CS61A Lecture 27

Friday, November 1, 2019

Announcements

- Homework 7 is released and due Thursday, November 7.

Calculator

We've learnt Python: that took 9 weeks. We learnt Scheme: that took a day! And now, we are shifting focus to building an interpreter.

The Scheme project will be released on Monday. We will spend the next two lectures talking about the structure of an interpreter. Today, we will discuss with a simpler language than Scheme.

Programming Languages

A computer typically executes programs written in many different programming languages.

So far we've learnt Python and Scheme, which are pretty similar. They are high-level languages, which provide a means of abstraction, such as naming, functions and objects. These statements are interpreted by other programs, or alternatively, they can be turned into something the machine can run. The machine runs a language called a **machine language**.

Machine languages include statements that are inteprreted by the hardware itself. It doesn't have a lot of features that we are used to, because they are a fixed set of instructions that invoke operations implemented by the circuitry of the CPU. Operations refer to specific hardware memory addresses. There are no features like naming and functions, and you need to write machine language for each individual system because they can't abstract away system details like the manufacturer and operating system.

When you run Python code, it converts code into something in-between. It converts your code into something called Python 3 Byte Code. You can see how byte code is run using:

```
from dis import dis
dis(<function>)
```

Metalinguistic Abstraction

This is an invitation to write new programming languages! A powerful form of abstraction is to define a new language is tailored to particular applications or problem domains.

Type of applications: Erlang was designed for concurrent programs. It has built-in elements for expressing concurrent communication. It is used, for example, to implement chat servers with many simultaneous connections.

Problem domain: The MediaWiki mark-up language was designed for generating static web pages. It has built-in elements for text formatting and cross-page linking. It is used, for example, to create Wikipedia pages.

A programming language has:

- Syntax: the legal statements and expressions in the language
- Semantics: the execution/evaluation rule for those statements and expressions

To create a new programming language, you either need a:

- Specification: A document describe the precise syntax and semantics of the language, or a
- Canonical Implementation: An interpreter or compiler for the language.

Parsing

What happens when you try to interpret a programming language?

First, you have to check the code is well-formed, legal according to its rules. Then, we will check for the code fits with other parts of the code so far. This is called **parsing**, and we will discuss parsing with Scheme.

A Scheme list is written as elements in parentheses:

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

Each `<element>` can be a combination or primitive.

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

As part of the Scheme project, the staff will release a program called the Scheme Reader, which will show you how Scheme expressions are interpreted in Scheme and Python. For example:

```
> 1
Scheme: 1
Python: 1
> +
Scheme: +
Python: '+' #Python uses strings to keep track of operators
> (+ 1 2)
Scheme: (+ 1 2)
Python: (Pair('+',Pair(1,Pair(2,nil))))
```

Parser

A parser takes text and returns an expression, like the one above. How does it work?

Well, the parser first takes text like:

```
(+ 1 24)
```

And performs **lexical analysis** on it. Lexical analysis is an iterative process that checks for malformed tokens, determines types of tokens, and processes one line at a time. It knows how to separate pieces of code, like operators and brackets. For the code above:

```
'(', '+', 1, 24, ')'
```

From these tokens, we do **syntactic analysis**. Syntactic analysis is a tree-recursive process (because it checks that parentheses are balanced, and all other expressions within are balanced). It returns a tree structure, and processes multiple lines at a time.

Syntactic Analysis

Syntactic analysis identifies hierarchial of an expression, which may be nested.

Each call to `scheme_read` (the analysis function) consumes the input tokens for exactly one expression.

Take the following example:

```
(+ 1 (* 3 3))
```

The base case for this function is when symbols and numbers are encountered, while the recursive case is `scheme_read` sub-expressions that need to be combined.

So, the first call to `scheme_read` is on the entire bracketed expression and consumes the entire set of tokens. When it encounters symbols or numbers, it continues on. Otherwise, when it meets the inner bracket, it calls `scheme_read` on `(* 3 3)`.

