Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions
2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers
3. Exercises

MCS 275 Lecture 17
Programming Tools and File Management
Jan Verschelde, 17 February 2017

Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions
2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers
3. Exercises

Arithmetical Expressions in Postfix

Infix is our common notation

In postfix format, the operator follows the operands.

Example: `2 + 3` is written as `2 3 +`

Operands, like `2` and `3` are separated by space.

Advantage: no brackets needed.

For example:

`3 2 + 4 *` is the same as `(3 + 2) * 4`

`2 4 * 3 +` is the same as `3 + 2 * 4`

This simple representation of expressions is used in stack based languages as POSTSCRIPT.

lists used as stacks

Pushing elements on the stack with `insert`:

```
>>> S = []
>>> S.insert(0,'2')
>>> S.insert(0,'3')
>>> S
['3', '2']
>>> S.insert(0,'+')
>>> S
['+', '3', '2']
```

To evaluate, do `pop`:

```
>>> op = S.pop(0)
>>> e = S.pop(0) + op + S.pop(0)
>>> e
'3+2'
>>> eval(e)
5
```

Evaluation of Postfix Expressions

parsing a postfix expression string

Use stacks to evaluate postfix expressions.

Scan string for operands and operators:

- if operand, then push to the stack
- if operator, then:
    1. pop first operand from stack
    2. pop second operand from stack
    3. compute the result of the operation
    4. push the result on the stack

At the end: value is on top of the stack.

the function `eval_postfix`

```
def eval_postfix(exp):
    """
    Returns the value of the expression exp,
    given as string in postfix notation.
    An exception handler prints the stack.
    """
    opd = ''
    stk = []
    try:
        for char in exp:
            (stk, opd) = update_postfix(stk, opd, char)
        print('S =', stk)
        return stk[0]
    except:
        print('exception raised at ')
        print('c =', char, 'S =', stk)
        return 0
```

running `eval_postfix`

Evolution of the stack, for `2 3 + 4 *`,
character after character:

```
S = []
S = ['2']
S = ['2']
S = ['3', '2']
S = ['5']
S = ['5']
S = ['5']
S = ['4', '5']
S = ['20']
```

the function update_postfix

```
OPERATORS = ['+', '-', '*', '/']

def update_postfix(stk, opd, char):
    """
    Evaluates operations to numbers, via an
    update of the stack stk with a character char,
    where opd is the current operand.
    If char is an operator, then its arguments
    are popped from the stack and the result
    of the operation is pushed on the stack.
    The new stk and opd are returned as (stk, opd).
    """
```

Multidigit arguments are read character after character and concatenated again as strings.

Also the intermediate values are stored as strings.

code for update_postfix

```
OPERATORS = ['+', '-', '*', '/']

def update_postfix(stk, opd, char):

    if char == ' ':
        if opd != '':
            stk.insert(0, opd)
            opd = ''
        return (stk, opd)
    elif char in OPERATORS:
        exp = stk.pop(1) + char + stk.pop(0)
        value = eval(exp)
        stk.insert(0, str(value))
        return (stk, opd)
    else:
        return (stk, opd + char)
```

Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions
2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers
3. Exercises

postfix to infix

Convert `2 3 + 4 *` into infix
by storing the arithmetical expression as a string:

```
S = []
S = ['2']
S = ['2']
S = ['3', '2']
S = ['3+2']
S = ['3+2']
S = ['3+2']
S = ['4', '3+2']
S = ['4*(3+2)']
```

brackets around expressions

In going from postfix to infix, we must place brackets around all expressions that are not all numerical.

```
def bracket(item):
    """
    Returns item with round brackets around
    it if item is not a number.
    """
    if item.isalnum():
        return item
    else:
        return '(' + item + ')'
```

evaluate to a string – specification

```
def eval_string(stk, opd, char):
    """
    Evaluates operations to a string, via an
    update of the stack stk with a character char,
    where opd is the current operand.
    If char is an operator, then its arguments
    are popped from the stack and the result
    of the operation is pushed on the stack.
    The new stk and opd are returned as (stk, opd).
    """
```

evaluate to a string – implementation

```
def eval_string(stk, opd, char):

    if char == ' ':
        if opd != '':
            stk.insert(0, opd)
            opd = ''
        return (stk, opd)
    elif char in OPERATORS:
        exp = bracket(stk.pop(1)) + char
        exp = exp + bracket(stk.pop(0))
        stk.insert(0, exp)
        return (stk, opd)
    else:
        return (stk, opd + char)
```

Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions

2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers

3. Exercises

evaluating infix expressions

Use stack to evaluate `(4*(3+2))`:

```
S = []
S = []
S = ['*', '4']
S = ['*', '4']
S = ['*', '4']
S = ['+', '3', '*', '4']
S = ['+', '3', '*', '4']
S = ['5', '*', '4']
S = ['20']
```

the function `update_infix`

```
def update_infix(stk, opd, char):
    """
    Evaluates operations to numbers, via an
    update of the stack stk with a character char,
    where opd is the current operand.
    If char is a closing bracket, then two
    operands and an operator are popped
    from the stack and the result
    of the operation is pushed on the stack.
    The new stk and opd are returned as (stk, opd).
    """
```

`update_infix()` is called by `eval_infix()`, a function that processes expression strings character by character, accumulating multidigit operands in the string `opd`.

the definition of `update_infix`

```
def update_infix(stk, opd, char):

    if char in OPERATORS:
        if opd != '':
            stk.insert(0, opd)
            opd = ''
        stk.insert(0, char)
        return (stk, opd)
    elif char == ')':
        if opd == '':
            opd = stk.pop(0)
        exp = stk.pop(0)
        exp = stk.pop(0) + exp + opd
        value = eval(exp)
        stk.insert(0, str(value))
        return (stk, '')
    elif char != '(':
        return (stk, opd + char)
    else:
        return (stk, opd)
```

Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions
2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers
3. Exercises

recursive function calls

Consider a recursive `gcd`:

```
def gcd(alpha, beta):
    """
    Returns greatest common divisor
    of the numbers alpha and beta.
    """
    rest = alpha % beta
    if rest == 0:
        return beta
    else:
        return gcd(beta, rest)
```

Goal: transform into an equivalent iterative version.

