Lecture #25: Calculator

Adminitrivia

- Extended TA office hours in labs Tuesday from 11AM.
- Exam is at 7PM on Wednesday; rooms to be assigned as happened last time (not the same rooms: see postings and email to come).
- No lecture on Wednesday, but I'll be in my office.
- Exam is open-book; no responsive devices.

A Sample Language: Calculator

- Source: John Denero.
- Prefix notation expression language for basic arithmetic Python-like syntax, with more flexible built-in functions.

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

Syntax and Semantics of Calculator

Expression types:

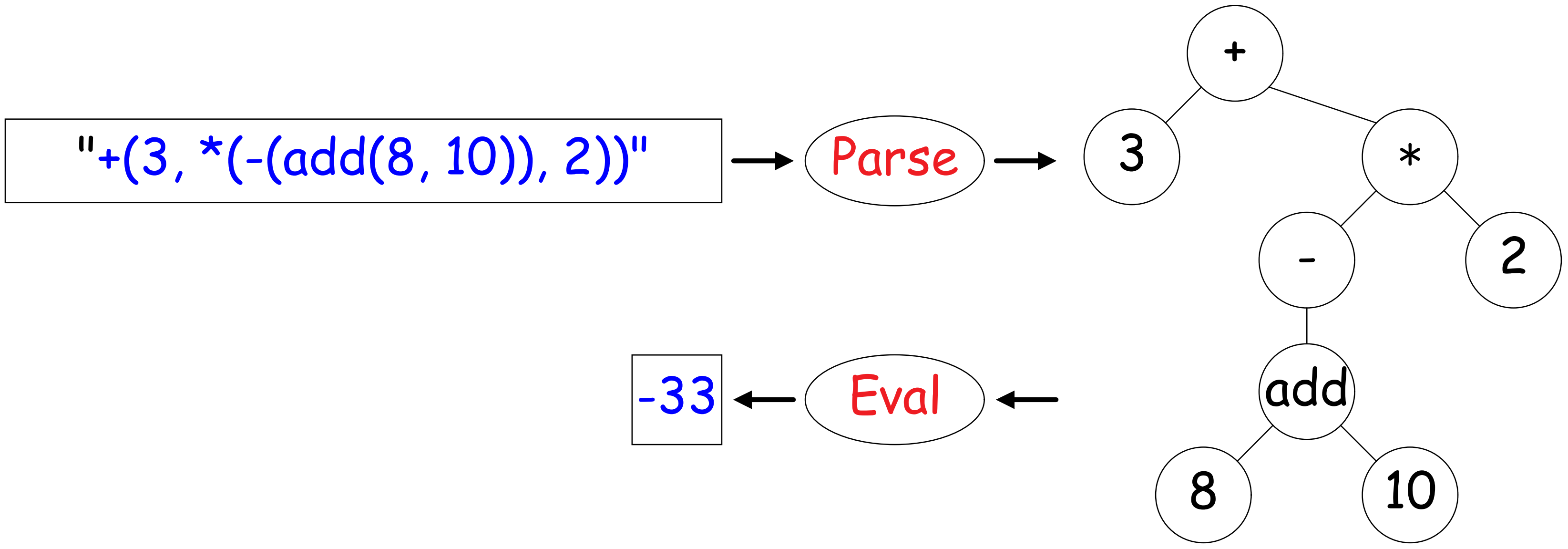
- A *call expression* is an operator name followed by a comma-separated list of operand expressions, in parentheses.
- A *primitive expression* is a number.

Operators:

- The add (or +) operator returns the sum of its arguments
- The sub (-) operator returns either
  - the additive inverse of a single argument, or
  - the sum of subsequent arguments subtracted from the first.
- The mul (*) operator returns the product of its arguments.
- The div (/) operator returns the real-valued quotient of a dividend and divisor.

Strategy

- Our calculator program represents expressions as trees (see Lecture #20).
- It consists of a *parser*, which produces *expression trees* from *input text*, and an *evaluator*, which performs the computations represented by the trees to produce *values*.
- You can use the term *"interpreter"* to refer to both, or to just the evaluator.



