61A LECTURE 21 – INTERPRETERS

Steven Tang and Eric Tzeng

July 30, 2013

Announcements

- Project 4 out today
  - Start soon – most time consuming project!
- Homework 11 due date pushed to Friday
  - Relatively short assignment. Great introduction to the project!
- Homework 12 out later today.

The Scheme-Syntax Calculator Language

A subset of Scheme that includes:

- Number primitives
- Built-in arithmetic operators: **+, -, *, /**
- Call expressions

```
> (+ (* 3 5) (- 10 6))
19
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

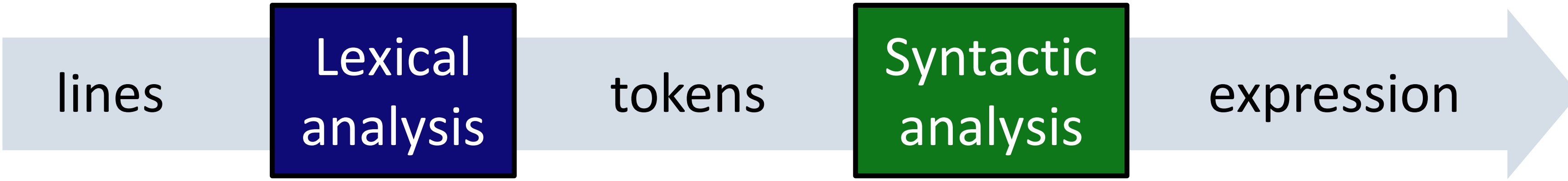
Input on multiple lines did not work in minicalc.

Allowing for input on multiple lines

- `read_exp` raises a `SyntaxError` if the input is not completely well formed
- Another version of Calculator: use `scalc` instead of `minicalc`.
- `scalc` makes use of the `yield` statement, which we will talk about next week.
- Simply know that `scalc` is essentially `minicalc`, but allows for input on multiple lines.
- `scalc` contains functions analogous to what’s used in project 4

Semi-Review: Parsing in scalc

A parser takes a sequence of lines and returns an expression.




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

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

```
Pair('+', Pair(1, ...))
```

printed as

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

- 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. `scheme_read` and `exp_read` are analogous.

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

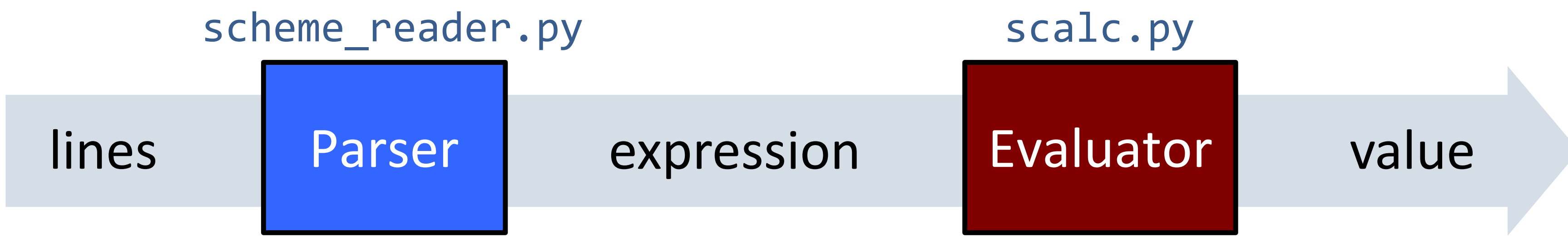
Base case: symbols and numbers

Recursive call: **`scheme_read`** sub-expressions and combines them as pairs

http://inst.eecs.berkeley.edu/~cs61a/book/examples/scalc/scheme_reader.py.html

Expression Trees

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



`'(+ 2 2)'`	`Pair('+', Pair(2, Pair(2, nil)))`	`4`

`'(* (+ 1'`
`'      (- 23)'`
`'      (* 4 5.6))’`
`'   10)'`
	`Pair('*', Pair(Pair(+, ...))`
	printed as
	`(* (+ 1 (- 23) (* 4 5.6)) 10)`
		`4`

Lines forming a Scheme expression	A number or a **`Pair`** with an operator as its first element	A number

Evaluation in Calculator

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

The Structure of an Evaluator

Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

- Eval(operands) of call expressions
- Apply(operator, arguments)
- Eval(sub-expressions) of special forms

Requires an environment for name lookup

Apply

Base cases:

- Built-in primitive procedures

Recursive calls:

- Eval(body) of user-defined proc's

Creates new environments when applying user-defined procedures

Break

Scheme Evaluation

The **scheme_eval** function dispatches on expression form:

- Symbols are bound to values in the current environment
- Self-evaluating primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

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

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

(define <name> <expression>)

(<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element

Anything not a known special form is a call expression

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (list 1 2))
```

Logical Special Forms

Logical forms may only evaluate some sub-expressions.

- **If** expression: (**if** <predicate> <consequent> <alternative>)
- **And** and **or**: (**and** <e_1> ... <e_n>), (**or** <e_1> ... <e_n>)
- **Cond** expr'n: (**cond** (<p_1> <e_1>) ... (<p_n> <e_n>) (**else** <e>))

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: **<consequent>** or **<alternative>**

do_if_form

- Evaluate that sub-expression in place of the whole expression.

scheme_eval

Quotation

The **quote** special form evaluates to the quoted expression

(quote <expression>)

Evaluates to the **<expression>** itself, not its value!

