Lecture #25: Calculator Example: Parsing

Recap: The strategy.

- *Parsing:* Convert text into expression trees.
- *Evaluation:* Recursively traverse the expression trees calculating a result

`'add(2, 2)'` $\Longrightarrow$ `Exp('add', (2, 2))` $\Longrightarrow$ `4`

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.

```
def calc_parse(line):    # From lect24.py
    """Parse a line of calculator input and return an expression tree."""
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
    return expression_tree
```

Tokens

- Purpose of tokenize is to perform a transformation like this:

```
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

- In principle, we could dispense with this step and go from text to trees directly, but
- We choose 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 'b'..."
  - 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, if 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.
- 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.

Quick-and-Dirty Tokenizing: The Code

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

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

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

```
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):
    """Return the translation of a prefix of 'tokens' that forms a
    calculator expression into a tree, removing the tokens used."""
token = analyze_token(tokens.pop(0))
if type(token) in (int, float):
    return token
else:
    return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    """Assuming that 'tokens' is a comma-separated list of
    expressions surrounded by '(...)', return their translations into a
    list of trees, removing all the tokens thus used."""
    operands = []
    while tokens.pop(0) != ')':
        operands.append(analyze(tokens))
    return operands
```

Detail: Token Coercion

- The analyze_token function converts numerals (text) into Python numbers.
- In actual compilers, this is often done by the lexical analyzer, but the boundary between lexer and parser is moveable.

```
def analyze_token(token):
    """Return the numeric value of token if it can be
    analyzed as a number, and otherwise token."""
    try:
        return int(token)    # Why try this first?
    except ValueError:
        try:
            return float(token)
        except ValueError:
            return token
```

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.

Dealing with Errors

- Code so far has assumed correct input. In real life, one must be less trusting.

```
known_operators = {'add', 'sub', 'mul', 'div', '+', '-', '*', '/'}

def analyze(tokens):
    if not tokens: raise SyntaxError('unexpected end of line')
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    if token in known_operators:
        return Exp(token, analyze_operands(tokens))
    else:
        raise SyntaxError('unexpected ' + token)
```

Dealing with Errors with a Little More Style

- Error-checking code clutters the program, so we might opt for something a bit clearer.

```
def next_token(tokens, allowed):
    if len(tokens) == 0:
        token, name = None, '*ENDLINE*'
    else:
        token = name = analyze_token(tokens.pop())
    if token in allowed or \
       (type(token) in [int, float] and int in allowed):
        return token
    else:
       raise SyntaxError('unexpected token: ' + name)

known_operators = {'add', 'sub', 'mul', 'div', '+', '-', '*', '/', int}

def analyze(tokens):
    token = next_token(tokens, known_operators)
    if type(token) in (int, float):
        return token
    else:
        return Exp(token, analyze_operands(tokens))
```