Lecture 21: Interpreters

CS 61A - Summer 2024
Raymond Tan

Exceptions

Raise Statements

- Python exceptions are raised with a raise statement
  - `raise <expression>`
- `<expression>` must evaluate to a subclass of BaseException
- Common error types:
  - TypeError -- A function was passed the wrong number/type of argument
  - NameError -- A name wasn't found
  - KeyError -- A key wasn't found in a dictionary
  - RecursionError -- Too many recursive calls

Raise Statement - Example

```python
def add_positive_numbers(x, y):
    """Takes in two positive numbers x and y and sums them together."""
    if x <= 0 or y <= 0:
        raise TypeError('Arguments should be positive!')
    return x + y
```

Try Statements

- Try statements are a way that we can ***handle*** exceptions
- Syntax:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

Try Statement - Example

```
def divide(x, y):
    try:
        ans = x / y
    except ZeroDivisionError as e:
        print('handling a', type(e))
        ans = 0
    return ans
```

Try Statements

- Execution rule:
    - The <try suite> is executed first
    - If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and
    - If the class of the exception inherits from <exception class>, then
    - The <except suite> is executed, with <name> bound to the exception

Programming Languages

Programming Languages

- A computer typically executes programs written in many different programming languages
- **Machine languages**: statements are interpreted by the hardware itself
  - A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
  - Operations refer to specific hardware memory addresses; no abstraction mechanisms
- **High-level languages**: statements & expressions are interpreted by another program or compiled (translated) into another language
  - Provide means of abstraction such as naming, function definition, and objects
  - Abstract away system details to be independent of hardware and operating system

Machine and High-Level Languages

- Example of a Machine/Assembly-Level Language: RISC-V

```
loop:   slli   s2, s1, 2
        add    s3, s0, s2
        lw     t1, 0(s3)
        mul    t2, t1, t1
        add    t3, t2, t1
        sw     t3, 0(s3)
        addi   s1, s1, 1
        blt    s1, t0, loop
```

- Example of a High-Level Language: Python

```python
def sum_list(lst):
    total = 0
    for x in lst:
        total += x
    return total
```

Metalinguistic Abstraction

- A powerful form of abstraction is to define a new language that is tailored to a particular type of application or problem domain
    - This is called “Metalinguistic Abstraction”
- Examples:
    - **Go** was designed for concurrent programs. It has built-in elements for expressing concurrent routines. It is used, for example, to implement chat servers, livestreams, and multiplayer game servers, with many simultaneous connections.
    - **LaTeX** was designed for generating static technical & scientific documentation. It has built-in elements for text formatting, mathematical expressions, and code blocks. It is used, for example, to write research papers.

Programming Languages - Definitions

- A programming language has:
  - **Syntax**: The legal statements and expressions in the language
  - **Semantics**: The execution/evaluation rule for those statements and expressions
- To create a new programming language, you either need a:
  - **Specification**: A document describe the precise syntax and semantics of the language
  - **Canonical Implementation**: An interpreter or compiler for the language

