Programming Languages

Computers have software written in many different languages.

Machine languages: statements can be interpreted by hardware
- All data are represented as sequences of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details
- Primitive data types beyond just bits
- Statements/expressions can be non-primitive (e.g., calls)
- Evaluation process is defined in software, not hardware

High-level languages are built on top of low-level languages

Metalinguistic Abstraction

**Metalinguistic abstraction**: Establishing new technical languages (such as programming languages)

\[ f(x) = x^2 - 2x + 1 \]

\[ \lambda f.(\lambda x.f(x))(\lambda x.f(x)) \]

In computer science, languages can be implemented:
- 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.
- The semantics and syntax of a language must be specified precisely in order to build an interpreter.

Syntax and Semantics of Calculator

Expression types:
- A **call expression** is a Scheme list
- A **primitive expression** is an operator symbol or number

Operators:
- The + operator **returns** the sum of its arguments
- The - operator **returns** either
  - the additive inverse of a single argument, or
  - the sum of subsequent arguments subtracted from the first
- The * operator **returns** the product of its arguments
- The / operator **returns** the real-valued quotient of a dividend and divisor (i.e., a numerator and denominator)

The Scheme-Syntax Calculator Language

A subset of Scheme that includes:
- Number primitives
- Built-in arithmetic operators: +, -, *, /
- Call expressions

\[ \begin{align*}
> & \ (+ \ (* \ 3 \ 5) \ (- \ 10 \ 6)) \\
& \quad 19 \\
> & \ (+ \ (* \ 3 \\
& \quad (+ \ (* \ 2 \ 4) \\
& \quad (+ \ (* \ 3 \ 5))) \\
& \quad (+ \ (- \ 10 \ 7) \\
& \quad 6)) \\
> & \quad 57
\end{align*} \]

Expression Trees

A basic interpreter has two parts: a parser and an evaluator

<table>
<thead>
<tr>
<th>lines</th>
<th>Parser expression</th>
<th>Evaluator value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(+ 2 2)'</td>
<td>Pair('+', Pair(2, Pair(2, nil)))</td>
<td>4</td>
</tr>
<tr>
<td>'(* (+ 1'</td>
<td>Pair('+', Pair(Pair('+', ...)))</td>
<td>4</td>
</tr>
<tr>
<td>'10)'</td>
<td>printed as ( (+ 1 \ (- 23) (+ 4 \ 5.6)) \ 10)</td>
<td>18)</td>
</tr>
<tr>
<td>Lines forming a Scheme expression</td>
<td>A number or a Pair with an operator as its first element</td>
<td>A number</td>
</tr>
</tbody>
</table>
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

Demo (http://inst.eecs.berkeley.edu/~cs61a/fa12/projects/scalc/scheme_reader.py.html)

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule.
• Primitive expressions 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

Demo

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(operator, args):
    '''Apply the named operator to a list of args.'''
    if operator == '+':
        return ...
    if operator == '-':
        ...
    ...
```

Dispatch on operator name
Demo

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which
• Reads an expression from the user,
• Parses the input to build an expression tree,
• Evaluates the expression tree,
• Prints the resulting value of the expression.

Demo

Handling Errors

The REPL handles errors by printing informative messages for the user, rather than crashing.

```
def calc_apply(operator, args):
    '''Apply the named operator to a list of args.'''
    ...
    if operator == '+':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator == '/':
        if len(args) != 2:
            raise TypeError(operator + ' requires exactly 2 arguments')
        ...
```

A well-designed REPL should not crash on any input!

Demo

Raising Application Errors

The sub and div operators have restrictions on argument number.

Raising exceptions in apply can identify such issues:

```
def calc_apply(operator, args):
    '''Apply the named operator to a list of args.'''
    ...
    if operator == '+':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        ...
    ...
    if operator == '/':
        if len(args) != 2:
            raise TypeError(operator + ' requires exactly 2 arguments')
        ...
```