61A Lecture 30

Wednesday, November 9

Functional Programming

All functions are pure functions

No assignment and no mutable data types

Name-value bindings are permanent

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated
- Sub-expressions can safely be evaluated in parallel or lazily
- Referential transparency: The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression.

The subset of Logo we have considered so far is functional (except for print/show)

The Logo Assignment Procedure

Logo binds variable names to values, as in Python

An environment stores name bindings in a sequence of frames

Each frame can have at most one value bound to a given name

The make procedure adds or changes variable bindings

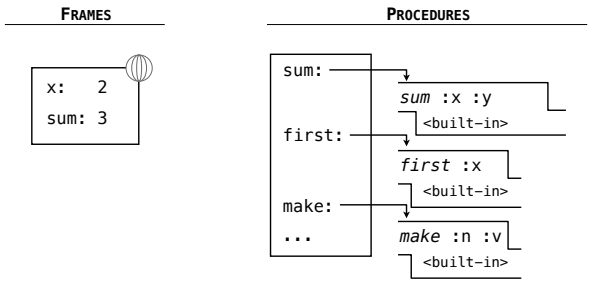
```
? make "x 2
```

Values bound to names are looked up using variable expressions

```
? print :x
2
```

Demo

Namespaces for Variables and Procedures




```
? make "sum 3
```

Demo

Assignment Rules

Logo assignment has different rules from Python assignment:

```
? make <name> <value>
```

- If the name is already bound, *make* re-binds that name in the first frame in which the name is bound.

Like non-local Python assignment

- If the name is not bound, *make* binds the name in the global frame.

Like global Python assignment

Implementing the Make Procedure

The implementation of make requires access to the environment

```
def logo_make(symbol, val, env):
    env.set_variable_value(symbol, val)


class Environment(object):
    def __init__(self, get_continuation_line=None):
        self.get_continuation_line = get_continuation_line
        self.procedures = load_primitives()
        self._frames = [dict()] # The first frame is global

    def set_variable_value(self, symbol, val):
        "*** YOUR CODE HERE ***"
```

Evaluating Definitions

A procedure definition (to statement) creates a new procedure and binds its name in the table of known procedures

```
? to factorial :n
> output ifelse :n = 1 [1] [:n * factorial :n - 1]
> end
```

```
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False,
                 needs_env=False, formal_params=None):
        ...
```

Formal parameters: a list of variable names (without colons)

Body: a list of Logo sentences

Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, extending the current environment

Evaluate each line of the body of the procedure in the environment that starts with this new frame

If any top-level expression evaluates to a non-None value, raise an error

Output values require special handling:

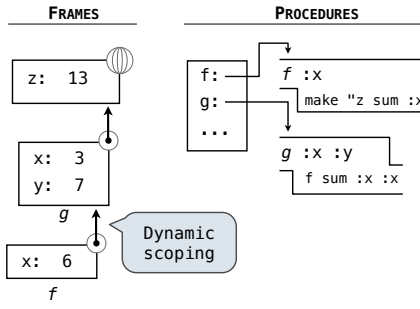
- Output returns a pair: ('OUTPUT', <value>)
- Stop returns a pair: ('OUTPUT', None)

logo_apply returns the <value> that is output by the body

Demo

Dynamic Scope and Environments

A new frame for an applied procedure extends the current frame

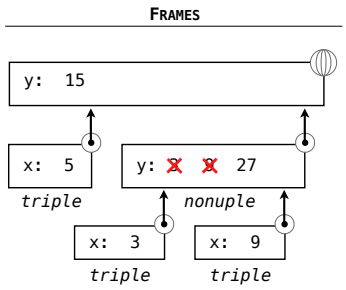


Demo

```
? to f :x
> make "z sum :x :y
> end
? to g :x :y
> f sum :x :x
> end
? g 3 7
? print :z
13
```


Dynamic Scope and Environments

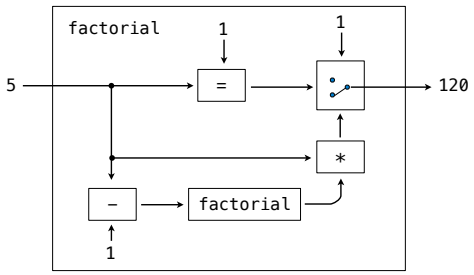
This example was presented in class on the chalkboard




```
? to triple :x
> make "y product :x 3
> output :y
> end
? to nonuple :y
> output triple triple :y
> end
? print triple 5
15
? print nonuple 3
27
? print :y
15
```

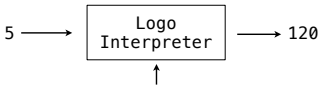

An Analogy: Programs Define Machines

Programs specify the logic of a computational device



Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine




```
to factorial :n
output ifelse :n = 1 [1] [:n * factorial :n - 1]
end
```

Our Logo interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself

Internally, it is just a set of manipulation rules

Interpretation in Python

eval: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

exec: Executes a statement in the current environment. Doing so may affect the environment.

```
eval('2 + 2')
```

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

os.system('python <file>'): Directs the operating system to invoke a new instance of the Python interpreter.

Demo