61A LECTURE 19 – CALCULATOR

Steven Tang and Eric Tzeng

July 25, 2013

Turtle graphics

Picture by Jonathan Zander

- STk has built in support for basic 2D graphics!
- Turtle sits on the canvas
- As the turtle "walks" around the canvas, it leaves a trail
- Images are drawn by issuing commands to the turtle

```
(define (triangle)
  (forward 100)
  (right 120)
  (forward 100)
  (right 120)
  (forward 100)
  (right 120))
```

Move forward 100 steps

Turn right 120 degrees

- Did we need the last call to `right`? Why?

The Begin Special Form

Begin expressions allow sequencing

```
(begin <exp1> <exp2> ... <expn>)
```

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120))))

(define (sier d k)
  (tri (lambda () (if (= k 1) (fd d) (leg d k)))))

(define (leg d k)
  (sier (/ d 2) (- k 1)) (penup) (fd d) (pendown))
```

You are now Scheme masters!

- That wraps up our discussion of Scheme
- From here on out, the focus is is going to be on interpreters
- In other words, we’re writing programs that understand programs!

Programming Languages

Computers have software written in many different languages

Machine languages: statements can be interpreted by hardware

- All data are represented as a sequence of bits
- All statements are primitive instructions

High-level languages: hide concerns about those details

- Primitive data types beyond just bits
- Statements/expressions, data 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

Machine Language	C	Python

Metalinguistic Abstraction

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

$$f(x) = x^2 - 2x + 1$$

$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

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

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
```

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)

Today…

- We’re going write an interpreter for this language
- And we’re going to do it from (almost) scratch!
    - We’re going to reuse some Rlist functions that you’ve seen before
- You know how to do everything in this lecture already!
- We’re just putting it together

Reading Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

A recursive 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

Parsers must validate that expressions are well-formed

Parsing

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

lines → **Lexical analysis** → tokens → **Syntactic analysis** → 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

- Tree-recursive process
- Balances parentheses
- Returns tree structure

Lexical analysis

- It’s hard to directly determine the program structure from a string

```
'(+ 1 (* 3 4) 2)'
```

Tokenization!

```
['(', '+', '1', '(', '*', '3', '4', ')', '2', ')']
```

- Split the string into a sequence of “tokens”
- From the token sequence, it’s a lot easier to determine the program structure

Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to **`read_exp`** consumes the input tokens for exactly one expression.

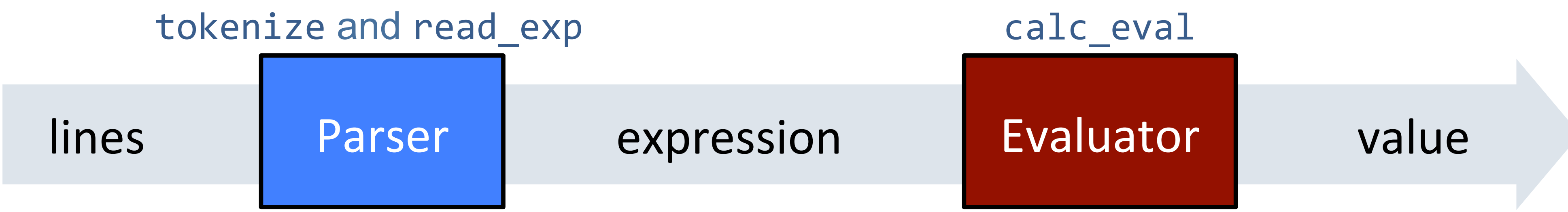
```
'(', '+', 1, '(', '-', 23, ')', '(', '*', 4, 5.6, ')', ')'
```

Base case: symbols and numbers

Recursive call: call **`read_tail`**, which uses **`read_exp`** for sub-expressions and combines them as pairs

Expression Trees

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



`'(+ 2 2)'`	`Pair('+', Pair(2, Pair(2, nil)))`	`4`
`'(* (+ 1 (- 23)) 2)'`	`Pair('*', Pair(Pair(+, ...))`	`-44`
String forming a Scheme expression	A number or a **`Pair`** with an operator as its first element	A number

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

Applying Operators

Calculator has a fixed set of operators that we can enumerate

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

Dispatch on operator name

Raising Application Errors

The – and / operators have restrictions on argument number

Raising exceptions in *apply* can identify such issues

```
def calc_apply(op, args):
    """Apply an operator to a list of args."""
    if op == '-':
        if len(args) == 0:
            raise TypeError('Not enough arguments')
        ...
    if op == '/':
        if len(args) == 2:
            raise TypeError('Not enough arguments')
         ...
```

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

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

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