Even though this is a tree-recursive process, it does not take exponential time. It takes roughly linear time, because every call to scheme_read uses up tokens.

Here's how `scheme_read` is written:

```
def scheme_read(src):
    """Read the next expression from src, a Buffer of tokens.

    >>> lines = ['(+ 1 ', '(+ 23 4)) (']
    >>> src = Buffer(tokenize_lines(lines))
    >>> print(scheme_read(src))
    (+ 1 (+ 23 4))
    """
    if src.current() is None:
        raise EOFError
    val = src.pop()
    if val == 'nil':
        return nil
    elif val not in DELIMITERS:  # ( ) ' .
        return val
    elif val == "(":
        return read_tail(src)
    else:
        raise SyntaxError("unexpected token: {0}".format(val))

def read_tail(src):
    """Return the remainder of a list in src, starting before an element or ).

    >>> read_tail(Buffer(tokenize_lines([')'])))
```

```
    nil
    >>> read_tail(Buffer(tokenize_lines(['2 3)'])))
    Pair(2, Pair(3, nil))
    >>> read_tail(Buffer(tokenize_lines(['2 (3 4))'])))
    Pair(2, Pair(Pair(3, Pair(4, nil)), nil))
    """
    if src.current() is None:
        raise SyntaxError("unexpected end of file")
    if src.current() == ")":
        src.pop()
        return nil
    first = scheme_read(src)
    rest = read_tail(src)
    return Pair(first, rest)
```

And here is the `Pair` class is written:

```
class Pair(object):
    """A pair has two instance attributes: first and second.  For a Pair to be
    a well-formed list, second is either a well-formed list or nil.  Some
    methods only apply to well-formed lists.

    >>> s = Pair(1, Pair(2, nil))
    >>> s
    Pair(1, Pair(2, nil))
    >>> print(s)
    (1 2)
    >>> len(s)
    2
    >>> s[1]
    2
    >>> print(s.map(lambda x: x+4))
    (5 6)
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return "Pair({0}, {1})".format(repr(self.first), repr(self.second))

    def __str__(self):
        s = "(" + str(self.first)
        second = self.second
        while isinstance(second, Pair):
            s += " " + str(second.first)
            second = second.second
        if second is not nil:
            s += " . " + str(second)
        return s + ")"

    def __len__(self):
        n, second = 1, self.second
        while isinstance(second, Pair):
```

```
            n += 1
            second = second.second
        if second is not nil:
            raise TypeError("length attempted on improper list")
        return n

    def __getitem__(self, k):
        if k < 0:
            raise IndexError("negative index into list")
        y = self
        for _ in range(k):
            if y.second is nil:
                raise IndexError("list index out of bounds")
            elif not isinstance(y.second, Pair):
                raise TypeError("ill-formed list")
            y = y.second
        return y.first

    def map(self, fn):
        """Return a Scheme list after mapping Python function FN to SELF."""
        mapped = fn(self.first)
        if self.second is nil or isinstance(self.second, Pair):
            return Pair(mapped, self.second.map(fn))
        else:
            raise TypeError("ill-formed list")

class nil(object):
    """The empty list"""

    def __repr__(self):
        return "nil"

    def __str__(self):
        return "()"

    def __len__(self):
        return 0

    def __getitem__(self, k):
        if k < 0:
            raise IndexError("negative index into list")
        raise IndexError("list index out of bounds")

    def map(self, fn):
        return self

nil = nil()
```

Calculator

With this, we can now demonstrate a basic interpreter for a special language called calculator!

