

Stacks of Function Calls

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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, 19 February 2010

Stacks of Function Calls

Stacks to
Evaluate
Expressions

in postfix format

postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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.

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.

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.

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.

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

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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
```

Lists used as Stacks

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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
```

Lists used as Stacks

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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
```

Lists used as Stacks

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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.

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.

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.

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.

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.

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.

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 EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def EvalPostFix(e):
    """
    Returns the value of the expression e,
    given as string in postfix notation.
    An exception handler prints the stack.
    """
    op = ''
    S = []
    try:
        for c in e:
            (S,op) = UpdatePValue(S,op,c)
            print 'S =', S
        return S[0]
    except:
        print 'exception raised at '
        print 'c =', c, 'S =', S
        return 0
```

The Function EvalPostFix

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def EvalPostFix(e):
    """
    Returns the value of the expression e,
    given as string in postfix notation.
    An exception handler prints the stack.
    """
    op = ''
    S = []
    try:
        for c in e:
            (S,op) = UpdatePValue(S,op,c)
            print 'S =', S
        return S[0]
    except:
        print 'exception raised at '
        print 'c =', c, 'S =', S
        return 0
```

The Function EvalPostFix

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def EvalPostFix(e):
    """
    Returns the value of the expression e,
    given as string in postfix notation.
    An exception handler prints the stack.
    """
    op = ''
    S = []
    try:
        for c in e:
            (S,op) = UpdatePValue(S,op,c)
            print 'S =', S
        return S[0]
    except:
        print 'exception raised at '
        print 'c =', c, 'S =', S
        return 0
```

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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']

Running EvalPostFix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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 UpdatePValue

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

```
def UpdatePValue(S,op,c):
```

```
    """
```

```
    Evaluates operations to numbers, via an
    update of the stack S with a character c,
    where op is the current operand.
```

```
    If c is an operator, then its arguments
    are popped from the stack and the result
    of the operation is pushed on the stack.
    The new S and op are returned as (S,op).
```

```
    """
```

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

Also the intermediate values are stored as strings.

The Function UpdatePValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

```
def UpdatePValue(S,op,c):
```

```
    """
```

```
    Evaluates operations to numbers, via an
    update of the stack S with a character c,
    where op is the current operand.
```

```
    If c is an operator, then its arguments
    are popped from the stack and the result
    of the operation is pushed on the stack.
    The new S and op are returned as (S,op).
```

```
    """
```

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

Also the intermediate values are stored as strings.

Code for UpdatePValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

def UpdatePValue(S,op,c):
    if c == ' ':
        if op != '':
            S.insert(0,op)
            op = ''
        return (S,op)
    elif c in operators:
        p = S.pop(0) + c + S.pop(0)
        value = eval(p)
        S.insert(0,str(value))
        return (S,op)
    else:
        return (S,op + c)
```

Code for UpdatePValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

def UpdatePValue(S,op,c):
    if c == ' ':
        if op != ' ':
            S.insert(0,op)
            op = ' '
        return (S,op)
    elif c in operators:
        p = S.pop(0) + c + S.pop(0)
        value = eval(p)
        S.insert(0,str(value))
        return (S,op)
    else:
        return (S,op + c)
```

Code for UpdatePValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

def UpdatePValue(S,op,c):
    if c == ' ':
        if op != ' ':
            S.insert(0,op)
            op = ' '
        return (S,op)
    elif c in operators:
        p = S.pop(0) + c + S.pop(0)
        value = eval(p)
        S.insert(0,str(value))
        return (S,op)
    else:
        return (S,op + c)
```

Stacks of Function Calls

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
 evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
 stack for the recursive factorial
 stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
 evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
 stack for the recursive factorial
 stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
 evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
 stack for the recursive factorial
 stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = ['2']$

$S = ['2']$

$S = ['3', '2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['3+2']$

$S = ['4', '3+2']$

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

Postfix to Infix

Convert $2\ 3\ +\ 4\ *$ into infix by storing the arithmetical expression as 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

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

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

Brackets around Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

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

Evaluate to a String

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

