems and *possibly there is nothing better* (Church-Turing thesis).

1.2 What is a programming language

A formal language that contains a set of instructions that will produce the same (modulo some numerical precision) answers even if executed in diferent machines. Therefore, we need not bother about the internals of a computer when we write a program.

All programming languages are again equally capable of solving problems, namely that they have rich enough instruction set. (It is actually a very small set by which we can carry out all computations.)

1.3 Can you compute everything

A very deep philosophical question that was answered in the negative by a German Mathematical called Goedel, who *proved* that one cannot compute everything. This was done in setting of proving theorems mechanically in an axiomatic system. Fortunately not many naturally occuring problems fall in the non-computatble category.

Another paradox is that all this was done even before the first real computer was built. So the notion of computation had existed much before computers were built.

1.4 What is interesting computation

A fixed sequence of instruction, for example, how to prepare cake is not particularly interesting since it has nothing unpredictable about it. Or say, how to go from Shivalik hostel to Kailash hostel is equally boring (perhaps not !) once someone has figured it out. However, given a campus map, how do we get from point A to point B is a more challenging exercise.

If the number of source-destinations is fixed then again you can do a *one-time* computation and store the routes. Just look up the table (as you do in case of a railway time table).

The problem becomes more interesting, if you are not told in advance which map you will be searching, i.e. the map is also now part of the input.

1.5 An algorithm

Before you write a program, which is ready to be fed into a computer, we must *reason out* a strategy to solve the problem. The strategy should be

- Finite (we can't have infinite instructions)
- Correct for **all possible inputs**
- Terminate (in finite time, infact the sooner the better)

The last two properties must be formally argued/proved before we translate it into a program. In fact the more time we spend on this, the chances of a successful program is higher. Any strategy that satisfies the above properties is called an *Algorithm* for a given problem (and there ican be more than one). With experience, the process of translating an algorithm into a program becomes more mechanical. One may ask now, how do we describe an algorithm ? It is usually done using less formal means although one can claim that a program itself describes an algorithm. There are many messy details that are usually left out in describing an algorithm.

1.6 Goodness of a program

Usually one argues at the level of the algorithm regarding use of resources like execution time and memory consumption. Scaling is often used to avoid absolute measurements in terms of a specific computer.

How many elementary operations do we do to multiply two 100 digit numbers ? How much paper do we consume ?

2 Counting and comparing infinite sets

Some of the most innovative proofs in mathematics are based on counting arguments. These involve infinite sets (finite sets are not interesting in this context). One of the most basic questions that come up in this context are

Are there more *real* numbers than *integers* ?

The intuitive answer is yes based upon an equally intuitive reasoning that between any two integers there are an infinite number of real numbers. But is this a sound basis for comparing two sets, namely $\mathbb{Z}$ and $\mathbb{R}$? By the same argument, that there are more rational numbers than integers. How about an even simpler question about the number of *even* integers and integers ? For the last question, you may say that since *even* integers is a proper subset of $\mathbb{Z}$, clearly ...

Unfortunately, the intuitive arguments are not adequate when we are looking at infinite sets. Without being too formal, we use the following test when we compare two infinite sets A and B -

Two sets A and B are *equinumerous* iff we can find a 1-1 mapping between elements of A and B

The map $f(i) = 2i$ is a 1-1 mapping from integers to even integers and hence they are *equinumerous*. With somewhat more effort we can define a 1-1 mapping between $\mathbb{Z} \times \mathbb{Z}$ and $\mathbb{Z}$ in the following way.
Consider an infinite two dimensional table $T = \mathbb{Z} \times \mathbb{Z}$ where the i, j-th entry is the tuple $((i, j)$. We can imagine scanning the table along the successive diagonals (top right to bottom left) - thereby getting the sequence $(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1) \ldots$. Notice that we are listing out the numbers in a way such that the sum of $i + j$ of the tuples i, j are $2, 3, 4, \ldots$. Therefore, you can predict the kth tuple of this sequence as also the k corresponding to a tuple (i, j). In other words with a little thought you can define a 1-1 mapping between the sequence of integers and the tuples, thereby establishing that these sets are equinumerous.