Expression Trees (augmented)

To create an expression tree:

```
class Exp(object):
       """An expression"""
   def __init__(self, operator_or_value, operands = None):
       """If OPERANDS is None, a primitive OPERATOR_OR_VALUE.
       Otherwise, an expression with OPERATOR_OR_VALUE as its
       operator and OPERANDS (a list of Exps) as its operands."""
       self._opval = operator_or_value
       self._operands = operands

   @property
   def operator(self): return self._opval
   @property
   def operands(self): return self._operands

   @property
   def is_primitive(self): return self._operands is None
   @property
   def value(self): return self._opval
```

Expression Trees By Hand

Let's define the methods `__repr__` and `__str__` to produce reasonable representations of expression trees:

```
>>> Exp('add', [Exp(1), Exp(2)])       # Intepreter uses .__repr__
Exp('add', [Exp(1), Exp(2)])

>>> str(Exp('add', [Exp(1), Exp(2)])) # str uses .__str__
'add(1, 2)'

>>> Exp('add', [Exp(1), Exp('*', [Exp(2), Exp(3), Exp(4)])])
Exp('add', [Exp(1), Exp('*', [Exp(2), Exp(3), Exp(4)])])

>>> str(Exp('add', [Exp(1), Exp('*', [Exp(2), Exp(3), Exp(4)])]))
'add(1, *(2, 3, 4))'
```

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) "evaluate to themselves" (corresponds to Exps evaluating to their .values.)
- Call expressions are evaluated recursively, following the tree structure:
  - Evaluate each operand expression, collecting values as a list of arguments.
  - Apply the named operator to the argument list.

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if exp.is_primitive:

    else:
```

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) "evaluate to themselves" (corresponds to Exps evaluating to their .values.)
- Call expressions are evaluated recursively, following the tree structure:
  - Evaluate each operand expression, collecting values as a list of arguments.
  - Apply the named operator to the argument list.

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if exp.is_primitive:
         return exp.value
    else:
```

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.

- Primitive expressions (literals) "evaluate to themselves" (corresponds to Exps evaluating to their .values.)
- Call expressions are evaluated recursively, following the tree structure:
  - Evaluate each operand expression, collecting values as a list of arguments.
  - Apply the named operator to the argument list.

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if exp.is_primitive:
         return exp.value
    else:
         arguments = list(map(calc_eval, exp.operands))
         return calc_apply(exp.operator, arguments)
```

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args (which are numbers).
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + 'requires at least 1 argument')
        if len(args) == 1:
            return -args[0]
        return sum(args[:1] + [-arg for arg in args[1:]])
    etc.
```

