Tik-79.148 **Spring 2001**
Introduction to Theoretical Computer Science
Tutorial 11
Answers to Demonstration Exercises

1. Primitive recursive functions are formed from three initial functions using two combinating rules.

   The initial functions are:

   (a) *The zero functions* $\mathsf{zero}_k(n_1, \ldots, n_k) = 0$ for all $k, n_1, \ldots, n_k \in \mathbb{N}$.
   (b) *Identity function* $\mathsf{id}_{k,j}(n_1, \ldots, n_k) = n_j$, for all $k \geq j > 0$, $n_1, \ldots, n_k \in \mathbb{N}$.
   (c) *The successor function* $\mathsf{succ}(n) = n + 1$ for all $n \in \mathbb{N}$.

   The initial functions zero and succ can be used to define natural numbers:

$$
\begin{aligned}
\mathsf{zero}() &= 0 \\
\mathsf{succ}(\mathsf{zero}()) &= 1 \\
\mathsf{succ}(\mathsf{succ}(\mathsf{zero}())) &= 2 \\
&\vdots \\
\mathsf{succ}(n) &= n + 1
\end{aligned}
$$

   We can combine initial functions using composition and primitive recursion:

   (a) Let $k, l \geq 0$, g k-ary functions ($g : \mathbb{N}^k \to \mathbb{N}$), and $h_1, \ldots, h_k$ a set of l-ary functions. Now the *composition* of g with $h_1, \ldots, h_k$ is the l-ary function:

$$f(n_1, \ldots, n_l) = g(h_1(n_1, \ldots, n_l), \ldots, h_k(n_1, \ldots, n_l))$$

   In composition the value returned by of one function is given as an argument to another function. It is not necessary to use different h_i functions. For example, the function $f(n, m) = n^2 + m^2$ can be written as the composition of plus and times:

$$f(n, m) = \mathsf{plus}(\mathsf{times}(n, n), \mathsf{times}(m, m))$$

   Here $l = k = 2$, $g = \mathsf{plus}$ and $h_1 = h_2 = \mathsf{times}$.

   (b) For all $k \geq 0$, let g be a k-ary function and h a $k+2$-ary function. The $k+1$-ary function f is obtained by *primitive recursion* if:

$$
\begin{aligned}
f(n_1, \ldots, n_k, 0) &= g(n_1, \ldots, n_k) \\
f(n_1, \ldots, n_k, m + 1) &= h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m))
\end{aligned}
$$

   for all $n_1, \ldots, n_k, m \in \mathbb{N}$.
   In primitive recursion we have a simple basic case ($m = 0$) as well as a set of recursion equations that are used to transform more complicated cases to the basic case.

The set of *primitive recursive functions* contains all initial functions as well as all functions that can be formed from them using composition and primitive recursion.

In the exercise we have to prove that the function

$$f(n) = \text{"}n + 1\text{th odd number"}$$

is primitive recursive. There are several ways to define it, here is one of them:

$$\begin{aligned} f(0) &= 1 \\ f(m+1) &= f(m) + 2 \end{aligned}$$

Since the numbers 1 and 2 are not initial functions, we replace them using compositions of the succ function:

$$\begin{aligned} f(0) &= \mathsf{succ}(\mathsf{zero}()) \\ f(m+1) &= \mathsf{succ}(\mathsf{succ}(f(m))) \end{aligned}$$

Here $k = 0$, $g = \mathsf{succ}(\mathsf{zero}())$ and $h = \mathsf{succ}(\mathsf{succ}(f(m)))$.

2. By adding a new operation, *minimization* we can construct a larger class of functions. A minimization statement is of the form:

$$\mu z.[P(z)] \ ,$$

where z is a variable and $P(z)$ is some predicate (that is, a function that has two possible values: *true* (1) and *false* (0)) whose valua is a function of z. The value of a minimization is the least z for which $P(z)$ is true.

Formally: Let g be a k+1-ary function, $k \geq 0$. Then the k-ary minimization function f of g is defined as follows:

$$f(n_1, \ldots, n_k) = \begin{cases} \text{the least } m \text{ that satisfies } g(n_1, \ldots, n_k, m) = 1 & \text{, if } m \text{ exists} \\ 0 & \text{, otherwise} \end{cases}$$

A predicate g is *regular* if $g(n_1, \ldots, n_k, m) = 1$ for some $m \geq 0$. It is not possible to know in the general case whether g is regular. The underlying problem is the same as in the Turing machine halting problem. A function is *μ-recursive* (or simply recursive) if if can be obtained from the initial functions using composition, primitive recursion, and minimization of regular predicates.