Since the number of rational numbers are no more than the number of elements in $\mathbb{Z} \times \mathbb{Z}$ (each rational may have many representations like $1/2 = 2/4 = 3/6$ etc.) and certainly no less than the number of integers, they are actually equinumerous.

Can we do the same for reals and integers ? It turns out that one can argue formally (using Cantor's diagonalization method) that no such mapping exists and hence the number of reals (far) exceed the number of integers. A very celebrated result due to Bertrand Russel is that the power set of a set and the set are not equinumerous (clearly so for finite sets and perhaps less clear for infinite sets).

2.1 Relevance to programming and computers

Why are we discussing seemingly (nay interesting) abstract result of set theory ? It should be clear that we can represent only integers inside the computer, viz. by a finite sequence of symbols. In many cases, the input to a program is an unspecified number of integers, say, we want to sort a sequence of arbitrary integers. This input is a subset of $\mathbb{Z}^k = \mathbb{Z} \times \mathbb{Z} \times \ldots$ k times$\mathbb{Z}$. where k is an arbitrary integer.

We can extend the previous arguments to show that $\mathbb{Z}^k$ is equinumerous with $\mathbb{Z}$ and therefore we can justify writing programs where the input is an arbitrary (not prespecified) set of integers.

3 Types and representation in a programming language

The most basic *type* of elements that must be handled in any programming language is a sequence of alpha-numeric characters called a **string**. This is what we write - however depending on the context a string can be interpreted as an integer, floating point or actually a string itself.

3.1 Integer Types

Given a string $n_k n_{k-1} \ldots n_2 n_1 n_0$, the integer in base b ($0 \leq n_i < b$), is the number $\sum_{i=0}^{k} n_i b^i$. Whereas we are used to base $b = 10$, the internal hardware representation within a computer is base 2. The number of symbols in base 2 are 0 and 1 which corresponds to two states (high and low) in the electronic circuits which is called a *bit*. A sequence of eight bits is a *byte*. The number of distinct states of k bits is 2^k and hence it can be used to represent 2^k integers. A computer has a fixed number of bits to represent an integer which is called a *word* - this may differ from one computer to another. Usually the word length is a power of 2. If the computation involves numbers larger than the maximum representable integer then it leads to an error called *overflow*.

In summary the integer type in any computer represents a finite set of consecutive integers from the (infinite) set of integers.

Exercise 1 *For any base b, and an integer b, show that the string representation is unique.*

Exercise 2 *How would you represent both positive and negative integers ?*

Exercise 3 *The maximum positive integer in ocaml is represented by* max_int. *Write a OCAML program that computes the square of the value of max_int.*

3.2 The floating point numbers

The floating point numbers have two attributes - *mantissa* and the *exponent*. Both are binary representations where the mantissa $m_k m_{k-1} \ldots m_0$ represents $0.m_k m_{k-1} \ldots m_0$ in binary where $m_k \neq 0$ (which makes te representation unique). If E is the exponent then the number represented is $0.m_k m_{k-1} \ldots m_0 \times 2^E$. The reader can verify that it allows us to represent very large numbers compared to integers if we use one word to represent a mantissa as well as an exponent.

Example
Suppose the word length is 32 bits. Then the largest integer is approximately 2^{32}.
The largest floating point is $0.111111 \ldots \times 2^{2^{32}}$ which is much larger.

However, the total number of representable floating points is less than 2^{32+32} which is again finite (compared to the real numbers in the same range). Therefore there are many (infinite) number of reals that we cannot represent in the same range. At best we can approximate a real number by a floating point and this is a non-trivial problem when we are computing with real numbers.

