CS61A Notes

Aaron Guo

Fall 2022

Please find the course notes for CS61A Structure and Interpretation of Computer Programs below, closely following the Fall 2022 asynchronous lecture videos by Professors John Denero and Justin Yokota. A special thank-you to Simon Kuang (simontheflutist@berkeley.edu) for the template.

Contents

1 Functions and Expressions

1.1 Expressions

- Call expression anatomy: <Operator>(<Operands>)
- Call expressions evaluate to *values*
- Order of evaluation:
  - Evaluate operator
  - Evaluate operands
  - Apply operator function onto operand values
- Primitive expressions
  - Number: 2
  - Name: add
  - String: 'hello'

1.2 Environment Diagrams

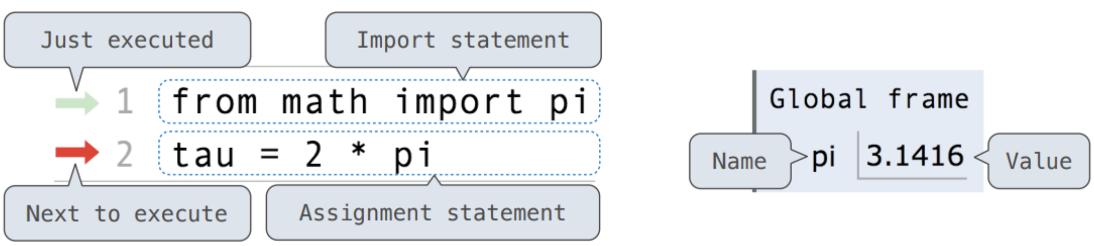

Figure 1: Example Environment Diagram

- Visualization fo the interpreter's process
- Left side: statements and expressions
  - Arrows indicate evaluation order
- Right side: names bounded to values in frames
  - Names cannot be repeated in the same frame
  - An environment in a *sequence of frames*: a frame, its parent frame, and so on
  - A name evaluates to the value bound to the name in the *earliest frame* of the current environment
  - Look in the local frame, then in the parent frame, and so on until the global frame
  - Every expression is evaluated in the context of its environment
- Assignment statements: right side is evaluated, and resulting value(s) are binded to the name(s) on the left side

1.3 Functions

- Function signature anatomy: <name>(<formal parameters>)
- Execution order for applying user-defined functions:
    1. Create local frame
    2. Bind formal parameters to arguments
    3. Execute the function body in the local frame
- None represents nothing
- The return statement completes the evaluation of the call expression
- A function that does not explicitly return a value returns None
- Pure function: just returns a value
- Non-pure function: has side effects

1.4 Statements

- A statement is *executed by the interpreter to perform an action*

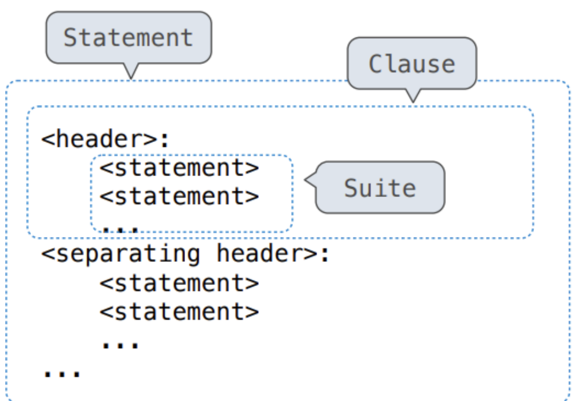

Figure 2: Compound Statement Anatomy

- Conditional statement: if clause, followed by elif/else clauses
- Execution:
    1. Evaluate header expression
    2. If true, execute the suite and skip the remaining clauses
- False values in Python: False, 0, ”, None
- True values are anything else
- While statements (iteration)
- Execution:
    1. Evaluate header expression
    2. If true, execute the suite and return to step 1

2 More Functions

2.1 Functions

- Function domain: the set of all inputs that are possible for arguments
- Function range: the set of all output values that are possible to return
- Function behavior: the relationship between input and output
- A function is an abstraction for its effect, behavior, or return value
- Functions are a type of value
- Higher-order function: a function that takes a function as an argument or returns a function
- Higher-order functions can be described with environment diagrams
- Self-reference: returning a function using its own name
- Parent frame of a function: the frame in which the function was *defined*