Parsing

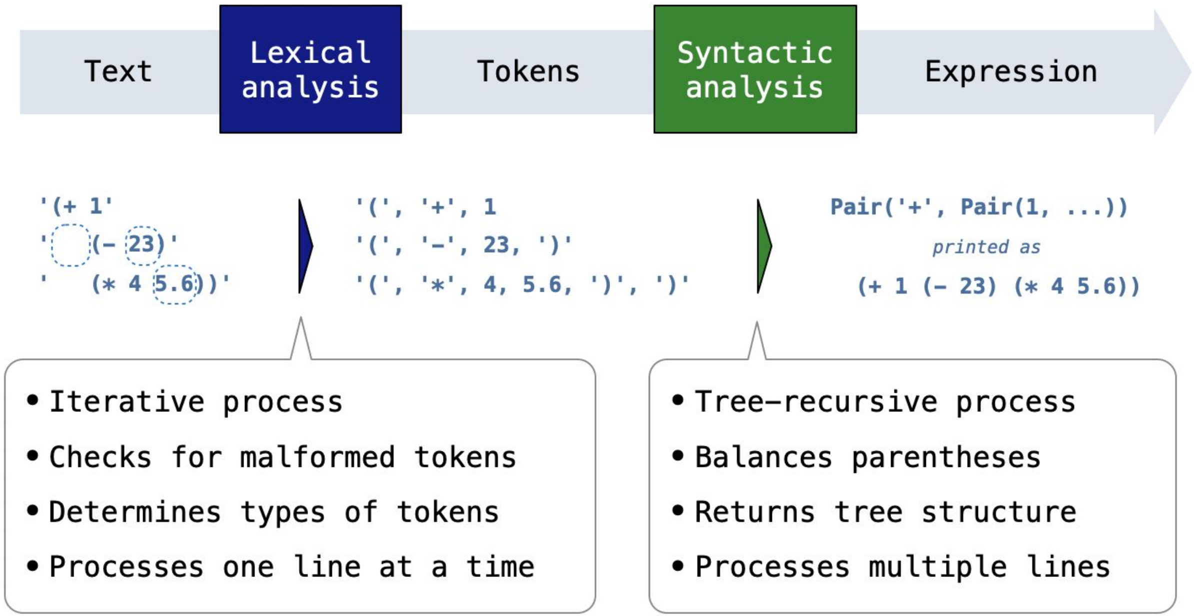
Reading Scheme Lists

- A Scheme list is written as elements in parentheses:
    - `(<element_0> <element_1> ... <element_n>)`
- Each <element> can be a combination or primitive
    - `(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))`
- The task of parsing a language involves coercing a string representation of an expression to the expression itself

Demo: Reading Scheme Lists

Parsing

- A parser takes text (represented as a String) and returns an ***expression***



Syntactic Analysis

- Syntactic analysis identifies the hierarchical structure of an expression, which may be nested
  - Each call to scheme_read consumes the input tokens for exactly one expression
- Base case: symbols and numbers
- Recursive call: scheme_read sub-expressions and combine them

Break

Scheme-Syntax Calculator

Calculator Syntax

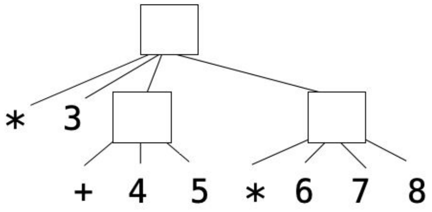
- The Calculator language has primitive expressions and call expressions. (That's it!)
- A primitive expression is a number: 2   -4   5.6
- A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3)   (/ 3 (+ 4 5))
- Expressions are represented as Scheme lists (Pair instances) that encode tree structures.

Calculator Syntax - Visualization

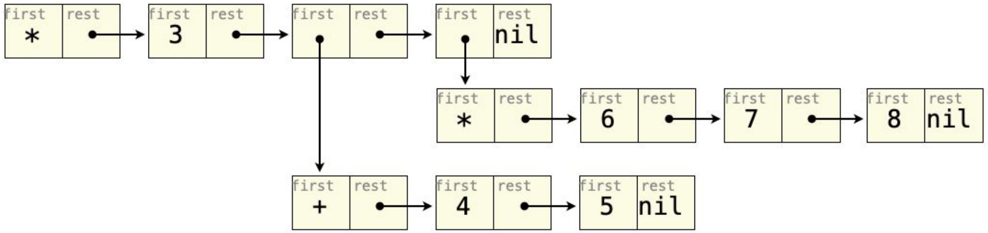
Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Representation as Pairs



