# Lecture #25: Programming Languages and Programs

- A *programming language,* is a notation for describing computations or processes.
- These range from *low-level* notations, such as machine language or simple hardware description languages, where the subject matter is typically finite bit sequences and primitive operations on them that correspond directly to machine instructions or gates, . . .
- . . . To *high-level* notations, such as Python, in which the subject matter can be objects and operations of arbitrary complexity.
- The universe of implementations of these languages is layered: Python can be implemented in C, which in turn can be implemented in assembly language, which in turn is implemented in machine language, which in turn is implemented with gates, which in turn are implemented with transistors.

# Metalinguistic Abstraction

- We've created abstractions of actions—functions—and of things—classes.
- *Metalinguistic abstraction* refers to the creation of languages—abstracting *description*. Programming languages are one example.
- Programming languages are *effective:* they can be implemented.
- These implementations *interpret* utterances in that language, performing the described computation or controlling the described process.
- The interpreter may be hardware (interpreting machine-language programs) or software (a program called an *interpreter*), or (increasingly common) both.
- To be implemented, though, the grammar and meaning of utterances in the programming language must be defined precisely.

# 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.

# Expression Trees (again)

- Our calculator program represents expressions as trees (see Lecture #12).
- It consists of a *parser*, which produces expression trees from input text, and an *evaluator*, which performs the computations represented by the trees.
- You can use the term *"interpreter"* to refer to both, or to just the evaluator.
- To create an expression tree:

```
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
```

# Expression Trees By Hand

As usual, we have defined (in `lect25.py`) the methods `__repr__` and `__str__` to produce reasonable representations of expression trees:

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

# Evaluation

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

- Primitive expressions (literals) "evaluate to themselves"
- 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 type(exp) in (int, float):
        return exp
    elif type(exp) == Exp:
        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.
    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
```

# 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))
```

# Handling Operands

```
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
```

**Notes:**

- Every trip through the **while** loop splits off an operand.
- The **while** condition has the side effect of removing the next token.
- This token is '(' on the first trip, ',' after each operand but the last, and ')' after the last operand.

# 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 itself."""

    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.

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

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

# Catching Errors in next_token

```
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)
```