2.2 Lambda Expressions

- Lambda expression: an expression that evaluates to a function
- Anatomy: lambda <arguments>: <single expression that evaluates to return value>
- Lambda expressions cannot contain statements
- "def" vs. "lambda": only def gives the function an intrinsic name in environment diagrams (doesn't affect execution)
- Function currying: transforming a multi-argument function into a single-argument, higher-order function

2.3 Logical Operators

- Evaluation of <left> and <right>:
  1. Evaluate <left>
  2. If the result is False, then the whole expression evaluates to False
  3. Otherwise, the whole expression evaluates to <right>
- Evaluation of <left> or <right>:
  1. Evaluate <left>
  2. If the result is True, then the whole expression evaluates to Talse
  3. Otherwise, the whole expression evaluates to <right>

2.4 Errors

- Syntax errors: detected by Python interpreter before execution
- Runtime errors: detected by Python interpreter during execution
- Logic & behavior errors: Not detected by the Python interpreter

2.5 Recursion

- Recursive function: a function that calls itself, directly or indirectly
- Usage: solving problems that *have smaller instances of the same problem*
- Anatomy:
  - "def" statement header
  - Conditional statements that check for *base cases*
  - Base cases are evaluated *without recursive calls*
  - Recursive cases (all other cases) are evaluated *with recursive calls*
- The recursive leap of faith: assumption that the smaller case is solved correctly to solve the current case
- Iteration is a special case of recursion
- Tree recusion: a recursive function makes more than one recursive call

3 Sequences

3.1 Lists and Containers

- List: a compound value (sequence of values)
- A method of combining data values satisfies the *closure property* if the result of the combination can itself be combined using the same method
- i.e. Lists can contain lists as elements

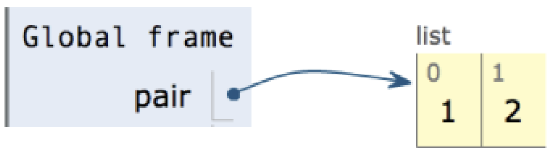

Figure 3: List Format in Environment Diagrams

3.2 For Statements

- Anatomy: for <name> in <expression>:
- Execution:
  1. evaluate <expression>, which must yield an iterable value (sequence)
  2. For each element in that sequence in order:
     a) Bind <name> to that element in the current frame
     b) Execute the suite

3.3 Ranges

- Range: a sequence of consecutive integers
- Starting value (inclusive) to enging value (exclusive)
- Length: ending value - starting value

3.4 List Comprehensions

- Combined expression that evaluates to a list
- Anatomy: [<map exp> for <name> in <iter exp> if <filter exp>]
- Anatomy without filter: [<map exp> for <name> in <iter exp>]
- For each element in <iter exp> if <filter exp> is True, then add <map exp>

3.5 Slicing

- Slicing a list creates new values, does not reference original list values
- Anatomy: <new list name> = <list name>[<start index>:<end index>]
- If no explicit start or end index, the beginning or end of the list is used
- Start index is inclusive, end index is exclusive

3.6 Aggregation

- Built-in functions:
- sum(iterable[, start]) -> value
- Returns the sum of an iterable + start (default start value is 0)
- max(iterable[, key=func]) -> value
- Returns the largest item in the iterable, based on key function (default is no function)
- all(iterable) -> value
- Returns True if all values in the iterable are True (including empty iterable), False otherwise

3.7 Strings

- String literabls are surrounded with single or double quotes
- A backslash "\" escapes the next character
- Line feed character "\n" represents a new line

3.8 Dictionaries

- Dictionary: a collection of key-value pairs
- Keys must be unique within the same dictionary
- Keys cannot be a list or dictionary
- Dictionary comprehension: {<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

4 Abstraction

4.1 Data Abstraction

- Data abstrction: a methodology by which functions enforce an abstraction barrier between *representaion* and *use*
- Identify a basic set of operations in which all manipulations of a data type can be expressed, then use only those operations for manipulating that data
- Abstraction barriers separate stages of an implementation into levels of abstraction
- Example - rational numbers:

Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	add_rational, mul_rational rationals_are_equal, print_rational
Create rationals or implement rational operations	numerators and denominators	rational, numer, denom
Implement selectors and constructor for rationals	two-element lists	list literals and element selection