$\rightarrow$ Use a stack to execute recursion.

running `gcdstack`

```
$ python gcdstack.py
give a : 2146
give b : 2244
S = ['gcd(2146,2244)']
S = ['gcd(2244,2146)']
S = ['gcd(2146,98)']
S = ['gcd(98,88)']
S = ['gcd(88,10)']
S = ['gcd(10,8)']
S = ['gcd(8,2)']
gcd(2146,2244) = 2
```

stack for handling recursion

```
def gcdstack(alpha, beta):
    """
    Builds the stack of function calls in
    a recursive gcd for alpha and beta.
    """
    from ast import literal_eval
    stk = ['gcd(%d,%d)' % (alpha, beta)]
    while stk != []:
        print('S =', stk)
        exp = stk.pop(0)
        (alpha, beta) = literal_eval(exp[3:len(exp)])
        rest = alpha % beta
        if rest == 0:
            result = beta
        else:
            stk.insert(0, 'gcd(%d,%d)' % (beta, rest))
    return result
```

Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions

2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers

3. Exercises

computing factorials recursively

If $n \leq 1$, then $n! = 1$, else $n! = n \times (n-1)!$.

```
def fac(nbr):
    """
    Returns the factorial of the number nbr.
    """
    if nbr <= 0:
        return 1
    else:
        return nbr*fac(nbr-1)
```

running `facstack`

Computing 5!:

```
S = ['F(5)']
S = ['F(4)', 'F(5)']
S = ['F(3)', 'F(4)', 'F(5)']
S = ['F(2)', 'F(3)', 'F(4)', 'F(5)']
S = ['F(1)', 'F(2)', 'F(3)', 'F(4)', 'F(5)']
F(5) = 120
```

an iterative version

```
def facstack(nbr):
    """
    Builds the stack of function calls in
    a recursion for the factorial of nbr.
    """
    from ast import literal_eval
    stk = ['F(%d)' % nbr]
    while stk != []:
        print('S =', stk)
        exp = stk.pop(0)
        nbr = literal_eval(exp[2:len(exp)-1])
        if nbr <= 1:
            result = 1
            while stk != []:
                exp = stk.pop(0)
                nbr = literal_eval(exp[2:len(exp)-1])
                result = result * nbr
        else:
            stk.insert(0, 'F(%d)' % nbr)
            stk.insert(0, 'F(%d)' % (nbr-1))
    return result
```

Stacks of Function Calls

1. Stacks to Evaluate Expressions
   - in postfix format
   - postfix to infix
   - evaluating infix expressions

2. Recursive Function Calls
   - transform into iteration via stack
   - stack for the recursive factorial
   - stack for the Fibonacci numbers

3. Exercises

computing Fibonacci numbers

$F_0 = 0$, $F_1 = 1$, for $n > 1$: $F_n = F_{n-1} + F_{n-2}$.

```
def fib(nbr):
    """
    Returns the n-th Fibonacci number, n = nbr.
    """
    if nbr == 0:
        return 0
    elif nbr == 1:
        return 1
    else:
        return fib(nbr-1) + fib(nbr-2)
```

running `fibstack`

Computing `F(4)`:

```
S = ['F(4)']
S = ['F(3)', 'F(2)']
S = ['F(2)', 'F(1)', 'F(2)']
S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']
S = ['F(0)', 'F(1)', 'F(2)']
S = ['F(1)', 'F(2)']
S = ['F(2)']
S = ['F(1)', 'F(0)']
S = ['F(0)']
F(4) = 3
```

an iterative version

```
def fibstack(nbr):
    """
    Builds the stack of function calls in
    a recursion for n-th Fibonacci number.
    """
    from ast import literal_eval
    stk = ['F(%d)' % nbr]
    result = 0
    while stk != []:
        print('S =', stk)
        exp = stk.pop(0)
        nbr = literal_eval(exp[2:len(exp)-1])
        if nbr <= 1:
            result = result + nbr
        else:
            stk.insert(0, 'F(%d)' % (nbr-2))
            stk.insert(0, 'F(%d)' % (nbr-1))
    return result
```

Exercises

1. Make a class `Stack` using a list as object data attribute encapsulating the list operations with the proper `push` and `pop` operations. Use this `Stack` in the evaluation of a postfix expression.
2. Write Python code to store a postfix arithmetical expression in a binary tree. The data at the nodes are the operators, while the operands are at the leaves. Provide routines to write the content of the tree using prefix, infix, and postfix traversal orders.
3. Consider a recursive definition of the Harmonic numbers H_n: $H_1 = 1$ and for $n > 1$: $H_n = H_{n-1} + 1/n$. Use a stack to write an equivalent iterative version.
4. Make an iterative version of `enumbits.py` (discussed in Lecture 11), using a stack.