In addition to regular minimization we may also use *bounded minimization* of the form:

$$\mu z_{\leq x}.[P(z)] \ ,$$

where x is the upper bound for the value of z. If no $z \leq x$ satisfies the predicate, the result of bounded minimization is 0. The value of bounded minimization can always be computed since x gives the upper bound for the number of computation steps. The bounded minimization can be implemented using primitive recursion so the class of primitive recursive functions is closed with respect to it.

Intuitively, a bounded minimization corresponds to the following program snippet:

$$\text{for i} := 1 \text{ to } n \text{ do}$$

The key point is that there is an upper bound for the loop counter.

An unbounded minimization corresponds to the snippet:

$$\textbf{while} \text{ not end do}$$

Here we do not know on advance when if ever the predicate 'end' is satisfied.

We can compute the remainder of two integers using the following μ-recursive function:

$$\text{mod } (x, y) = \begin{cases} 0 & , x = 0 \vee y = 0 \\ \mu z_{\leq x} . \left[\mu w_{\leq x} . \left[y \cdot w + z = x \right] \right] & , \text{otherwise.} \end{cases}$$

Since both above minimizations are bounded, the function is actually even primitive recursive.

The function corresponds to the following C-language program:

```
int modulo(int x, int y)
{
    int z, w;
    if (x == 0 || y == 0)
        return 0;

    for (z = 0; z <= x; z++) {
        for (w = 0; w <= x; w++) {
            if (w*y + z == x)
                return z;
        }
    }
}
```

3. Since recursive functions are from numbers to numbers, we can't use them to directly handle functions on strings. However, we can use a systematic way to encode the strings into numbers. This process is called *Gödel* numbering. There are many different ways to define Gödel numbers. The one we use here transforms strings over a n-letter alphabet into $n+1$ base integers that are further transformed into decimal numbers.

   In the exercise we have the following alphabet: $\Delta = \{a, b, c\}$ and we impose an order on the individual letters: järjestykseen

   $$d_1 = a$$
   $$d_2 = b$$
   $$d_3 = c$$

   The number base will be $\beta = |\Delta| + 1 = 4$.

A string $w = d_{i_1} d_{i_2} \dots d_{i_k}$ is transformed into a number using the usual method for changing the base:

$$\mathsf{gn}(w) = \beta^{k-1} \cdot i_1 + \beta^{k-2} \cdot i_2 + \cdots + \beta^1 \cdot i_{k1} + i_k$$

In addition we use the symbol d_0 to represent the empty string e. It is added because otherwise there would be numbers (for example, β) that don't correspond to any strings. However, we now have a problem that by adding the empty string we may change the Gödel number of a string. In this case the canonical number is one without any empty strings at all.

(a) The Gödel number of abc is obtained as follows:

$$\begin{aligned} \mathsf{gn}(abc) &= 1 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 \\ &= 27 \end{aligned}$$

(b) The Gödel number 19 corresponds to a string:

$$\begin{aligned} 19 &= 1 \cdot 4^2 + 0 \cdot 4^1 + 3 \cdot 4^0 \\ \mathsf{gn}^{-}1(19) &= d_1 d_0 d_3 = aec = ac \end{aligned}$$

4. A problem is in the complexity class NP if it can be solved using a non-deterministic Turing machine in polynomial time with respect to the length of the input. In practice this means that if we can verify the answer in polynomial time using a deterministic Turing machine, then it is in NP.

   A problem is NP-complete if all NP problems can be *reduced* to it in a polynomial time. Here the intuition is to show that a problem A is at least as difficult as the problem B by transforming an instance of A into an instance of B.

   We usually prove that a problem is NP-complete in two phases:

   (a) Show that the problem is in NP.

   (b) Reduce some known NP problem into it.

   The kernel of a graph $G = (V, E)$ is a $K \subseteq V$ such that:

   (a) There are no edges between nodes in K.

   (b) All nodes not in the kernel can be reached in one step from some node in kernel.

