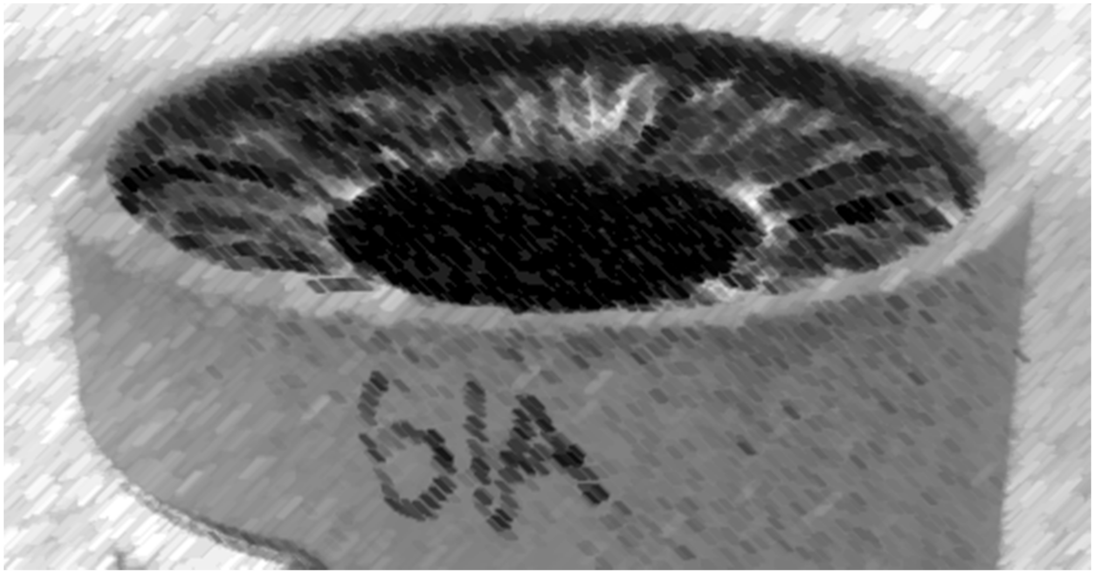


CS61A Lecture 33

Amir Kamil

UC Berkeley

April 8, 2013

Announcements

- Hog revisions due tonight
- HW10 due Wednesday
- Last chance to fill out survey on Piazza
  - We need to schedule alternate final exam times for those who have a conflict, so if you do, let us know on the survey when you are available

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)

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))
```

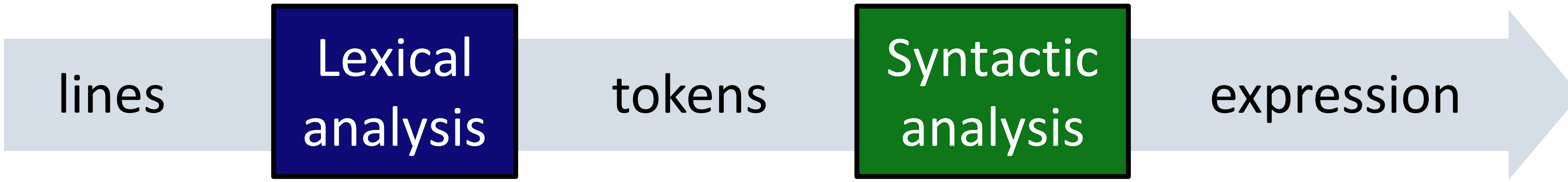
The task of *parsing* a language involves coercing a string representation of an expression to the expression itself

Parsers must validate that expressions are well-formed

(http://inst.eecs.berkeley.edu/~cs61a/sp13/projects/scalc/scheme_reader.py.html)

Parsing

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.

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

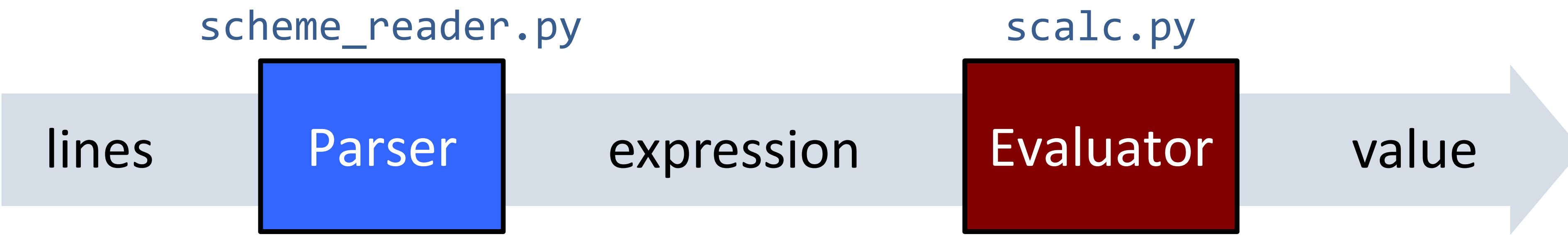
Base case: symbols and numbers

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

(http://inst.eecs.berkeley.edu/~cs61a/sp13/projects/scalc/scheme_reader.py.html)

Expression Trees

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



lines	expression	value
`'(+ 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

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

Dispatch on operator name

(http://inst.eecs.berkeley.edu/~cs61a/sp13/projects/scalc/scalc.py.html)

Raising Application Errors

The **-** and **/** 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')
        ...
```

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!