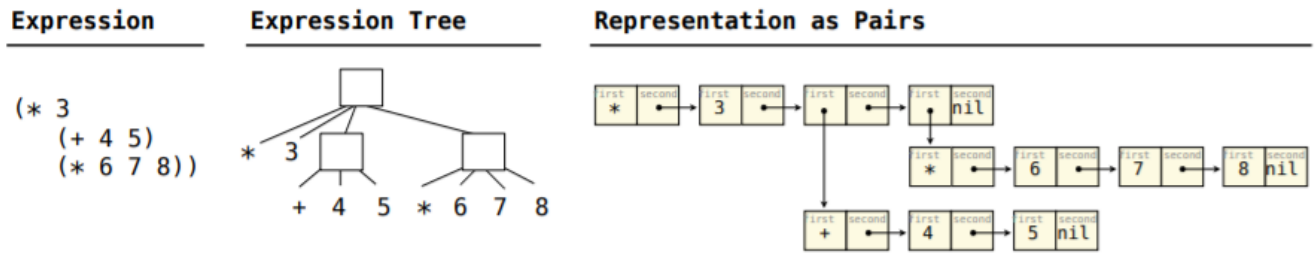
```
> (+ 1 1)
```

```
2
> (- 3)
-3
> (/ 4)
0.25
```

The Calculator language has primitive expressions and call expressions. (That's it!)

- A primitive expression is a number: 2, -4, 5.6
- A call expression is a combination that begins with an operator `(+, -, *, /)` followed by 0 or more expressions: `(+ 1 2 3)` , `(/ 3 (+ 4 5))`

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.



Calculator Semantics

The value of a calculator expression is defined recursively:

A primitive is a number that evaluates by it.

A call expression evaluates to its argument combined by an operator.

- `+` : Sum of the arguments
- `*` : Product of the arguments
- `-` : If one argument, negate it. If more than one, subtract the rest from the first.
- `/` : If one argument, invert it. If more than one, divide the rest from the first.

The Eval Function

The `eval` function computes the value of an expression, which is always a number. It is a generic function that dispatches on the type of the expression (primitive or call).

```
def calc_eval(exp):
     if type(exp) in (int, float):
        return exp
     elif isinstance(exp, Pair):
         arguments = exp.second.map(calc_eval)
         return calc_apply(exp.first, arguments)
     else:
        raise TypeError
```

The `eval` function is short, because it offloads most of its work to the `calc_apply` function. It does the built-in error handling.

The Apply Function

The `apply` function applies some operation to a Scheme list of argument values.

In calculator, operators are named by built-in operators.

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args.

    if not isinstance(operator, str):
        raise TypeError(str(operator) + ' is not a symbol')
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        elif len(args) == 1:
            return -args.first
        else:
            return reduce(sub, args.second, args.first)
    elif operator == '*':
        return reduce(mul, args, 1)
    elif operator == '/':
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        elif len(args) == 1:
            return 1/args.first
        else:
            return reduce(truediv, args.second, args.first)
    else:
        raise TypeError(operator + ' is an unknown operator')
```

User Interface

The user interface for many programming languages is an interactive interpreter:

1. Print a prompt
2. Read text input from the user
3. Parse the text into an expression
4. Evaluate the expression
5. If errors occur, report them, otherwise
6. Print the value of the expression and repeat

Raising Exceptions

Exceptions are sort of all over the place, because they are checked where they make the most sense, not in some place simply because we think it will look better.

- Lexical analysis: The token 2.3.4 raises `ValueError` ("invalid numeral")
- Syntactic analysis: An extra) raises `SyntaxError` ("unexpected token")
- Eval: An empty combination raises `TypeError` ("() is not a number or call expression")
- Apply: No arguments to `--` raises TypeError ("-- requires at least one argument")

Handling Exceptions

An interactive interpreter prints information about each error.

A well-designed interactive interpreter should not halt completely on an error, so the user has an opportunity to try again in the current environment.

We do this with the following code:

```
@main
def read_eval_print_loop():
    """Run a read-eval-print loop for Calculator."""
    while True:
        try:
            src = buffer_input()
            while src.more_on_line:
                expression = scheme_read(src)
                print(calc_eval(expression))
        except (SyntaxError, TypeError, ValueError, ZeroDivisionError) as err:
            print(type(err).__name__ + ':', err)
        except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
            print('Calculation completed.')
            return
```

This code uses `while True` to run in an infinite loop, unless the user specifically instructs the interpreter to stop with `KeyboardInterrupt`.