COSC 4315 - Homework 3

Madison Pratt

October 19, 2018

Contents

Requirements

1. No loops may be used. Recursion must be used instead.
2. Must support nested functions in input, up to a depth of 3.
3. No global variables may be used.
4. Code must be organized into functions.
5. All parameters must be passed by value. Passing by reference is disallowed.
6. All functions must be commented with preconditions and postconditions.
7. All math must operate on lists representing "infinite integers."
8. Program must declare invalid or malformed input.

String to Infinite Integer

Strings are read into "infinite integer" lists by starting from the end of the string and reading n characters back recursively until fewer than n characters remain (where n represents the digits per node). Each returning step of the recursion then adds its n ending characters to the resulting list of the previous step until the full integer is built. This solution naturally "aligns" digits such that any non-full nodes are shifted to the most significant digits.

Pseudocode

Algorithm 1 Building an infinite integer recursively

```
procedure StrToInfInt(s : string; n : int)
    if len(s) <= n then
        return a list composed of an integer representation of s
    else
        s_i = an integer representing the last n digits of s
        remove the last n digits of s
        t = StrToInfInt(s, n)
        append s_i to the end of t
        return t
    end if
end procedure
```

Implementation

```
def StrToInfInt(s, n):
    if(len(s) <= n):
        return [int(s)]
    else:
        si = int(s[-n:])
        t = StrToInfInt(s[:-n], n)
        t.append(si)
        return t
```

Infinite Integer to String

Pseudocode

Algorithm 2 Returns a string representation of the integer

```
procedure InfIntToStr(a : list; i, n : int)
    if i = len(a) then
        return “”
    else if i = 0 then
        result = InfIntToStr(a, i + 1, n)
        result = a[i] as a trimmed string + result
    else
        result = InfIntToStr(a, i + 1, n)
        result = a[i] as an n-digit string + result
    end if
    return result
end procedure
```

Implementation

```
def InfIntToStr(s, i, n):
    if i == len(s):
        return ‘‘’’
    elif i == 0:
        result = str(s[i]) + InfIntToStr(s, i + 1, n)
    else:
        result = str(s[i]).zfill(n) + InfIntToStr(s, i + 1, n)
    return result
```

Infinite Integer Addition

Two "infinite integers" are given to be added together. Starting from the least significant digits, nodes are added together (similarly to how addition by hand is performed, but using multiple digits at a time). If the sum exceeds 10^n, we will carry a 1 in the next recursive step and reduce the sum for the current step by 10^n. The last elements in both lists are removed and addition continues towards the most significant digits. Each recursive step returns the full "infinite integer" list for the addition performed so far.

Pseudocode

Algorithm 3 Recursively adding two infinite integers together

```
procedure AddInfInt(a, b : list; c, n : int)
    if a and b are empty lists then
        if c ≠ 0 then
            return a list containing c
        else
            return nothing
        end if
    else
        sum = c + the least significant nodes of a and b
        if sum >= 10^n then
            sum = sum - 10^n
            carry = 1
        else
            carry = 0
        end if
        remove the least significant nodes from a and b
        result = AddInfInt(a, b, carry, n)
        append sum to result
        return result
    end if
end procedure
```

Implementation

```
def AddInfInt(a, b, c, n):
    if a == b == []:
        if c != 0:
            return [c]
        else:
            return
    if len(a) > 0:
        ai = a[-1]
    else:
        ai = 0
    if len(b) > 0:
        bi = b[-1]
    else:
        bi = 0
    sum = c + ai + bi
    if sum >= 10 ** n:
        sum -= 10 ** n
        carry = 1
    else:
        carry = 0
    a = a[:-1]
    b = b[:-1]
    result = AddInfInt(a, b, carry, n) or []
    result += [sum]
    return result
```

Infinite Integer Multiplication

This operation is performed similarly to multiplication by hand. The algorithm multiplies groups of digits together and sums the result. Mathematically, this can be represented by the following:

$$\sum_{i=0}^{n}\sum_{j=0}^{m}10^{d(i+j)}b_i a_j \tag{1}$$

(1) n: total nodes in b, m: total nodes in a, d: digits per node; indices move right to left

However, using list-represented integers introduces some challenges. Instead of multiplying the nodes by $10^{d(i+j)}$, it is best to create an infinite integer with $(i+j)$ zero-value nodes to the right. It is also important to be mindful of carry-out values; while the equation above does not need to respect carry-out, it must be ensured that the algorithm respects the specific digits per node requirement. Therefore, each step of the recursion should produce an infinite integer of the form `[carry][product] (i+j)*[0]` to be added to each successive step.

