CS 314 Midterm Review

March 12, 2019

Regular expressions

We define a regular expression with characters and a few operators:

- concatenation: ab means a followed by b
- alternation: $a|b$ means either a or b
- Kleene star: a^* means 0 or more copies of a
- and parentheses for grouping

(and ϵ denotes the empty string)

Context-free grammar

We can define a *grammar* using Backus-Naur form (BNF):

$$
\begin{aligned}
\langle \textit{expr} \rangle \quad &::= \langle \textit{expr} \rangle + \langle \textit{expr} \rangle \\
&\quad\mid \langle \textit{expr} \rangle \text{ - } \langle \textit{expr} \rangle \\
&\quad\mid \langle \textit{variable} \rangle \\
&\quad\mid \langle \textit{number} \rangle
\end{aligned}
$$

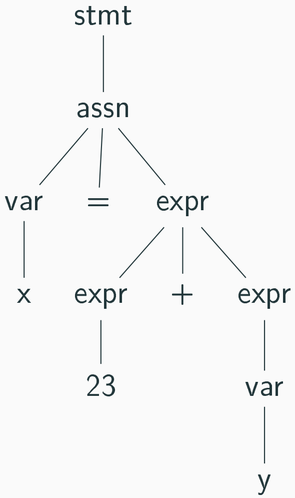
$$\langle \textit{variable} \rangle ::= \text{a} \mid \text{b} \mid \text{c} \mid \ldots \mid \text{z}$$

$$\langle \textit{number} \rangle ::= 1 \mid 2 \mid 3 \mid \ldots \mid 9$$

Parsing

How does a program get read? Going from tokens to a parse tree (assuming a reasonable grammar):

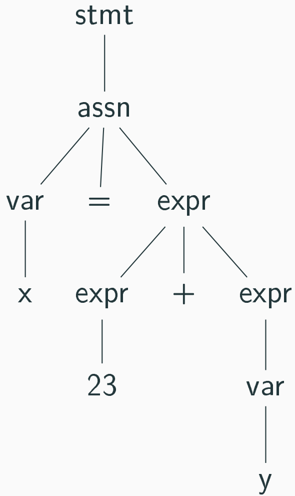
```
x = 23 + y;
```

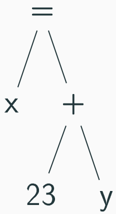


Abstract syntax trees (ASTs)

Parse trees are *concrete*.

But usually we don't care about the full derivation:





Duck typing

```python
class Duck:
    def fly(self):
        print('Duck flying')

class Airplane:
    def fly(self):
        print('Airplane flying')

def lift_off(entity):
    entity.fly()

duck = Duck()
airplane = Airplane()

lift_off(duck)      # prints `Duck flying`
lift_off(airplane) # prints `Airplane flying`
```

Anonymous functions

```python
def dbl(x):
    return x * 2

dbl(10)
```

```python
dbl = lambda x: x * 2

dbl(10)
```

Program state

```
x = 42
# {x: (int, 42)}

y = 'hi'
# {x: (int, 42), y: (str, 'hi')}

z = 3.5
# {x: (int, 42), y: (str, 'hi'), z: (float, 3.5)}
```

Lambda calculus

- variables
  - $x, y, z, ...$
- abstraction
  - $\lambda x.x$
- application
  - $(\lambda x.x)y$

Evaluation

What does $(\lambda x.(\lambda y.x))(\lambda z.w)$ mean?

- $\lambda x.(\lambda y.x)$ is a function with parameter x and body $\lambda y.x$.
- Replace every occurance of the parameter in the body with the actual argument $(\lambda z.w)$.
- Replace every x in $\lambda y.x$ with $\lambda z.w$.
- $\lambda y.(\lambda z.w)$

Free and bound variables

In the expression $\lambda x.xy$, we say the x in the body is *bound* (by the enclosing λ), but y is free.

Bound variables

BV denotes the bound variables of a lambda term:

- $BV\ x = \{\}$
- $BV\ (\lambda x.M) = (BV\ M) \cup \{x\}$
- $BV\ (M\ N) = (BV\ M) \cup (BV\ N)$

Free variables

FV denotes the free variables of a lambda term:

- $FV\ x = \{x\}$
- $FV\ (\lambda x.M) = (FV\ M) - \{x\}$
- $FV\ (M\ N) = (FV\ M) \cup (FV\ N)$

If $(FV\ M) = \{\}$, M is called *closed*. M is also called a *combinator*.

Variable capture

This is called variable capture: a variable that was free becomes bound.

$$(\lambda x.\lambda y.xy)yz \Rightarrow (\lambda y.yy)z$$

- The x in $\lambda y.xy$ is free (although bound in $\lambda x.\lambda y.xy$)
- But both ys in $\lambda y.yy$ are bound.

α-conversion

We can rename xs in $\lambda x.M$ with y, as long as y is not already a free variable in the body:

$\lambda x.M \equiv \lambda y.M[x := y]$, where $y \notin FV\ M$

- $\lambda x.xx = \lambda y.yy$
- $\lambda x.xy \neq \lambda y.yy$

η-reduction

One last rule (η-reduction):

Given an expression of the form $\lambda x.fx$, we can replace this with f.

```
double mySin(double x)
{
    return sin(x);
}
```

Normal form

Does every lambda term have a normal form? Consider this expression:

$$(\lambda x.xx)(\lambda x.xx)$$

- Replace every x in xx with the argument $(\lambda x.xx)$
- We get $(\lambda x.xx)(\lambda x.xx)$
- We can do function application!
- ...