Exercise 4 *How are the floating points distributed in the range of the real of the real numbers ?*

3.3 A note on OCAML integers

The OCAML integers are represented *modulo* 2^w bits where w is the word size. Therefore there is no overflow in integer arithmetic - however there will be errors if the numbers exceed max_int that the programmer has to be aware of. In modulo m arithmetic $-k, 1 \leq k \leq m-1$ is the same as $m-k$. Therefore 0 has only one representation since 0 = -0. As integers can be both negative and positive, the numbers with the MSB (Most significant bit as 1) are interpreted as negative number. For $m = 2^w$, we represent the negative numbers $--x$, with $1 \leq x \leq 2^{w-1}$ as $2^w - x$ and one can verify that the MSB is 1 [1]. The positive numbers represented are $1 \ldots 2^{w-1} - 1$ (and 0). This is often referred to as *2's complement representation. By representing negative numbers, we can actually only add numbers to implement subtraction.*

> Example
> Suppose the word length is 32 bits[2]. Then $-2^{31} = (2^{32} - 2^{31}) \mod 2^{32} = 2^{31}$. On the other hand $+2^{31}$ cannot be represented in 32 bits as MSB is 1. So we can represent one more negative number compared to the positive numbers (0 is neutral).

In the modulo world there cannot be any overflows due to any operation but the final answer may not be what we are interested in. For example, when $w = 3$, if we add 101 (-3 in 2's complement) and 110 (-2), then the answer is 011 (3 mod 8) whereas we would have liked -5. But -5 does not not have a representation in 3 bits - we need to go to 4 bits (1011) for that. More precisely, -5 and 3 are *indistinguishable* modulo 8 or that this the case of an *overflow*. So we need to safeguard against this since OCAML doesn't give an any overflow signal. Note that it is not simply a carry out of the MSB, since adding 111 (-1) and 110 (-2) also produces a carry but the answer 101 (-3) is correct since -3 can be represented using 2's complement in 3 bits. **Therefore as a programmer, we need to devise ways to detect overflow.**

Exercise 5 *Convince yourself that for any integers x and y*
(i) $x \mod m + y \mod m = (x + y) \mod m$.
(ii) If $m = 2^w$, then the $t \leq w$ least significant binary digits of $x \mod m$ are the least t binary digit representation of x.
Thereby conclude that whenever $x + y$ does not go out of range, we obtain the correct answer irrespective of the "signed" interpretation of x and y.
(iii) Write your own function **myadd** *that adds two integers when there is no overflow and signals overflow otherwise.*

4 Strenghtening Induction assertions

To prove that `ifib (0,1,1,n)` returns *Fibn* for $n \geq 2$, one is tempted to prove it by induction on n. Although base case ($n = 2$) is easy, carrying out the inductive step is difficult without making any assertions for the other parameters. Therefore we must *strengthen* the induction

[1] This is very useful at the hardware level to detect if an integer is negative
[2] In Ocaml, one of the bits is a *tag* bit for special purpose - so effectively you have 31 bits.

hypothesis to assert about the remaining parameters also. One such asserion can be

Claim 1 *For all* $n \geq 2$, `ifib(0,1,1,n)` *calls* `ifib(Fib i-1, Fib i, i,n)` *for* $2 \leq i \leq n$.

Proof: If the claim is true (for all n) then, `ifib(0,1,1,n)` will call `ifib(Fib n-1,Fib n, n,n)` which will return $Fibn$ and we are done. We shall prove it by induction on i.
Base case: For all $n \geq 2$, `ifib(0,1,1,n)` calls `ifib(1,1,2,n)`. Since $Fib1 = 0$ and $Fib2 = 1$, we are done[3].
Inductive Step: Suppose the assertion is true for all $2 \leq i < n$, i.e. (for all $n \geq 2$), for all $2 \leq i < n$, `ifib(0,1,1,n)` calls `ifib(Fib i-1 ,Fib i,i,n)`. Since `ifib(Fib n-1 ,Fib n-2 ,n-1,n)` calls `ifib (Fib n-1 , Fib n-1 + Fib n-2, n ,n)`, the inductive step is complete. Note that in the inductive proof, we always argued "For all $n \geq 2$" - this is crucial for the eventual goal of computing $Fibn$ for all $n \geq 2$. □