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop that

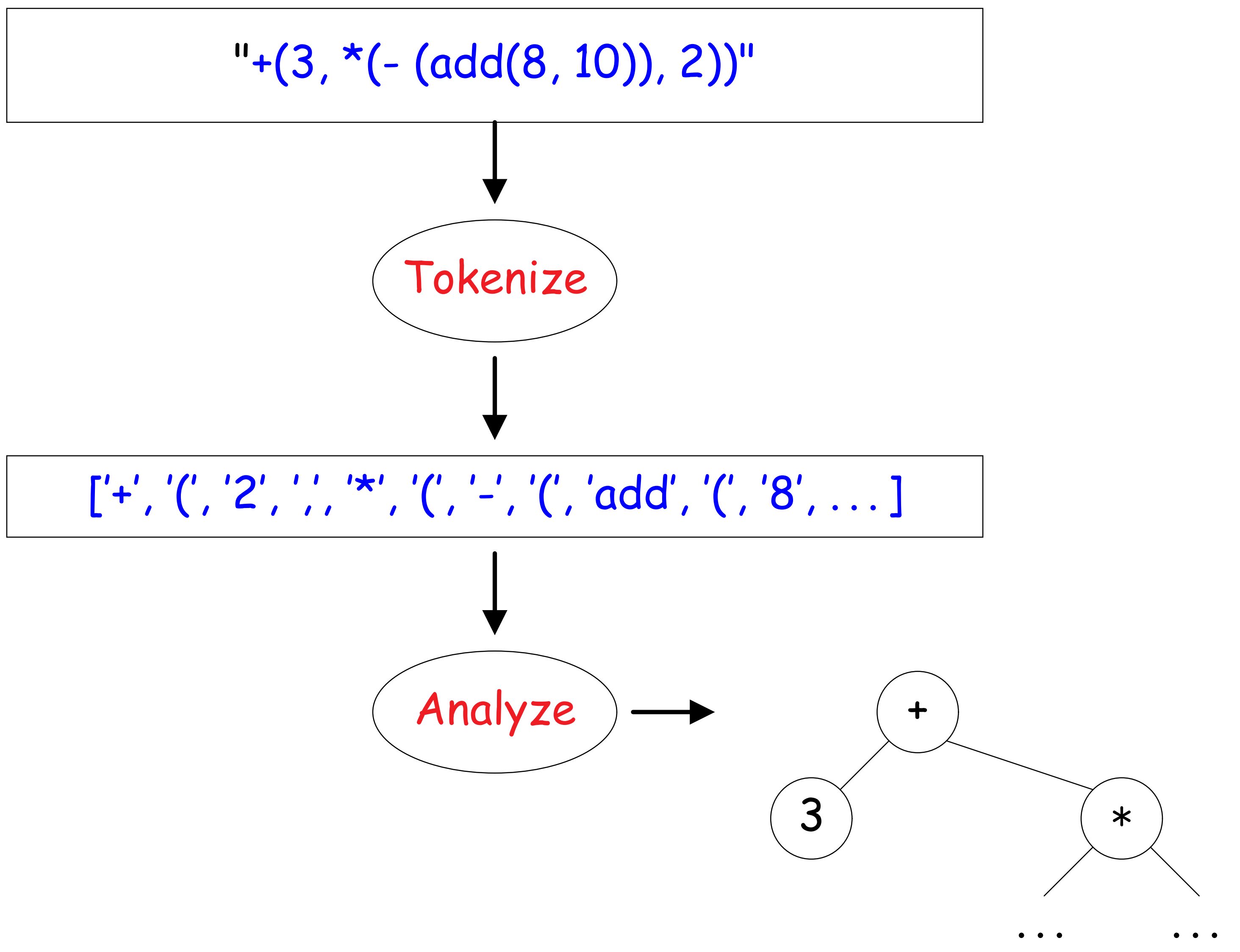
- Reads an expression from the user
- Parses the input to build an expression tree
- Evaluates the expression tree
- Prints the resulting value of the expression

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except:
            print error message and recover
```

Parsing: Lexical and Syntactic Analysis

- To *parse* a text is to analyze it into its constituents and to describe their relationship or structure.
- Thus, we can parse an English sentence into nouns, verbs, adjectives, etc., and determine what plays the role of subject, what is plays the role of object of the action, and what clauses or words modify what.
- When processing programming languages, we typically divide task into two stages:
  - *Lexical analysis* (aka *tokenization*): Divide input string into meaningful *tokens*, such as integer literals, identifiers, punctuation marks.
  - *Syntactic analysis:* Convert token sequence into trees that reflect their meaning.

Parsing Strategy


"+(3, *(- (add(8, 10)), 2))"
Tokenize
['+', '(', '2', ',', '*', '(', '-', '(', 'add', '(', '8', . . .]
Analyze
+
3
*
. . .
. . .

Tokenization

- In principle, we could dispense with tokenizing and go from text to trees directly, but
- We choose to break input into these particular chunks because they correspond to how we think about and describe the text, and thus make analysis simpler:
  - We say “the word ‘add’ ”, not “the character ‘a’ followed by the character ‘d’. . . ”
  - We don’t mention spaces at all.
- In production compilers, the lexical analyzer typically returns more information, but the simple tokens will do for this problem.

Quick-and-Dirty Tokenizing

- For our simple purposes, we can use a few simple Python routines to do the job.
- For example, suppose all our tokens were separated by whitespace we could use the .split() method on strings to break up the input, after first using the .strip() method to remove any leading or trailing whitespace:

```
>>> " add ( 2 , 2 ) ".strip().split()
['add', '(', '2', ',', '2', ')']
```

- [Gee. How did I find out about these useful methods? What prompted me to go looking?]
- So now, we just need to get a string with everything separated.

Quick-and-Dirty Tokenizing: Adding Blanks

- Since integer literals and words (like 'add' or '+') are not supposed to be next to each other in the syntax, it would suffice to surround any punctuation characters with spaces.