Calculator Semantics

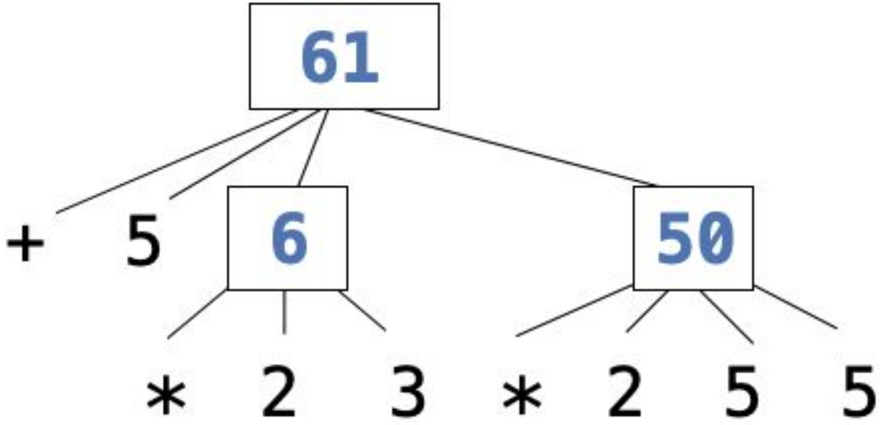
- The value of a calculator expression is defined recursively
  - **Primitive**: A number evaluates to itself.
  - **Call**: A call expression evaluates to its argument values combined by an operator.
- Here are the operators we'll be supporting in the Scheme Calculator Language:
  - **+**: Sum of the arguments
  - *****: Product of the arguments
  - **-**: If one argument, negate it. If more than one, subtract the rest from the first.
  - **/**: If one argument, invert it. If more than one, divide the rest from the first.

Calculator Semantics Visualization

Expression

```
(+ 5
   (* 2 3)
   (* 2 5 5))
```

Expression Tree



Implementation of the Scheme-Calculator

- You'll be implementing a more advanced version of the Scheme-Calculator fully in lab **next Monday (Lab 10)**!

calc_eval and calc_apply for lab

```
def calc_apply(op, args):
    return op(args)
```

- calc_eval takes in an expression, and evaluates it as Calculator code
- calc_apply simply takes in an operator and arguments, and applies the operator onto the arguments

```
def calc_eval(exp):
    """
    >>> calc_eval(Pair("define", Pair("a", Pair(1, nil))))
    'a'
    >>> calc_eval("a")
    1
    >>> calc_eval(Pair("+", Pair(1, Pair(2, nil))))
    3
    """
    if isinstance(exp, Pair):
        operator = ____________ # UPDATE THIS FOR Q2
        operands = ____________ # UPDATE THIS FOR Q2
        if operator == 'and': # and expressions
            return eval_and(operands)
        elif operator == 'define': # define expressions
            return eval_define(operands)
        else: # Call expressions
            return calc_apply(___________, ___________) # UPDATE THIS FOR Q2
    elif exp in OPERATORS:   # Looking up procedures
        return OPERATORS[exp]
    elif isinstance(exp, int) or isinstance(exp, bool):   # Numbers and booleans
        return exp
    elif _________________: # CHANGE THIS CONDITION FOR Q4
        return _________________ # UPDATE THIS FOR Q4
```

Interactive Interpreters

Read-Eval-Print Loop (REPL)

- The user interface for many programming languages is an interactive interpreter that follows these steps:
    - Print a prompt
    - Read text input from the user
    - Parse the text input into an expression
    - Evaluate the expression
    - If any errors occur, report those errors, otherwise
    - Print the value of the expression and repeat

Raising Exceptions

- Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply
- Example exceptions
  - Lexical analysis: The token 2.3.4 raises ValueError("invalid numeral")
  - Syntactic analysis: An extra) raises SyntaxError("unexpected token")
  - Eval: An empty combination raises TypeError("() is not a number or call expression")
  - Apply: No arguments to - raises TypeError("- requires at least 1 argument”)

Handling Exceptions

- An interactive interpreter prints information about each error
- A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment

Summary

- We can raise our own exceptions in Python through the raise statement, and handle exceptions with the try/except statement
- Interpreters go through lexical and syntactic analysis to parse code
    - Lexical analysis breaks text into tokens, and syntactic analysis takes the tokens and converts to expressions
- The REPL is an interactive interpreter designed to help test and understand a programming language