```
def EvalString(S,op,c):
    if c == ' ':
        if op != '':
            S.insert(0,op)
            op = ''
        return (S,op)
    elif c in operators:
        p = Bracket(S.pop(0)) + c
        p = p + Bracket(S.pop(0))
        S.insert(0,p)
        return (S,op)
    else:
        return (S,op + c)
```

Evaluate to a String

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def EvalString(S,op,c):  
    if c == ' ':  
        if op != '':  
            S.insert(0,op)  
            op = ''  
        return (S,op)  
    elif c in operators:  
        p = Bracket(S.pop(0)) + c  
        p = p + Bracket(S.pop(0))  
        S.insert(0,p)  
        return (S,op)  
    else:  
        return (S,op + c)
```

Evaluate to a String

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def EvalString(S,op,c):  
    if c == ' ':  
        if op != '':  
            S.insert(0,op)  
            op = ''  
        return (S,op)  
    elif c in operators:  
        p = Bracket(S.pop(0)) + c  
        p = p + Bracket(S.pop(0))  
        S.insert(0,p)  
        return (S,op)  
    else:  
        return (S,op + c)
```

Stacks of Function Calls

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
**evaluating infix
expressions**

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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 ']

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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 ']

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = []$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['5', '*', '4']$

$S = ['20']$

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = []$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['5', '*', '4']$

$S = ['20']$

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = []$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['5', '*', '4']$

$S = ['20']$

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = []$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['5', '*', '4']$

$S = ['20']$

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = []$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['5', '*', '4']$

$S = ['20']$

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

$S = []$

$S = []$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['+', '3', '*', '4']$

$S = ['5', '*', '4']$

$S = ['20']$

Evaluating Infix Expressions

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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 UpdateIValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

```
def UpdateIValue(S,op,c):
    """
    Evaluates operations to numbers, via an
    update of the stack S with a character c,
    where op is the current operand.
    If c 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 S and op are returned as (S,op).
    """
```

UpdateIValue() is called by EvalInFix(), a function that processes expression strings character by character, accumulating multidigit operands in the string `op`.

Code for UpdateIValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

```
def UpdateIValue(S,op,c):
    if c in operators:
        if op != '':
            S.insert(0,op)
            op = ''
        S.insert(0,c)
        return (S,op)
    elif c == ')':
        if op == '': op = S.pop(0)
        p = S.pop(0)
        p = S.pop(0) + p + op
        value = eval(p)
        S.insert(0,str(value))
        return (S,'')
    elif c != '(':
        return (S,op + c)
    else:
        return (S,op)
```

Code for UpdateIValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

```
def UpdateIValue(S,op,c):
    if c in operators:
        if op != '':
            S.insert(0,op)
            op = ''
        S.insert(0,c)
        return (S,op)
    elif c == ')':
        if op == '': op = S.pop(0)
        p = S.pop(0)
        p = S.pop(0) + p + op
        value = eval(p)
        S.insert(0,str(value))
        return (S,'')
    elif c != '(':
        return (S,op + c)
    else:
        return (S,op)
```

Code for UpdateIValue

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

```
def UpdateIValue(S,op,c):
    if c in operators:
        if op != '':
            S.insert(0,op)
            op = ''
        S.insert(0,c)
        return (S,op)
    elif c == ')':
        if op == '': op = S.pop(0)
        p = S.pop(0)
        p = S.pop(0) + p + op
        value = eval(p)
        S.insert(0,str(value))
        return (S,'')
    elif c != '(':
        return (S,op + c)
    else:
        return (S,op)
```

Stacks of Function Calls

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Consider a recursive gcd:

```
def gcd(a,b):  
    """  
    Returns greatest common divisor  
    of the numbers a and b.  
    """  
    r = a % b  
    if r == 0:  
        return b  
    else:  
        return gcd(b,r)
```

Goal: transform into an equivalent iterative version.

→ Use a stack to execute recursion.

Recursive Function Calls

Consider a recursive gcd:

```
def gcd(a,b):  
    """  
    Returns greatest common divisor  
    of the numbers a and b.  
    """  
    r = a % b  
    if r == 0:  
        return b  
    else:  
        return gcd(b,r)
```

Goal: transform into an equivalent iterative version.

→ Use a stack to execute recursion.

Recursive Function Calls

Consider a recursive gcd:

