Today

- Interpretation: Basics
- Calculator Language
- Review: Scheme Lists

Programming Languages

Computer software today is written in a variety of programming languages.

Programming Languages

Computers can only work with 0s and 1s. In machine language, all data are represented as sequences of bits, all statements are primitive instructions (ADD, DIV, JUMP) that can be interpreted directly by hardware.

High-level languages like Python allow a user to specify instructions in a more human-readable language, which eventually gets translated into machine language, are evaluated in software, not hardware, and are built on top of low-level languages, like C.

The details of the 0s and 1s are abstracted away.
STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

THE TREACHERY OF IMAGES
(René Magritte)

INTERPRETATION
What is a kitty?

kitty is a word made from k, i, t and y.

We interpret (or give meaning to) kitty, which is merely a string of characters, as the animal.

Similarly, 5 + 4 is simply a collection of characters: how do we get a computer to “understand” that 5 + 4 is an expression that evaluates (or produces a value of) 9?

INTERPRETATION
An interpreter for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression.

It allows a computer to interpret (or to “understand”) strings of characters as expressions, and to work with resulting values.
INTERPRETATION

Many interpreters will read the input from the user, evaluate the expression, and print the result.

They repeat these three steps until stopped.

This is the read-eval-print loop (REPL).

ANNOUNCEMENTS

• Project 3 is due Thursday, July 26.
• Homework 11 is due Friday, July 27.

Please ask for help if you need to. There is a lot of work in the weeks ahead, so if you are ever confused, consult (in order of preference) your study group and Piazza, your TAs, and Jom.

Don’t be clueless!

ANNOUNCEMENTS: MIDTERM 2

• Midterm 2 is tonight.
   – Where? 2050 VLSB.
   – When? 7PM to 9PM.
• Closed book and closed electronic devices.
• One 8.5” x 11” ‘cheat sheet’ allowed.
• Group portion is 15 minutes long.

THE CALCULATOR LANGUAGE (OR “CALC”)

We will implement an interpreter for a very simple calculator language (or “Calc”):

```
calc> add(3, 4)
7
calc> add(3, mul(4, 5))
23
calc> +(3, *(4, 5), 6)
29
calc> div(3, 0)
ZeroDivisionError: division by zero```

http://berkeley.edu/map/maps/ABCD345.html
THE CALCULATOR LANGUAGE (OR “CALC”)

The interpreter for the calculator language will be written in Python. This is our first example of using one language to write an interpreter for another.

This is not uncommon: this is how interpreters for new programming languages are written.

Our implementation of the Python interpreter was written in C, but there are other implementations in Java, C++, and even Python itself!

SYNTAX AND SEMANTICS OF CALCULATOR

There are two types of expressions:
1. A primitive expression is a number.
2. A call expression is an operator name, followed by a comma-delimited list of operand expressions, in parentheses.

SYNTAX AND SEMANTICS OF CALCULATOR

Only four operators are allowed:
1. add (or +)
2. sub (or -)
3. mul (or *)
4. div (or /)

All of these operators are prefix operators: they are placed before the operands they work on.

READ-EVAL-PRINT LOOP

def read_eval_print_loop():
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except ...:
            # Error-handling code not shown

READ

The function calc_parse reads a line of input as a string and parses it.

Parsing a string involves converting it into something more useful (and easier!) to work with.

Parsing has two stages: tokenization (or lexical analysis), followed by syntactic analysis.
def calc_parse(line):
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
    return expression_tree

Remember that a string is merely a sequence of characters: **tokenization** separates the characters in a string into **tokens**.

As an analogy, for English, we use spaces and other characters (such as . and ,) to separate tokens (or “words”) in a string.

**Tokenization**, or **lexical analysis**, identifies the **symbols** and **delimiters** in a string.

A **symbol** is a sequence of characters with meaning: this can either be a name (or an **identifier**), a literal value, or a reserved word.

A **delimiter** is a sequence of characters that defines the syntactic structure of an expression.

For Calc, we can simply insert spaces and then split at the spaces.

```python
def tokenize(line):
    spaced = line.replace('(', ' ( ')
    spaced = spaced.replace(')', ' ) ')
    spaced = spaced.replace(',', ', ')
    return spaced.strip().split()
```

Removes leading and trailing white spaces. Returns a list of strings separated by white spaces.

Now that we have the tokens, we need to understand the structure of the expression: What is the operator? What are its operands?

As an analog, in English, we need to determine the grammatical structure of the sentence before we can understand it.
For English (and human languages in general), this can get complicated and tricky quickly:

Jom lectures to the student in the class with the cat.
The horse raced past the barn fell.
Buffalo buffalo Buffalo buffalo buffalo.

For Calc, the structure is much easier to determine.

Expressions in Calc can nest other expressions:

```
add(3, mul(4, 5))
```

Calc expressions establish a hierarchy between operators, operands, and nested expressions.

What is a useful data structure to represent hierarchical data?