4.1 Two kinds of induction proofs

Inductive proofs are typically used to prove a property (predicate) for all non-negative integers. For example, to prove that the sum of the first n integers is $\frac{n \cdot (n+1)}{2}$, we can have $P(i)$ represent the *predicate* that $\sum_{j=1}^{j=i} j = \frac{i \cdot (i+1)}{2}$ for any integer $i \geq 0$. The *Principle of Mathematical Induction* (PMI) states that

$$P(0) \wedge \forall k[P(k) \Rightarrow P(k+1)] \Rightarrow \forall n P(n)$$

A variation of the PMI, called *Principle of Complete Induction* (PCI) states the following

$$P(0) \wedge \forall k[P(0) \wedge P(1) \ldots P(k) \Rightarrow P(k+1)] \Rightarrow \forall n P(n)$$

Often this variation is more useful, especially in situations that involve *structural induction* where a bigger structure is decomposed in terms of smaller structures but not necessarily having size exactly less than one (as the PMI requires).

Before we proceed to use it, let us convince ourselves that the two *avatars* are essentially equivalent, namely, that if we believe one, the other follows by logical inference. For this, it should be clear that PCI implies PMI (Why ?). Let us try to prove the converse, viz., PMI implies PCI.

Given an arbitrary predicate $P(i)$ that we are trying to prove, let us define another predicate $P'(i)$ as $\forall k, k \leq i, P(k)$. So $P'(i)$ is a predicate that holds if $P(i)$ holds for all $k \leq i$. Clearly

$$[\forall n P(n)] \Leftrightarrow [\forall n, P'(n)]$$

although $P(i)$ and $P'(i)$ are not equivalent. From PMI, we know that

$$P'(0) \wedge \forall k[P'(k) \Rightarrow P'(k+1)] \Rightarrow \forall n P'(n) \Rightarrow [\forall n P(n)] \quad (1)$$

[3]We have the index starting from 0, i.e. $F_0 = 0$ $F_1 = 1$ and so on

where the last implication follows from the previous observation. From the definition of $P'(i)$, the antecedent $\forall k[P'(k) \Rightarrow P'(k+1)]$ can be rewritten as

$$\forall k[P(0) \wedge P(1) \ldots P(k) \Rightarrow P(0) \wedge P(1) \ldots P(k) \wedge P(k+1)]$$

Since $P(i) \Rightarrow P(i)$, the above is equivalent to $\forall k[P(0) \wedge P(1) \ldots P(k) \Rightarrow P(k+1)]$ and since $P'(0) \Leftrightarrow P(0)$, equation 1 can be rewritten as

$$P(0) \wedge \forall k[P(0) \wedge P(1) \ldots P(k) \Rightarrow P(k+1)] \Rightarrow [\forall n P(n)]$$

which is precisely the statement for PCI.

5 Efficiency of programs

The running times of different programs (corresponding to the same problem) may vary significantly depending on the underlying algorithm. Of course, the same program will take different time to execute depending on the computer where it is running. So rather than comparing the real-time of execution, we will try to compare the number of instructions that are executed by the different programs. Here again we assume that each instruction takes unit time. [4]

The number of instructions executed will depend on the input or more precisely the *input-size*. The larger the input, the more the number of instructions executed. For example, for computing Fibonacci numbers, more instructions are executed to compute the 50th Fibonacci number than the 10th Fibonacci number. In fact, we would like to express the number of instructions to compute the k-th Fibonacci number as a function of k. So, the efficiency of a program can be expressed in terms of a function that represents the number of instructions executed in terms of the input-size.
This function is often referred to as time complexity

5.1 How to compare functions

If we have two programs have time complexities $3n + 6$ and $2n^2 + 8$ respectively then it is clear that we will prefer the first program. Here n is the input size. Suppose they have time complexities $9n + 20$ and $n^2 + 1$, then the choice may not be clear since the second program is superior initially but for larger n then the first program is better. In such situations, we will choose the program that *asymptotically* (eventually) becomes better. That is for some n_o, for all $n \geq n_o$ the program executes fewer instructions than others.

