Announcements
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

```python
from dis import dis
dis(square)

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

**Python 3 Byte Code**

<table>
<thead>
<tr>
<th>Python 3 Byte Code</th>
<th>Python 3 Byte Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD_FAST 0 (x)</td>
<td>LOAD_FAST 0 (x)</td>
</tr>
<tr>
<td>LOAD_FAST 0 (x)</td>
<td>LOAD_FAST 0 (x)</td>
</tr>
<tr>
<td>BINARY_MULTIPLY</td>
<td>BINARY_MULTIPLY</td>
</tr>
<tr>
<td>RETURN_VALUE</td>
<td>RETURN_VALUE</td>
</tr>
</tbody>
</table>
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

**Type of application:** Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

**Problem domain:** The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.

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:

\[(\text{element}_0 \ \text{element}_1 \ \ldots \ \text{element}_n)\]

A Scheme list

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)

http://composingprograms.com/examples/scalc/scheme_reader.py.html
Parsing

A Parser takes text and returns an expression

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines
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

```
'(+, '+', 1, '(, '-', 23, ')', '(, '*', 4, 5.6, ')', ')
```

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them

(Demo)
The Pair Class

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

```python
class Pair:
    """A Pair has two instance attributes: first and second.
    For a Pair to be a well-formed list, 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

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
>>> len(Pair(1, Pair(2, 3)))
Traceback (most recent call last):
  ...
TypeError: length attempted on improper list
```

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

(Demo)
**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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Tree</th>
<th>Representation as Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(* 3 (+ 4 5) (* 6 7 8))</td>
<td>[Diagram of expression tree]</td>
<td><img src="image" alt="Pairs representation" /></td>
</tr>
</tbody>
</table>
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.

Expression

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

Expression Tree

[Diagram of an expression tree for the given expression]
Evaluation
The Eval Function

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).

Implementation

```python
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
```

Language Semantics

- **A number evaluates**...
  - to itself
- **A call expression evaluates**...
  - to its argument values
  - combined by an operator

Recursive call returns a number for each operand.

A Scheme list of numbers.
Applying 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: +, -, *, /

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def calc_apply(operator, args):</code></td>
<td></td>
</tr>
<tr>
<td>if operator == '+':</td>
<td><code>+:</code></td>
</tr>
<tr>
<td>return reduce(add, args, 0)</td>
<td>\textit{Sum of the arguments}</td>
</tr>
<tr>
<td>elif operator == '-':</td>
<td><code>-:</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>elif operator == '*':</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>elif operator == '/':</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>raise TypeError</td>
<td></td>
</tr>
<tr>
<td>(Demo)</td>
<td></td>
</tr>
</tbody>
</table>
Interactive Interpreters
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

(Demo)
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")

(Demo)
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.

(Demo)