   The problem of finding a kernel is clearly in NP since we can solve it with a non-deterministic Turing machine in the following fashion:

   (a) Guess a set K of nodes. We need $O(|K|)$ steps for this.

   (b) Iterate over all edges (a, b) leaving from nodes in K. If the edge goes into another node in K, we reject the answer. Otherwise we put a mark on b. This phase takes $O(|E|)$ steps.

   (c) Finally we check that all nodes not in the kernel are marked and reject the answer if some one is not. This takes $O(|V|)$ steps.

Next we have to reduce some NP-complete problem to the kernel problem. One of the most fundamental NP-complete problems is the so called 3-SAT-problem where we are given a set of propositional clauses that each have exactly three literals and we want to find a way to assign truth values to the propositional atoms so that each clause has at least one true literal.

For example, the clause set

$$\{\{x_1, \neg x_2, x_3\}, \{\neg x_1, \neg x_2, \neg x_3\}\}$$

is satisfiable since choosing x_1 and x_2 to be *true* and x_3 to be *false*, each clause has at least one true literal. The atom x_1 satisfied the first clause and the literal $\neg x_3$ satisfies the second. A truth assigment is also called a *valuation* and it may be denoted by several different ways. For example, the above valuation would be $\mathcal{V} = \{x_1, x_2, \neg x_3\}$, or $\mathcal{V}(x_1) = \mathcal{V}(x_2) = T$, $\mathcal{V}(x_3) = F$.

The 3-SAT-to-kernel reduction goes as follows:

Suppose that a 3-SAT instance has the variables $x_1, \ldots, x_n$ and the clauses $C = \{C_1, \ldots, C_m\}$ Construct a graph $G = (V, E)$ such that for each clause C_i there will be a single node c_i and for each variable x_i there are two nodes, x_i and $\neg x_i$. The adjacency relation is defined as follows:

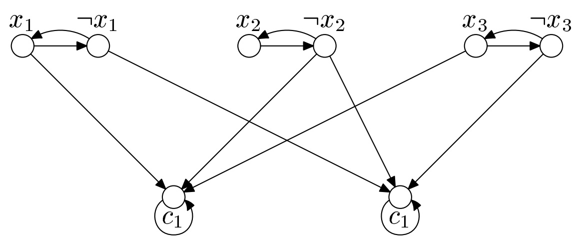
(a) For all x_i, there are $(x_i, \neg x_i), (\neg x_i, x_i) \in E$.

(b) For all $x_i \in C_j$, $(x_i, c_j) \in E$.

(c) For all $\neg x_i \in C_j$, $(\neg x_i, c_j) \in E$.

(d) For all c_i, $(c_i, c_i) \in E$.

For example, the graph corresponding to the above set of clauses is:



Now a node x_i is in a kernel if and only if the variable x_i is true in a valuation that satisfies the clauses. Similarily, $\neg x_i$ is in a kernel if x_i is false in the valuation. Both x_i and $\neg x_i$ may not be in a kernel since there are always arcs between them. On the other hand, since all nodes c_i have a self-loop, either x_i or $\neg x_i$ has to be in the kernel.

The above graph has a kernel $K = \{x_1, x_2, \neg x_3\}$ that corresponds to the valuation $x_1 = T$, $x_2 = T$, $x_3 = F$.

Finally, we construct a formal proof that the reduction works. If the graph G has a kernel K, we may construct a satisfying valuation as follows: $\mathcal{V}(x_i) = T$, iff $x_i \in K$, otherwise $\mathcal{V}(x_i) = F$. Since K is a kernel, then by construction of G there is at least some node $x \in K$ such that $(x, c_i) \in E$ for each c_i. Again, by construction $x \in C_i$ so the clause C_i is satisfied. This holds for all clauses so $\mathcal{V}$ satisfies C.

If C is satisfiable, there exists at least one $\mathcal{V}$ that satisfies it. Construct K by having $x_i \in K$ whenever $\mathcal{V}(x_i) = T$ and $\neg x_i \in K$ whenever $\mathcal{V}(x_i) = F$. Since all clauses are satisfied, there exists at least one *true* literal x for each clause C_i. By construction of G, there is an edge between x and c_i, so all nodes c_i are covered by K. Similarily, the complement of x is also covered. As a valuation assigns a unique value for each propositional atom, there may not be two nodes with a connecting edge in K. Thus, K is a kernel.