Therefore, it is meaningful only to look at the leading term (with the largest exponent) and we also ignore the multiplicative constants. In other words $3n^2 + 6n + 8$ will be approximated by n^2 - the notation used is $O(n^2)$ (read big-O of n^2).

[4] This is somewhat of an oversimplification as multiplication takes longer than addition - but still serves our purpose to get estimates of the running times

5.2 Efficiency of recursive functions

Since induction forms the basis of recursive functions, we express the number of instructions as a inductive relation often called *recurrence relations*. We then find solutions (which are functions) that satisfy this relation (analogous to finding roots of simple equations). Finding solutions to recurrence relations is often a very difficult exercise.

Consider the Fibonacci program based on the recursive definition. The number of instructions for Fib(n) is the the summation of the number of instructions for Fib(n-1) and Fib(n-2) plus three more instructions (corresponding to two conditionals and one addition). Therefore we can express the recurrence relation for $T(n)$ = number of instructions for computing Fib(n) as

$$T(n) = T(n-1) + T(n-2) + 3$$

What function of n will satisfy this relation ?
Can you get an upper-bound based on Exercise 15 of the ocaml handout ?

However the other recursive version (fibnew preceding section 5.2 of the ocaml notes) has a simpler structure since the function makes exactly one recursive call with `i` incremented by 1. Therefore we can express $T(i,n) = T(i+1,n)+3$ and $T(n,n) = 1$ which yields $T(i,n) = 3(n-i)$ and in particular $T(3,n) = 3(n-3)$ since the initial recursive call is with $i = 3$.

6 Functions in Caml and λ calculus

Lambda calculus (or λ-calculus as is popularly known) is perhaps one of the earliest formal programming languages. In spite of its seemingly limited forms it manages to match (in some sense define) our notion of mechanical computation. With minimal syntax (that is defined) below it can be used to represent all known programming constructs and virtually does away with all semantics that often creates ambiguity in implementing programming languages. Yet, because of the extreme cumbersome nature of writing programs, its application is primarily limited to proving theorems rather than writing software.

In its *purest* form λ-calculus , uses only two constructs - *abstraction* and *application*. The former is to define (new) functions and the latter is to actually apply a function to given parameters. A typical λexpression takes the form $\lambda x.fx$ which is to be interpreted as a one parameter[5] function f that can be *reduced* by *applying* it to an *actual* parameter. The parameter x is a *formal* parameter. The variable x is *bound* within the *scope*. A variable that is not bound is called *free*. For example, in $\lambda y.\lambda z.xz(yz)$, x is *free* whereas z, y are *bound*. In this brief overview we are avoiding rigorous definitions that are used to identify *free* and *bound* variables. Note that valid λ expressions are written using the alphabet $\lambda, ., (,)$ in addition to the parameter names. Without parentheses, the symbols associate from left, i.e., abc is the same as $((ab)c)$.

The result of applying an abstract $(\lambda x.M)$ to an argument N is to "substitute" N for all *free* occurences of x in M. Moreover we must ensure that the free variables of N are not bound

[5]there is a natural extension of this notation to multiple parameter, viz., $\lambda xy.fxy$

in M. So in case of conflicts, we must "rename" these variables. We are again avoiding formal definitions.

Pure λ-calculus doesn't support any (predefined) functions or constants but these can be defined as λ-terms λ-calculus .