Implementation of lists

Figure 4: List Format in Environment Diagrams

5 Mutability

5.1 Objects

- Objects represent *information* and consist of attributes - *data* and *behavior*
- A type of onject is called a *class*
- Object-oriented programming: a methodology for organizing larger programs by:
  - Using data abstraction
  - Bundling together information and related behavior (objects)
- Functions do one thing, objects do *many related things*
- Mutability: the ability for an object to change

5.2 Tuples

- Tuple: an immutable sequence
- An immutable sequence can still be changed if it *contains* a mutable value as an element

5.3 Identity Operators

- Identity: <exp0> is <exp1>
- True if both <exp0> and <exp1> evaluate to the *same object*
- Equality: <exp0> == <exp1>
- True if both <exp0> and <exp1> evaluate to *equal values*

5.4 Files, Strings, and Lists

- .strip() : returns a string without whitespace on the ends
- .split() : returns a list of strings that were separated by whitespace
- .replace(a, b) : returns a string with all instances of string a replaced by string b

6 Iterators and Generators

6.1 Iterators

- Any container can provide an iterator that provides elements in Order
- iter(iterable) : returns an iterator
- next(iterator) : returns the next element in an iterator
- For dictionaries, the order of items (key-value pairs) is the order in which they were added
- Iterators make few assumptions about the data, so others are more likely to be able to use your code on their data
- Iterators keep track of position within the sequence, ensuring each element is only processed once

6.2 Built-In Functions for Iteration

- map(func, iterable) : iterate over x in iterable using func(x)
- filter(func, iterable) : iterate over x in iterable if func(x)
- zip(iter1, iter2): iterate over co-indexed (x, y) pairs,
  - Skips extras if iterables are different length
  - Can take more than two lists as arguments
- reversed(sequence) : iterate over x in a sequence in reverse order
- Functions to view the contents of an iterator:
- list(iterable) : return a list containing all x in iterable
- tuple(iterable) : return a tuple containing all x in iterable
- sorted(iterable) : return a sorted list containing all x in iterable

6.3 Generators

- Generator: a function that *yields* values instead of returning them
- A generator can yield multiple times
- A generator is an iterator that is created by calling a *generator function*
- "yield from" statement yields all values from an iterator/iterable

7 Objects

7.1 Classes

```
class <name>:
    <suite>
```

- Class: a type/category of objects
- Objects are created by calling Classes
- Classes are objects as well
- Every object that is an instance of a user-defined class has a unique identity
- "is" and "is not" test if two expressions evaluate to the same object
- Binding an object to a new name using assignment does not create a new object
- Methods are functions definined in the suite of a class statement
- Defining methods:

```
class <class name>:
    def <method name>(self, <formal parameters>):
        <suite>
```

- Dot expressions: <exp>.<name>
  - <exp> must evaluate to an object
  - If <name> is a method, then "self" parameter is automatically supplied
  - Evaluation order:
    1. Evaluate <exp>
    2. <name> is looked up in the instance attributes of that object
    3. If not found, <name> is looked up in the class
    4. If the value is a function, a bound method is returned. If not, the value is returned
- Looking up an attribute using a string: getattr(<object expression>, "<name>")
- Instance attributes are attributes of a specific object instance
- Class attributes are shared across all instances of a class
- Assignment using dot expressions: <exp1>.<name> = <exp2>
  - The value of <exp2> is binded to the attribute <name> found in the evaluated <exp1> object

7.2 Inheritance

- Inheritance: a technique for relating classes together
- Often used for specialization

```
class <class name>(<base class>):
    <suite>
```

- Subclass inherits attributes of its base class
- Certain inherited attributes may be overridden

- Base class attributes are not copied into subclasses
- Inheritance is best for representing *is-a relationships*
- Composition is best for representing *has-a relationships*
- A class may inherit from multiple base classes in Python

7.3 Representation

- All objects have two forms of string representations
- "str" is legible to humans (same value as what is printed with "print" function)
- "repr" is legible to the Python interpreter
- For most object types, eval(repr(object)) == object
- F-Strings for string interpolation
  - String interpolation: evaluating a string literal that contains expressions
  - Equivalent examples:
  - String concatenation: 'pi starts with ' + str(pi) + '...'
  - String interpolation: f'pi starts with {pi}...'
  - Equivalent output: 'pi starts with 3.141592653589793...'