```
def gcd(a,b):  
    """  
    Returns greatest common divisor  
    of the numbers a and b.  
    """  
    r = a % b  
    if r == 0:  
        return b  
    else:  
        return gcd(b,r)
```

Goal: transform into an equivalent iterative version.

→ Use a stack to execute recursion.

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

```
$ 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
```

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
$ 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
```

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
$ 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
```

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
$ 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
```

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
$ 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
```

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
$ 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
```

Running gcdstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
$ 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

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def gcdstack(a,b):
    """
    Builds the stack of function calls
    in a recursive gcd for a and b.
    """
    S = ['gcd(%d,%d)' % (a,b)]
    while S != []:
        print 'S =', S
        e = S.pop(0)
        (a,b) = eval(e[3:len(e)])
        r = a % b
        if r == 0:
            result = b
        else:
            S.insert(0,'gcd(%d,%d)' % (b,r))
    return result
```

Stacks of Function Calls

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
**stack for the
recursive factorial**
stack for the
Fibonacci numbers

Exercises

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

based on a recursive definition

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

```
def F(n):  
    """  
    Returns the n-th factorial.  
    """  
    if n <= 0:  
        return 1  
    else:  
        return n*F(n-1)
```

Computing Factorials Recursively

based on a recursive definition

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

```
def F(n):  
    """  
    Returns the n-th factorial.  
    """  
    if n <= 0:  
        return 1  
    else:  
        return n*F(n-1)
```

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
**stack for the
recursive factorial**
stack for the
Fibonacci numbers

Exercises

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
```

Running facstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
**stack for the
recursive factorial**
stack for the
Fibonacci numbers

Exercises

Computing 5!:

$$S = ['F(5)']$$

$$S = ['F(4)', 'F(5)']$$

$$S = ['F(3)', 'F(4)', 'F(5)']$$

$$S = ['F(2)', 'F(3)', 'F(4)', 'F(5)']$$

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

$$F(5) = 120$$

Running facstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
**stack for the
recursive factorial**
stack for the
Fibonacci numbers

Exercises

Computing 5!:

$$S = ['F(5)']$$

$$S = ['F(4)', 'F(5)']$$

$$S = ['F(3)', 'F(4)', 'F(5)']$$

$$S = ['F(2)', 'F(3)', 'F(4)', 'F(5)']$$

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

$$F(5) = 120$$

Running facstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
**stack for the
recursive factorial**
stack for the
Fibonacci numbers

Exercises

Computing 5!:

$$S = ['F(5)']$$

$$S = ['F(4)', 'F(5)']$$

$$S = ['F(3)', 'F(4)', 'F(5)']$$

$$S = ['F(2)', 'F(3)', 'F(4)', 'F(5)']$$

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

$$F(5) = 120$$

Running facstack

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix expressions

Recursive Function Calls

transform into iteration via stack
stack for the recursive factorial
stack for the Fibonacci numbers

Exercises

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
```

Running facstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
**stack for the
recursive factorial**
stack for the
Fibonacci numbers

Exercises

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

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

```
def facstack(n):
    """
    Builds the stack of function calls in
    a recursion for the factorial of n.
    """
    S = ['F(%d)' % n]
    while S != []:
        e = S.pop(0)
        n = eval(e[2:len(e)-1])
        if n <= 1:
            result = 1
            while S != []:
                e = S.pop(0)
                n = eval(e[2:len(e)-1])
                result = result * n
        else:
            S.insert(0, 'F(%d)' % n)
            S.insert(0, 'F(%d)' % (n-1))
    return result
```

Stacks of Function Calls

Stacks to Evaluate Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

from a recursive definition

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

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

Computing Fibonacci Numbers

from a recursive definition

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

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

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

Computing $F(4)$:

$S = ['F(4)']$

$S = ['F(3)', 'F(2)']$

$S = ['F(2)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(0)', 'F(1)', 'F(2)']$

$S = ['F(0)', 'F(1)', 'F(2)']$

$S = ['F(1)', 'F(2)']$

$S = ['F(2)']$

$S = ['F(1)', 'F(0)']$

$S = ['F(0)']$

$F(4) = 3$

Running fibstack

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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

Stacks to
Evaluate
Expressions

in postfix format
postfix to infix
evaluating infix
expressions

Recursive
Function Calls

transform into
iteration via stack
stack for the
recursive factorial
stack for the
Fibonacci numbers

Exercises

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