For example, constants can be defined as follows (which are also called Church numerals). The integers 0 and 3 are represented as

$$\lambda f(\lambda x.x) \text{ and } \lambda f(\lambda x.f(f(fx)))$$

Imagine how messy it is to represent 1000 (essentially a unary representation that applies f 1000 times). Very often, we use shorthand like f^k to denote $f(f(f \ldots k\text{times}$, i.e., the i-th composition of f with itself; but this is not part of the standard vocabulary of λ-calculus . The addition function can be represented as

$$\lambda xy.(\lambda fz.xf(yfz))$$

However, if you have followed the intuition behind the Church's numerals, viz., that integer n corresponds to n fold composition of f (zero corresponds to just the parameter), then you can easily verify by induction that the addition of m and n correspond to $f^m(f^n(x)$ that is $f^{m+n}(x)$.

Following a similar line of arguement try to prove that the multiplication function can be represented as

$$\lambda mn.\lambda f(m(nf))$$

The boolean constants **true** and **false** are represented as

$$\lambda x.(\lambda yx) \text{ and } \lambda x.(\lambda yy)$$

The standard conditional construct like if-then-else can be thought of as a function with three parameters - a condition (which is boolean), a "then" part and an "else" part. The following expression can be used to represent it.

$$\lambda cond(\lambda then(\lambda else((cond\ then)else)))$$

As an example when we apply it to the parameters **true**, **this that**, we obtain

$$\lambda cond(\lambda then(\lambda else((cond\ then)else)))\lambda x.(\lambda yx) \text{ this that}$$

$$\begin{aligned}
&= (\lambda then(\lambda else((true\ then)else))) \text{ this that} \\
&= (\lambda then(\lambda else((\lambda x.(\lambda yx)then)else))) \text{ this that} \\
&= (\lambda then(\lambda else(((\lambda y\ then))else))) \text{ this that} \\
&= (\lambda then(\lambda else(then))) \text{ this that} \\
&= (\lambda else(\text{this})) \text{ that} \\
&= \text{this}
\end{aligned}$$

This should convince you somewhat that it is possible to build complex instructions using λ-calculus although it does become messy and hence not used widely. A careful reader may have already noticed the similarity between λ expressions and function definition in Ocaml. More specifically $\lambda x.f$ is written as `function x -> f` in Ocaml and the reader can try out the above examples.

Exercise 6 *Verify the following λ terms for the boolean functions*

1. $and = \lambda p(\lambda q((p\ q)false))$
2. $or = \lambda p(\lambda q((p\ true)q))$
3. $not = \lambda p((p\ false)true)$

6.1 An Ocaml example

A very common problem is to write a function that composes two given function f and g (assuming that $f(g)$ is well defined). Since the λ expressions are isomorphic to the Ocaml functions, let us write out a lambda expression for composition. The λ expressions for f and g are $\lambda x.fx$ and $\lambda x.gx$ respectively. Their composition can be defined as another λ expression of the form $\lambda x.f(gx)$ [6] The input to the composition function are two λ-expressions (functions) and the output is also a λ-expression.

Consider the following λ expression

$$\lambda a\lambda b.\lambda x a(bx)$$

If we *apply* it to $a = \lambda x.fx$ and $b = \lambda y.gy$ (we have used y for clarity - it is not really necessary), then we obtain the following sequence of reductions

$$\lambda a\lambda b.\lambda x a(bx)\ \ \lambda x.fx\ \ \lambda y.gy$$

$$\Rightarrow \lambda x\lambda x.fx(\lambda y.gy\ \ x)$$

$$\Rightarrow \lambda x\lambda x.fx(g\ x)$$

$$\Rightarrow \lambda x f(g\ x)$$

which is what we wanted as the output. The equivalent Ocaml program is

```
# let comp a b = function x -> a ( b x) ;;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Let us verify by trying out

[6] The parenthesis is necessary in this case, otherwise it will be treated as $\lambda x.(fg)x$.