7.4 Polymorphic Functions

- Polymorphic function: a function that applies to many different forms of data
- Examples: "str" and "repr"

7.5 Interfaces

- Interface: a set of shared messages, along with a specification of what they management
- Example: classes that implement __repr__ and __str__ methods that return string representations implement an interface for producing string representations

7.6 Special Method Names in Python

- __init__ : method invoked automatically when an object is construction
- __repr__ : method invoked to display an object as a Python expression
- __add__ : method invoked to add one object to another
- __bool__ : method invoked to convert an object to True or False
- __float__ : method invoked to convert an object to a float

7.7 Generic Functions

- A polymorphic function may take arguments that vary in types
- Type dispatching: inspect the type of an argument to select behavior
- Type coercion: convert one value to math the type of another

8 Composition

8.1 Linked Lists

- Linked list structure: a linked list is either empty or a first value and the rest of the linked list
- Linked lists are mutable

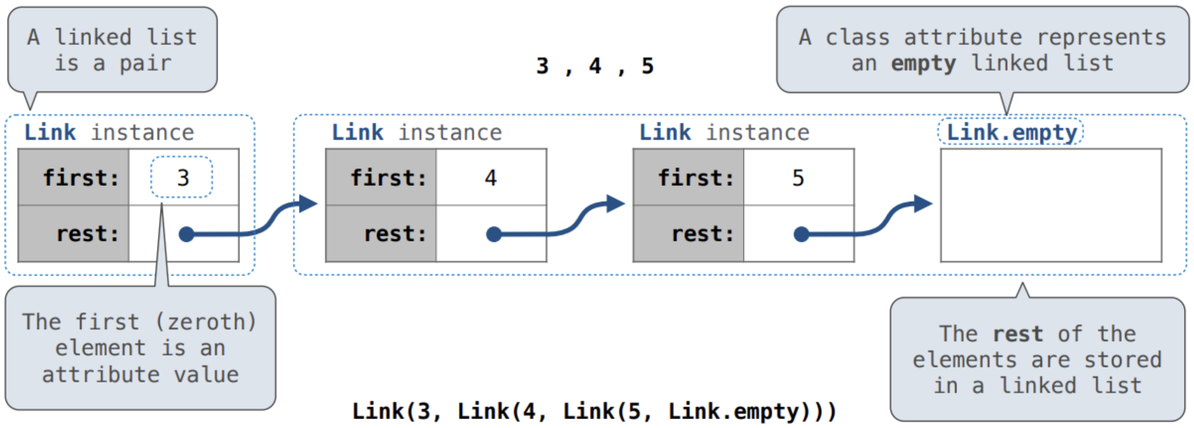

Figure 5: Linked List Example

```
class Link:
    empty = () # some zero-length sequence
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link) # verify rest is a linked list
        self.first = first
        self.rest = rest
```

8.2 Trees

- Recursive description of trees
  - A tree has a root label and a list of branches
  - Each branch is a tree
  - A tree with zero branches is called a leaf
  - A tree starts at the root
- Relative description of trees
  - Each location in a tree is called a node
  - Each node has a label that can be any value
  - One node can be the parent/child of another
  - The top node is the root node
- Tree processing often uses recursion, with the leaf as a base case
- Example tree implementation:

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
        for branch in branches:
    def tree(label, branches=[]):
        for branch in branches:
            assert is_tree(branch)
        return [label] + list(branches)
    def label(tree):
        return tree[0]
    def branches(tree):
        return tree[1:]
```

- Pruning: removing subtrees from a tree
- Pruning can be used before recursive processing to speed up computation

8.3 Modular Design

- A design principle: isolate different parts of a program that address different concerns
- Each modular component can be developed and tested independently
- Example:

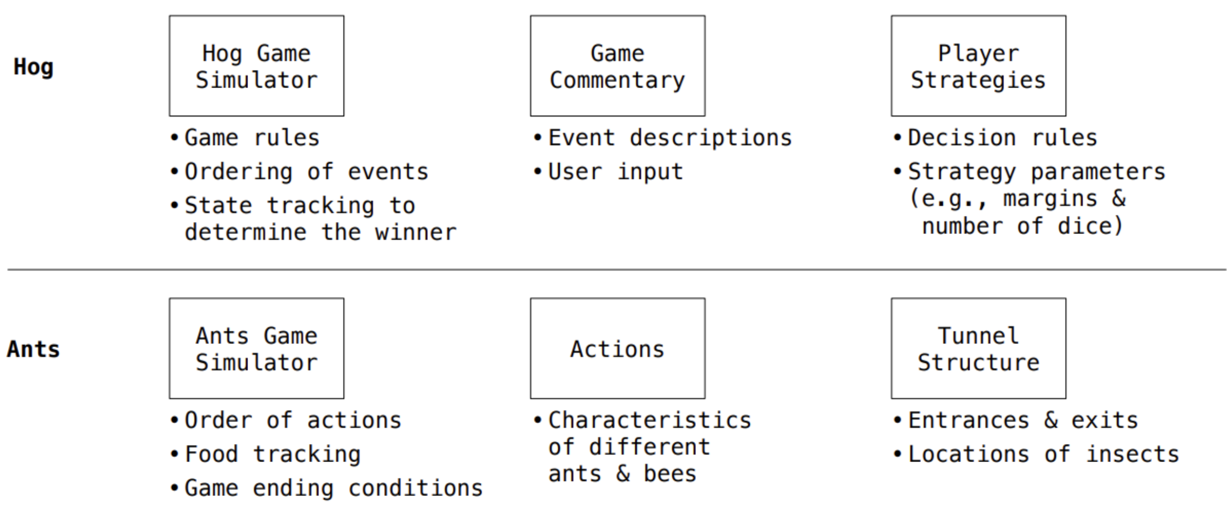

Figure 6: Modular Design Example

9 Efficiency

9.1 Memoization

- Memoization: "remembering" the results that have been computed before
- Example - fibonacci numbers:
- Store the n^{th} fibonacci number in a cache so that its value can be used to compute larger fibonacci numbers later

9.2 Exponentiation

- Taking advantage of repeated multiplication
- One more multiplication allows the problem size to be doubled
- Example:

```
# doubling imput doubles the time
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

# doubling the input increases the time by one step
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

def square(x):
    return x * x
```

9.3 Orders of Growth

Constant growth - $\Theta(1), O(1)$**:** increasing n doesn't increase time

Logarithmic growth - $\Theta(\log n), O(\log n)$**:** doubling n increases time by a constant

Linear growth - $\Theta(n), O(n)$**:** incrementing n increases time by a constant

Quadratic growth - $\Theta(n^2), O(n^2)$**:** incrementing n increases time by n * constant

- Example: functions that process all *pairs* of values in a sequence

Exponential growth - $\Theta(b^n), O(b^n)$**:** incrementing n multiplies time by a constant

- Example: tree-recursive functions (fibonacci without memoization)

9.4 Space

- Active environments:
- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

10 Data Examples

10.1 Lists in Lists in Environment Diagrams

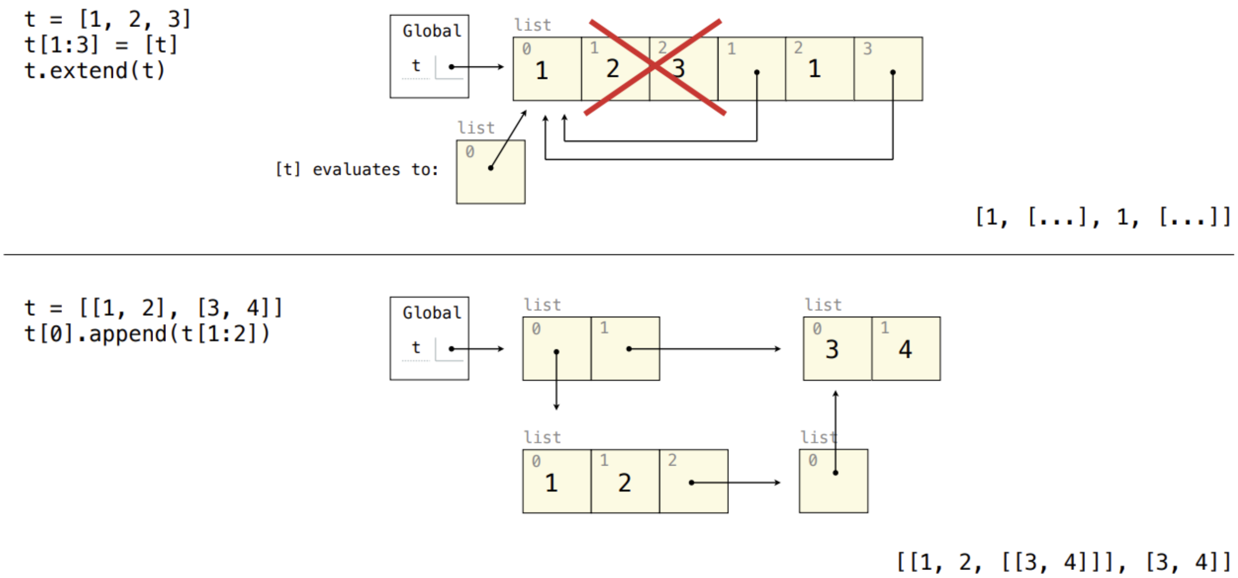

Figure 7: Example of Lists within Lists in an Environment Diagram

10.2 Objects

11 Scheme

11.1 Scheme

- Scheme is a dialect of Lisp
- Scheme programs consist of expressions:
  - Primitive expressions: 2, 3.3, true, +, quotient
  - Combinations: (quotient 10 2), (not true)
- Call expressions incude an operator and 0 or more operands in parentheses
- Special forms: combinations that are not call expressions
- Examples: **if**, **and**, **or**, binding symbols, defining procedures, **cond**, **begin**