Pseudocode

Algorithm 4 Recursively multiplying two infinite integers together

  procedure MultiplyAcross(a: list; b, i, n: int)
    if i = len(a) **then**
      return 0 represented as an infinite integer
    else
      $product_{i+1}$ = MultiplyAcross(a, b, i + 1, n)
      $product_i$ = b * a[i]
      if $product_i >= 10^n$ **then**
        carry = $\lfloor product_i / 10^n \rfloor$
        $product_i = product_i \bmod 10^n$
      end if

```
        append len(a) − i − 1 zero-value nodes to product_i
        prepend carry to product_i if it exists
        return product_{i+1} + product_i
    end if
end procedure

procedure MultiplyInfInt(a, b: list; i, n: int)
    if i = len(b) then
        return 0 represented as an infinite integer
    else
        result_{i+1} = MultiplyInfInt(a, b, i + 1, n)
        result_i = MultiplyAcross(a, b[i], 0, n)
        append len(b) − i − 1 zero-value nodes to the end of result_i
        return result_i + result_{i+1}
    end if
end procedure
```

Implementation

```
def MultiplyAcross(a, b, i, n):
    if i == len(a):
        return [0]
    else:
        pnext = MultiplyAcross(a, b, i + 1, n)
        product = b * a[i]
        carry = 0
        if product >= 10 ** n:
            carry = math.floor(product / 10 ** n)
            product = product \% (10 ** n)
        product = [product] + [0] * (len(a) - i - 1)
        if carry > 0:
            product = [carry] + product
    return AddInfInt(product, pnext, 0, n)

def MultiplyInfInt(a, b, i, n):
    if i == len(b):
        return [0]
    else:
        resultnext = MultiplyInfInt(a, b, i + 1, n)
        result = MultiplyAcross(a, b[i], 0, n)
        result = result + [0] * (len(b) - i - 1)
        return AddInfInt(result, resultnext, 0, n)
```

Expression Evaluation

The goal is to take a single line of input in the form of an expression, and produce an infinite integer result. As with prior algorithms, this will be a recursive approach.

All input provided to this algorithm will be in the form `[function name]` followed by either an integer constant or another `[function name] (...)`. Function calls should be recursively evaluated and reduced to infinite integer constants, which can be operated upon. Regular expressions will be used to identify sections of the input string in the form `[function name]([constant], [constant])`. These subexpressions will be solved and their results re-inserted into the input expression until the entire expression is solved.

Pseudocode

Algorithm 5 Recursively solving an expression

```
procedure SolveLine(s, n : int)
    use regular expressions to test for the desired function call form
    if expression does not exist then        ▷ The input is unsolvable and thus invalid
        return [-1]
    else
        if match = s then                    ▷ This is the base case
            if s contains "add" then
                x = left + right sides of expression
            else
                x = left * right sides of expression
            end if
            return x
```

else
s_1 = slice s around regex match
x = SolveLine(s_1)
replace s_1 in s with x
return SolveLine(s)
end if
end if
end procedure

Implementation

```
def SolveLine(s, n):
    find = re.search('((add|multiply)\(\d+,\d+\))', s)
    if not find:
        return [-1]
    else:
        if find.group(1) == s:
            si = s.split('(')
            numResults = re.match(''([0-9]+),([0-9]+)", si[1])
            nums = [StrToInfInt(numResults.group(1), n), StrToInfInt(numResults.grou
            if si[0] == ''add'':
                x = AddInfInt(nums[0], nums[1], 0, n)
            elif si[0] == ''multiply'':
                x = MultiplyInfInt(nums[0], nums[1], 0, n)
            return x
        else:
            s1 = s[find.start(1):find.end(1)]
            x = SolveLine(s1, n)
            s = s.replace(s1, InfIntToStr(x, 0, n))
            return SolveLine(s, n)
```

Processing Input

Psuedocode

Algorithm 6 Recursively solving all lines of input

```
procedure SolveInput(s, n)
    if len(s) = 0 then
        return
    else
        result = SolveLine(s[0], n)
        if result ≠ [-1] then
            output s[0] and result
        else
            output “invalid expression”
        end if
        remove s[0]
        SolveInput(s, n)
    end if
end procedure
```

Implementation

```python
def SolveInput(s, n):
    if len(s) == 0:
        return
    result = SolveLine(s[0], n)
    if result != [-1]:
        print(s[0], ''='', InfIntToStr(result, 0, n))
    else:
        print(``Invalid expression: ``, s[0])
    SolveInput(s[1:], n)
```