Lecture #24: 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.

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 commaseparated 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 #20).
- 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 `lect24.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
```