```
(if <predicate> <consequent> <alternative>)

(and <exp1> ... <expn>)

(or <exp1> ... <expn>)

(define <symbol> <expression>)

(define (<symbol> <formal parameters>) <body>)

(cond (<predicate 1> <consequent 1>)
      (<predicate 2> (begin <consequent 2a> ... <consequent 2m>))
      ...
      (else <consequent n>))
```

- Lambda expressions:

```
(lambda (<formal parameters>) <body>)
```

- Let expression: binds symbols to values just for one expression

```
(let <symbol> <expression>)
```

11.2 Scheme Lists

- Scheme lists are similar to Python linked-lists

  cons: two-argument procedure that creates a linked list
  car: procedure that returns the first element of a list
  cdr: procedure that returns the rest of a list
  nil: the empty list

- Example list:

```
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
> (define x (cons 1 (cons 2 nil)))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
```

11.3 Symbolic Programming

- Symbols normally refer to values, single quote is used to refer to symbols directly

```
> (define a 1)
> (define b 2)
> (list 'a b)
(a 2)

> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

11.4 Built-in List Processing Procedures

(append s t): list the elements of s and t (can be called on more than two lists)

(map f s): call a procedure f on each element of a list s and list results

(filter f s): call a procedure f on each element of a list s and list the elements for which true is the result

(apply f s): call a procedure f using elements of list s as its arguments

12 Exceptions

12.1 Raise

- Raise statements raise an exception in Python

```
raise <expression>
```

- <expression must evaluate to a subclass or instance of BaseException

  – **TypeError:** incorrect type of argument was passed into a function
  – **NameError:** a name wasn't found
  – **KeyError:** a key wasn't found in a Dictionary
  – **RecursionError:** too many recursive calls

12.2 Try

- Try statements handle Exceptions

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

- Execution order:

  1. <try suite> is executed first
  2. If, in the try suite, an exception is raised that is not handled otherwise and if the class exception inherits from <exception class>, then
  3. <except suite> is executed, with <name> bound to the exception

13 Programming Languages

13.1 Programming Languages

Machine languages: interpreted by the hardware itself

High-level languages: interpreted by another program or compiled into another language

- Abstracts away system details
- Creates independence from hardware and operating system

Syntax: legal statements and expressions in the language

Semantics: execution/evaluation rules for those statements and expressions

13.2 Parsing

- Parsing: turning text into an expression
- Syntactic analysis: identifying the hierarchical structure of an expression
- Evaluation: the computation of the value of an expression

13.3 Interpreters

- Programs specify the logic of a computational device
- An interpreter is a *general computing machine*

