Addis Ababa University, Amist Kilo July 13, 2011

Algorithms and Programming for High Schoolers

Lab 8

Exercise 1: Write a recursive procedure `addDigits`(n) which takes a nonnegative integer n and returns the sum of the digits of n.

Example solution:

```
def addDigits(n):
    if n==0:
        return 0
    return (n%10) + addDigits(n/10)
```

Exercise 2: Python already has a function `reverse()` for lists (`L.reverse()` reverses the `list` L). Let's implement our own `reverse()` function using recursion. `reverse`(L) should be a recursive function that outputs a `list` which contains the elements of L in reverse. Remember that L[i:] evaluates to a `list` containing only the elements of L from index i onward.

Example solution:

```
def reverse(L):
    if len(L)==0:
        return []
    return [L[len(L)-1]] + reverse(L[:len(L)-1])
```

Exercise 3: Write a recursive procedure `minElement`(L) which takes a `list` L of integers and returns the minimum element in the `list`.

Example solution:

```
def minElement(L):
    if len(L)==1:
        return L[0]
    return min(L[0], minElement(L[1:]))
```

Exercise 4: A *superknight* is on a chessboard, at grid location $(0, 0)$ (the bottom left corner). How many ways can he get to the location (x, y) if his allowed moves are given in the `list` L? Write a function `numKnightWays`(x,y,L) that returns this number. Each element in L is a list of size two [i,j] signifying that it is possible for the knight to move from (a, b) to $(a+i, b+j)$. i, j are always both positive.

Example solution:

```
# return the number of ways to get to x,y given that we are currently
# at position (atx, aty)
def knightRecurse(atx, aty, x, y, L):
    if (atx>x) or (aty>y):
        return 0
    elif atx==x and aty==y:
        return 1
    ans = 0
    for t in L:
        ans += knightRecurse(atx+t[0], aty+t[1], x, y, L)
    return ans

def numKnightWays(x, y, L):
    return knightRecurse(0, 0, x, y, L)
```

The above solution can be sped up using memoization, by memoizing based on `atx` and `aty`.

```
def knightRecurse(atx, aty, x, y, L, mem):
    if (atx>x) or (aty>y):
        return 0
    elif atx==x and aty==y:
        return 1
    elif mem[atx][aty] != -1:
        return mem[atx][aty]
    mem[atx][aty] = 0
    for t in L:
        mem[atx][aty] += knightRecurse(atx+t[0], aty+t[1], x, y, L, mem)
    return mem[atx][aty]

def numKnightWays(x, y, L):
    mem = []
    for i in xrange(x+1):
        mem += [[-1]*(y+1)]
    return knightRecurse(0, 0, x, y, L, mem)
```

Exercise 5: An *expression* is defined recursively as follows. An integer is an expression, which evaluates to the integer itself. If `EXPR` is an expression, then so is (`EXPR`), and it evaluates to whatever `EXPR` evaluated to. Finally, if `EXPR1` and `EXPR2` are expressions, then (`OP EXPR1 EXPR2`) is an expression, where `OP` can be any one of $+, -, *$, and it evaluates to `evaluate`(`EXPR1`) `OP` `evaluate`(`EXPR2`). You should write a function `evaluate` which takes a `str` and evaluates the expression it is a valid expression, and outputs "INVALID" if it is not a valid expression. For example:

- `evaluate`('(+ 1 5)') gives 6.

- `evaluate`('(* 3 (- 5 2))') gives 9 (first (- 5 2) is evaluated as $5 - 2 = 3$, and then we have $3 * 3 = 9$).
- `evaluate`('(+ 1 (+ 5))') gives "INVALID" since (+ 5) is not a valid expression.
- `evaluate`('()') gives "INVALID" since the empty string is not a valid expression.

Example solution: Below is a recursive solution.

```
# returns True if c represents a digit from '0' to '9'
# and False otherwise
def isDigit(c):
    return c>='0' and c<='9'

def isOperator(c):
    return c=='+' or c=='-' or c=='*'

# find the first prefix of expr which could be a valid expression
# return INVALID of no such prefix exists
def locateExpression(expr):
    if len(expr) == 0:
        return 'INVALID'
    elif isDigit(expr[0]):
        # try to build an integer expression as a prefix
        at = 1
        while at<len(expr) and isDigit(expr[at]):
            at += 1
        return expr[:at]
    elif expr[0] == '-':
        # try to build a negative integer expression as a prefix
        if len(expr) == 1:
            return 'INVALID'
        elif not isDigit(expr[1]):
            return 'INVALID'
        at = 2
        while at<len(expr) and isDigit(expr[at]):
            at += 1
        return expr[:at]
    elif expr[0] == '(':
        # find the matching parenthesis
        x = 1
        at = 1
        while at<len(expr) and x>0:
            if expr[at]=='(':
                x += 1
            elif expr[at]==')':
```

```
                x -= 1
            at += 1
        if x != 0:
            return 'INVALID'
        return expr[:at]
    else:
        return 'INVALID'

def evaluate(expr):
    if len(expr) == 0:
        return 'INVALID'
    elif expr[0] != '(':
        if locateExpression(expr) == expr:
            return int(expr)
        else:
            return 'INVALID'
    else:
        if len(expr) == 1:
            # if the first letter is '(', we at least need ')' at end
            return 'INVALID'
        elif expr[len(expr)-1] != ')':
            return 'INVALID'
        elif isOperator(expr[1]):
            # in this case we need to apply an OP to two exprs
            if (len(expr) < 7) or (expr[2] != ' '):
                # we need at least 7 characters for (, OP, two spaces, and two exprs
                return 'INVALID'
            expr1 = locateExpression(expr[3:])
            if expr1 == 'INVALID':
                return 'INVALID'
            elif len(expr1) >= len(expr)-4:
                # if expr1 is too long, there's no room left for expr2
                return 'INVALID'
            elif expr[3+len(expr1)] != ' ':
                # a space should separate the two expressions
                return 'INVALID'
            expr2 = locateExpression(expr[4+len(expr1):])
            if expr2 == 'INVALID':
                return 'INVALID'
            elif len(expr1)+len(expr2)+5 != len(expr):
                # expr should be (OP space expr1 space expr2)
                return 'INVALID'
            # evaluate the two expressions recursively
            A = evaluate(expr1)
            B = evaluate(expr2)
```

```
    if (A==‘INVALID’) or (B==‘INVALID’):
        return ‘INVALID’
    elif expr[1] == ‘+’:
        return A + B
    elif expr[1] == ‘-’:
        return A - B
    else:
        return A * B
else:
    return evaluate(expr[1:len(expr)-1])
```