```
# comp (function x -> x*x*x) (function x -> x*x) ;; (* composing the square
                                    and the cube functions *)
- : int -> int = <fun>
# comp (function x -> x*x*x) (function x -> x*x) 2 ;;  (* applying the
                     the previous composition to compute 2^6 *)
- : int = 64
```

Can you reason out the type of `comp` returned by Ocaml ?
One way to reason will be - suppose the final answer is in domain γ, i.e., $f(gx)$ is of type γ. Let gx be of type β, then f must be of type $\beta \to \gamma$. If x is of type α, then g must be of type $\alpha \to \beta$. So the output of `comp` is an expression of type $\alpha \to \gamma$. The input is the type of f which is $\beta \to \gamma$. So we can write the type as

$$(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

Verify using $\alpha = \ 'c, \beta = \ 'a, \gamma = \ 'b$.

6.2 Versatility of functional inputs

Many function/programs require functions/programs as inputs by definition. Some of the most common examples are Computing derivative of a function, finding roots of a function (say, by the bisection method), composition of functions (that we discussed in the previous section) among others. In addition, there are many situations where we can use the power of abstraction to make the programs more versatile. That is, depending on the application, it can adapt by using a specific function. Consider the following example of an n-fold (n is an arbitrary integer) composition of an arbitrary function with itself, namely $f^{(n)} = f(f($..n times. In Ocaml, we can write such a function easily as follows

```
# let rec selfcomp f n = if n = 0 then (function x -> x) else
                   function x -> f ( selfcomp f (n-1) x) ;;
val selfcomp : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

One useful application of this function could be generating a sequence, where the n-th term of the sequence can be obtained by applying an *appropriate* function to the first term. For example, consider an arithmetic progression

$$1, 4, 7, 10 \ldots 1 + 3(i-1)$$

We can generate the n-th term by using the function `function x -> x + 3`. Therefore for $n = 10$, we get

```
selfcomp (function x -> x+3) 9 1 ;; (* for the n-th term, we compose n-1
times*)
- : int = 28
```

For the geometric series

$$2, 6, 18, 54 \ldots 2 \cdot 3^{i-1}$$

we have

```
# selfcomp (function x -> x*3) 4 2 ;; (* the 5th term of the sequence *)
- : int = 162
```

Try the same for a harmonic sequence (the reciprocal of the terms form an arithmetic progression). We can try to even generate the Fibonacci sequence with a more sophisticated function.

```
# selfcomp (function (x,y) -> (x+y , x)) 4 (1,0) ;;
- : int * int = (5, 3)
```

Here the function is applied to two parameters which are similar to the i-th and the $i-1$-st terms of the Fibonacci sequence.

Here is another way of writing the n-fold composition. Verify its correctness. The function `comp` is the familiar composition function. Is it more efficient ?

```
let rec fastselfcomp f n = if n =0 then (function x -> x ) else
  if (n mod 2 = 0) then
       comp (fastselfcomp f (n/2)) (fastselfcomp f (n/2))
         else comp f (fastselfcomp f (n-1)) ;;
```

Exercise 7 *How would you use the* `selfcomp` *to compute*
(i) A harmonic sequence ?
(ii) The sum of the first n terms of a geometric sequence ?

Using lambda expressions we can also easily move back and forth between multi-parameter function to single parameter function, also called *curry* and vice versa. In Ocaml, we can write a higher-order function that converts a 2 parameter function as follows.

```
let curry f = function x -> function y -> f(x,y) ;;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let mysum (a,b) = a + b ;;
val mysum : int * int -> int = <fun>
# let mysum1 = curry mysum ;;
val mysum1 : int -> int -> int = <fun>
# mysum 2 ;;
This expression has type int but is here used with type int * int
# mysum1 2 ;;
- : int -> int = <fun>
```