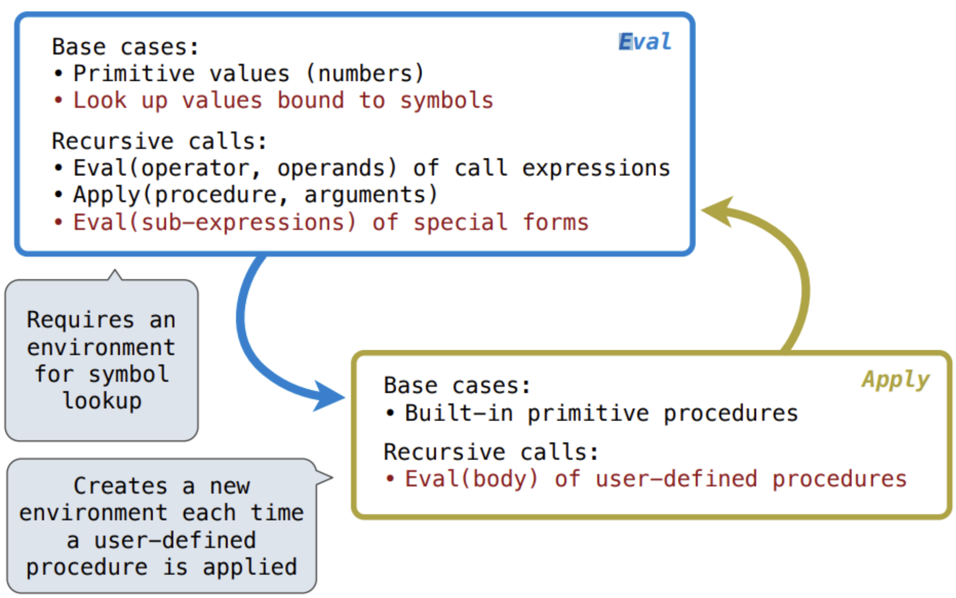

Figure 8: Interpreter Structure

14 Tail Recursion

14.1 Dynamic Scope

Lexical scope: the parent of a frame is the environment in which a procedure was *defined*

```
(define <symbol> (lambda <formal parameters> <body>))
```

Dynamic scope: the parent of a frame is the environment in which a procedure was *called*

```
(define <symbol> (mu <formal parameters> <body>))
```

14.2 Tail Recursion

- Referential transparency: the value of an expression does not change when we substitute one of its subexpressions with the value of that subexpression

- Tail recursion eliminates the "middleman" frames to save space

- Tail call: a call expression in a tail context

  – The last sub-expression in a **lambda** expression's body
  – Sub-expressions <consequent> and <alternative> in an **if** expression that is in a tail context
  – All non-predicate sub-expressions in a tail context **cond**
  – The last sub-expression in a tail context **and, or, begin,** or **let**

14.3 Macros

- Macro: an operation performed on the source code of a program before evaluation

- Scheme example:

```
> (define-macro (twice expr) (list 'begin expr expr))
> (twice (print 2))
2
2
```

15 SQL

15.1 Databases

- Table: a collection of rows that have a value for each column
- SQL is a *declarative* programming languages
  - **Declarative language (SQL, Prolog):** a program is a description of the desired result, and the program figures out how to generate the result
  - **Imperative language (Python, Scheme):** a program is a description of computational processes that the interpreter carries out

15.2 SQL

- A **select** statement creates a new table, either from scratch bor by projecting a table
  - Always includes a comma-separated list of column descriptions

```
select [expression] as [name], [expression] as [name]; ...
select [columns] from [table] where [condition] order by [order];
```

- A **create table** statement gives a global name to a table

```
create table [name] as [select statement];
```

- Two or more tables can be joined together

```
select parent from parents, dogs where child = name and fur = "curly"
```

15.3 Aliases and Dot Expressions

- Aliases and dot expressions clear up ambiguity with column names
- Example of using aliases and dot expressions when joining a table with itself:

```
select a.child as first, b.child as second
    from parents as a, parents as b
    where a.parent = b.parent and a.child < b.child;
```

- Other statements: **analyze, delete, explain, insert, replace, update,** etc.
- Expressions can contain function calls and arithmetic Operators
- String values can be combined to form longer strings

```
> select "hello," || " world";
hello, world
```

15.4 Aggregate Functions

- An aggregate function in the [columns] clause computes a value from a group of rows

```
select [columns] from [table] where [condition] order by [order];
```

- Aggregate functions: **max, min, avg**

15.5 Grouping Rows

- Rows in a table can be grouped, then aggregation can be performed on each group
- Number of groups is the number of unique values of an expression
- A **having** clause filters the set of groups that are aggregated

```
select [columns] from [table] group by [expression] having [filter expression]
```