'<expression> is shorthand for **(quote <expression>)**

(quote (1 2))

'(1 2)

The **scheme_read** parser converts shorthand to a combination

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

(lambda (x) (* x x))
```

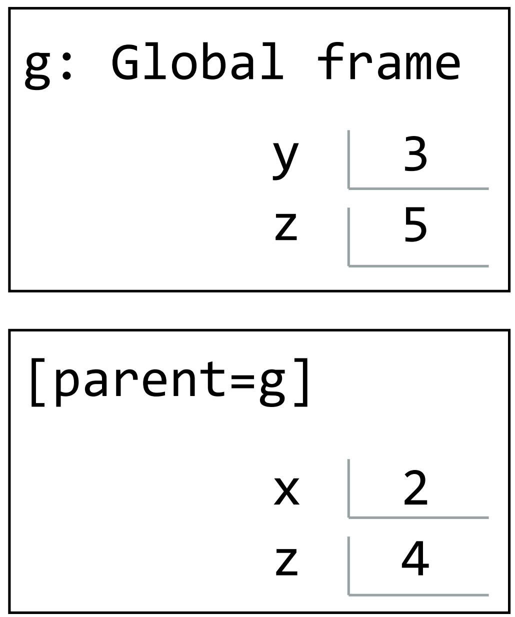
```
class LambdaProcedure(object):

    def __init__(self, formals, body, env):

        self.formals = formals      A scheme list of symbols

        self.body = body            A scheme expression

        self.env = env              A Frame instance
```

Frames and Environments

A frame represents an environment by having a parent frame

`Frames` are Python instances with methods **`lookup`** and **`define`**

In Project 4, **`Frames`** do not hold return values



Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

```
(define <name> <expression>)
```

Evaluate the **<expression>**

Bind **<name>** to the result (**define** method of the current **Frame**)

```
(define x 2)
```

Procedure definition is a combination of define and lambda

```
(define (<name> <formal parameters>) <body>)

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

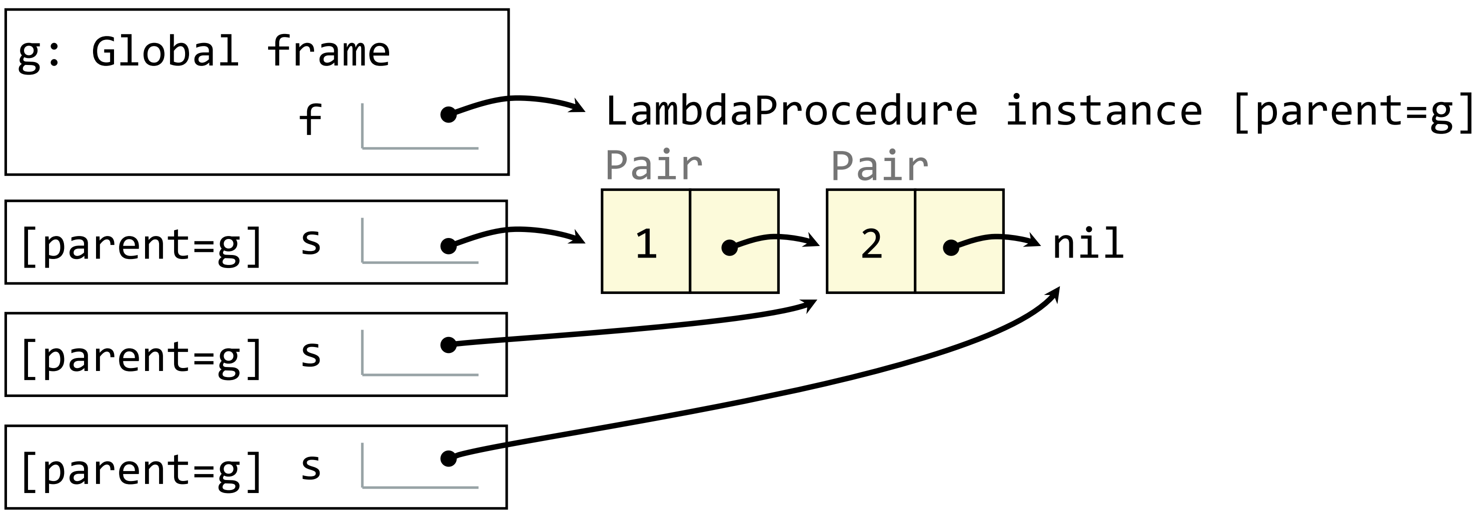
Applying User-Defined Procedures

Create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure

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

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
```

```
(f (list 1 2))
```



Break: Eval/Apply in Lisp 1.5

```
apply[fn;x;a] =
      [atom[fn] → [eq[fn;CAR] → caar[x];
                   eq[fn;CDR] → cdar[x];
                   eq[fn;CONS] → cons[car[x];cadr[x]];
                   eq[fn;ATOM] → atom[car[x]];
                   eq[fn;EQ] → eq[car[x];cadr[x]];
                   T → apply[eval[fn;a];x;a]];
      eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
      eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                    caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
      atom[car[e]] →
              [eq[car[e],QUOTE] → cadr[e];
              eq[car[e];COND] → evcon[cdr[e];a];
              T → apply[car[e];evlis[cdr[e];a];a]];
      T → apply[car[e];evlis[cdr[e];a];a]]
```

Dynamic Scope

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*)

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

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

Special form to create dynamically scoped procedures

```
                mu
(define f (~~lambda~~ (x) (+ x y)))

(define g (lambda (x y) (f (+ x x))))

(g 3 7)
```

Lexical scope: The parent for **f**'s frame is the global frame

Error: unknown identifier: y

Dynamic scope: The parent for **f**'s frame is **g**'s frame

13

Practice

```
y = 5
def foo(x):
    return x + y
def garply(y):
    return foo(2)
```

What does `garply(10)` return? What about if Python used dynamic scoping?

Functional Programming

All functions are pure functions

No re-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 sub-expression with the value of that sub-expression