```
class Exp:
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
```

An expression tree is a hierarchical data structure that represents a Calc expression.

```
Exp('add', [3, Exp('mul', [4, 5])])
```
PARSING: SYNTACTIC ANALYSIS

The function `analyze` takes in a list of tokens and constructs the expression tree as an `Exp` object.

It also changes the tokens that represent integers or floats to the corresponding values.

```python
>>> analyze(tokenize('add(3, mul(4, 5))))
Exp('add', [3, Exp('mul', [4, 5])])
```

(We will not go over the code for `analyze`.)

---

READ-EVAL-PRINT LOOP

```python
def read_eval_print_loop():
    while True:
        try:
            expression_tree = calc_parse(input('calc> '))
            print(calc_eval(expression_tree))
        except ...
            # Error-handling code not shown
```

---

EVALUATION: RULES

- Primitive expressions (literals) are evaluated **directly**.
- Call expressions are evaluated **recursively**:
  - **Evaluate** each operand expression.
  - Collect their values as a list of arguments.
  - **Apply** the named operator to the argument list.

---

EVALUATION

`Evaluation` finds the value of an expression, using its corresponding expression tree.

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

```python
def calc_eval(exp):
    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)
```

---

CASE 1: If the expression is a number, then:
`Numbers are self-evaluating` they are their own values.

CASE 2: Otherwise, evaluate the arguments:
```python
1. if type(exp) in (int, float):
   return exp
2. elif type(exp) == Exp:
   arguments = list(map(calc_eval, exp.operands))
   return calc_apply(exp.operator, arguments)
3. evaluate the values in a list
4. ... and then apply the operator on the argument values.
```
EVALUATION

Why do we need to evaluate the arguments? Some of them may be nested expressions!

We need to know what we are operating with (operator) and the values of all that we are operating on (operands), before we can completely evaluate an expression.

APPLY

def calc_apply(operator, args):
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 1:
            return -args[0]
        return sum(args[:1] + \[-arg for arg in args[1:]]
    ...
    ... and so on, for all of the operators that we want Calc to understand.

It may seem odd that we are using Python’s addition operator to perform addition.

Remember, however, that we are interpreting an expression in another language. As we interpret the expression, we realize that we need to add numbers together.

The only way we know how to add is to use Python’s addition operator.

EVAL AND APPLY

Notice that calc_eval calls calc_apply, but before calc_apply can apply the operator, it needs to evaluate the operands. We do this by calling calc_eval on each of the operands.

calc_eval calls calc_apply,
which itself calls calc_eval.

Mutual recursion!

EVAL AND APPLY

The eval-apply cycle is essential to the evaluation of an expression, and thus to the interpretation of many computer languages. It is not specific to calc.

eval1 receives the expression (and in some interpreters, the environment) and returns a function and arguments; apply applies the function on its arguments and returns another expression, which can be a value.

READ-EVAL-PRINT LOOP: WE ARE DONE!

def read_eval_print_loop():
    while True:
        try:
            expression_tree = \calc_parse(input('calc> '))
lcalc
        print(calc_eval(expression_tree))
except ...:
    # Error-handling code not shown
THREE MORE BUILTIN FUNCTIONS

str(obj) returns the string representation of an object obj.
It merely calls the __str__ method on the object:
str(obj) is equivalent to obj.__str__()

eval(str) evaluates the string provided, as if it were an expression typed at the Python interpreter.

THREE MORE BUILTIN FUNCTIONS

repr(obj) returns the canonical string representation of the object obj.
It merely calls the __repr__ method on the object:
repr(obj) is equivalent to obj.__repr__()

If the __repr__ method is correctly implemented, then it will return a string, which contains an expression that, when evaluated, will return a new object equal to the current object.

In other words, for many classes and objects (including Exp),
eval(repr(obj)) == obj.

CONCLUSION

• An interpreter is a function that, when applied to an expression, performs the actions required to evaluate that expression.
• We saw how to implement an interpreter for a simple calculator language, Calc.
• The read-eval-print loop reads user input, evaluates the expression in the input, and prints the resulting value.
• Reading also involves parsing user input into an expression tree.
• Evaluation works on the provided expression tree to obtain the value of the corresponding expression.
• Preview: An interpreter for Python, written in Python.

GOOD LUCK TONIGHT!