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.

```
from dis import dis
dis(square)
def square(x):
    return x * x
```

Python 3

**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.

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

Expression Tree

Representation as Pairs

The Pair class represents Scheme pairs and lists. A list is a pair whose second element is either a list or nil.

```
class Pair:
    """A Pair has two instance attributes:
    first and second.
    For a pair to be well-formed, second is either a well-formed list or nil.
    Some methods only apply to well-formed lists."

def __init__(self, first, second):
    self.first = first
    self.second = second

>>> a = Pair(1, Pair(2, Pair(3, nil)))
>>> print(a)
(1 2 3)
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
```

Scheme expressions are represented as Scheme lists! Source code is data.
**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.

- `+`: 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.

```
9 (* 3 (+ 4 5) (* 6 7 8))
```

**Evaluation**

The eval function computes the value of an expression, which is always a number. It is a generic function that dispatches on the type of the expression (primitive or call).

```
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

A number evaluates...

to itself

A call expression evaluates...

to its argument values combined by an operator

**Implementing Built-in Operators**

The apply function applies some operation to a (Scheme) list of argument values.

In calculator, all operations are named by built-in operators: `+`, `-`, `*`, `/`

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

**Interactive Interpreters**

**Read-Eval-Print Loop**

The user interface for many programming languages is an interactive interpreter.

1. Print a prompt
2. Read text input from the user
3. Parse the text input into an expression
4. Evaluate the expression
5. If any errors occur, report those errors, otherwise
6. Print the value of the expression and repeat

**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.

```
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")
```