```
def tokenize(line):
    """Convert a string into a list of tokens."""
    spaced = line with spaces around '(', ')', and ','
    return spaced.strip().split()
```

- Option 1: Use the .replace method on strings:

```
spaced =
```

- Option 2: same as Option 1, but use a loop to make it more easily extensible:

- Option 3: Import the package re, and use pattern replacement:

Quick-and-Dirty Tokenizing: Adding Blanks

- Since integer literals and words (like 'add' or '+') are not supposed to be next to each other in the syntax, it would suffice to surround any punctuation characters with spaces.

```
def tokenize(line):
    """Convert a string into a list of tokens."""
    spaced = line with spaces around '(', ')', and ','
    return spaced.strip().split()
```

- Option 1: Use the .replace method on strings:

```
spaced = line.replace('(',' ( ').replace(')',' ) ').replace(',', ' , '){
```

- Option 2: same as Option 1, but use a loop to make it more easily extensible:

- Option 3: Import the package re, and use pattern replacement:

Quick-and-Dirty Tokenizing: Adding Blanks

- Since integer literals and words (like 'add' or '+') are not supposed to be next to each other in the syntax, it would suffice to surround any punctuation characters with spaces.

```
    def tokenize(line):
        """Convert a string into a list of tokens."""
        spaced = line with spaces around '(', ')', and ','
        return spaced.strip().split()
```

- Option 1: Use the .replace method on strings:

```
spaced = line.replace('(',' ( ').replace(')',' ) ').replace(',', ' , '){
```

- Option 2: same as Option 1, but use a loop to make it more easily extensible:

```
punc = "(),"
spaced = line
for c in "(),":
    spaced = spaced.replace(c, ' ' + c + ' ')
```

- Option 3: Import the package re, and use pattern replacement:

Quick-and-Dirty Tokenizing: Adding Blanks

- Since integer literals and words (like 'add' or '+') are not supposed to be next to each other in the syntax, it would suffice to surround any punctuation characters with spaces.

```
    def tokenize(line):
        """Convert a string into a list of tokens."""
        spaced = line with spaces around '(', ')', and ','
        return spaced.strip().split()
```

- Option 1: Use the .replace method on strings:

```
spaced = line.replace('(',' ( ').replace(')',' ) ').replace(',', ' , '){
```

- Option 2: same as Option 1, but use a loop to make it more easily extensible:

```
punc = "(),"
spaced = line
for c in "(),":
    spaced = spaced.replace(c, ' ' + c + ' ')
```

- Option 3: Import the package re, and use pattern replacement:

```
spaced = re.sub(r'([(),])', r' \1 ', line)
```

Syntactic Analysis: Find the Recursion

- Consider the definition of a calculator expression:
  - A numeral, or
  - An operator, followed by a '(', followed by a sequence of *calculator expressions* separated by commas, followed by a right parenthesis.
- The recursion in the definition suggests the recursive structure of our analyzer.
- This particular syntax has two useful properties:
  - By looking at the first token of a calculator expression, we can tell which of the two branches above to take, and
  - By looking at the token immediately after each operand, we can tell when we've come to the end of an operand list.
- That is, we can *predict* on the basis of the next (as-yet unprocessed) token, what we'll find next.
- Allows us to build a *predictive recursive-descent parser* that uses *one token of lookahead.*

Analysis from the Top

- Plan: organize our program into two mutually recursive functions: one for expressions, and one for operand lists.
- Each of these will input a list of tokens and consume (remove) the tokens comprising the expression or list it finds, returning tree(s).

```
def analyze(tokens):
    >>> tokens = [ '+', '(', '1', ',', '3', ')' ]
    >>> analyze(tokens)
    Exp('+', [ Exp(1), Exp(3) ])
    >>> tokens
    []
    >>> tokens = [ '1', ',', '3', ')' ]
    >>> analyze(tokens)
    Exp(1)
    >>> tokens
    [ ',', '3', ')' ]
    """
```

Limitations of Predictive Parsers

- Not all languages lend themselves to predictive parsing.
- Consider the English sentence:

$$\overbrace{\text{The horse raced past the barn}}^{\text{Subject of the sentence}}\ \text{fell.}$$

- This is an example of a *garden-path sentence*:
  - You expect (might reasonably predict) that the subject is “The horse,” and ends just before “raced.”
  - But “raced” here means “that was raced,” which you can’t tell until you get to the last word.
- One can use *backtracking* in this case (like the maze program).
- Requires a different program structure.