Evaluation order

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$$

- Applicative order: evaluate the argument to a normal form first
- Normal order: evaluate the top-most element first

Church booleans

We can do more than just symbol manipulation with lambda calculus.

Let's define boolean values in terms of lambda expressions:

- $\lambda x.\lambda y.x \equiv \text{true}$
- $\lambda x.\lambda y.y \equiv \text{false}$

Boolean functions

not is a function that negates its argument:

x	not x
$\lambda xy.x$	false
$\lambda xy.y$	true

not: $\lambda p.(p$ false true$)$

Church numerals

- zero: $\lambda f.\lambda x.x$
- one: $\lambda f.\lambda x.fx$
- two: $\lambda f.\lambda x.f(fx)$

Church numerals

The successor function, succ: $\lambda n.\lambda f.\lambda x.f(nfx)$

- succ zero
- $(\lambda nfx.f(nfx))(\lambda fx.x)$
- $(\lambda fx.f((\lambda fx.x)fx))$
- $(\lambda fx.fx)$
- one

Functional programming

Fundamental concept: application of (mathematical) functions to values

- Referential transparency: The value of a function application is independent of the context in which it occurs
  - value of $f(a, b, c)$ depends only on the values of f, a, b and c
  - It does not depend on the global state of computation
  - $\Rightarrow$ all vars in function must be local (or parameters)

Pure Functional Languages

The concept of assignment is not part of functional programming.

- no explicit assignment statements
- variables bound to values only through the association of actual parameters to formal parameters in function calls
- function calls have no side effects
- thus no need to consider global state

Scheme

- Expressions are written in prefix, parenthesized form
- (function arg 1 arg 2 ...arg n)
- (+ 4 5)
- (+ (* 3 4 5) (- 5 3))

Operational semantics: In order to evaluate an expression:

- evaluate function to a function value
- evaluate each arg i in order to obtain its value
- apply the function value to these values

S-expressions

$$\langle \textit{S-expression} \rangle ::= \langle \textit{Atom} \rangle \mid \text{`('}\ \{\langle \textit{S-expression} \rangle\}\ \text{`)'}$$

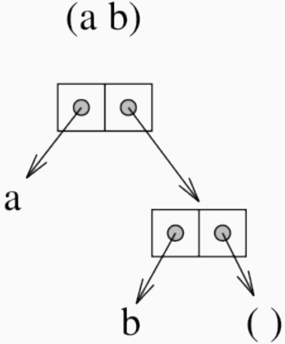
$$\langle \textit{Atom} \rangle ::= \langle \textit{Name} \rangle \mid \langle \textit{Number} \rangle \mid \texttt{\#t} \mid \texttt{\#f}$$

```
#t
()
(a b c)
(a (b c) d)
((a b c) (d e (f)))
(1 (b) 2)
```

Lists have nested structure.

Lists in Scheme

The building blocks for lists are pairs or cons-cells. Lists use the empty list ( ) as an "end-of-list" marker.



Special (Primitive) Functions

- eq?: identity on names (atoms)
- null?: is list empty?
- car: selects first element of list (contents of address part of register)
- cdr: selects rest of list (contents of decrement part of register)
- (cons element list): constructs lists by adding element to front of list
- quote or ': produces constants

Higher-order Functions: map

```
(define map
  (lambda (f l)
    (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l))))))
```

- map takes two arguments: a function and a list
- map builds a new list by applying the function to every element of the (old) list

More on Higher Order Functions

reduce: a higher order function that takes a binary, associative operation and uses it to "roll-up" a list

```
(define reduce
  (lambda (op l id)
    (if (null? l)
        id
        (op (car l) (reduce op (cdr l) id)) )))
```

Lexical Scoping and let, let*, and letrec

- let: binds variables to values (no specific order), and evaluates body e using the bindings; new bindings are not effective during evaluation of any e i .
- let*: binds variables to values in textual order of write-up (left to right, or here: top down); new binding is effective for next e i (nested scopes).
- letrec: bindings of variables to values in no specific order; independent evaluations of all e i to values have to be possible; new bindings effective for all e i ; mainly used for recursive function definitions.

The Y-combinator

Is there a λ-term Y that "computes" a fixed point of a function $F = \lambda f.(\ldots f \ldots)$, i.e., $(YF) = (F(YF))$?

YES. Y is called the fixed point combinator.

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

- (YF)
- $= ((\lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))))\ F)$
- $= ((\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$
- $= (F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))))$
- $= (F(YF))$

Pure

Expressions are *referentially transparent*:

- No mutation
- No side effects
- Same function + same arguments = same value

Laziness

Expressions aren't evaluated until their results are needed

- Easy to define new "syntax"
- Infinite data structures
- Easy to compose functions together

But it complicates understand the time/space usage of your code.

Functions

Choices can also be made using Boolean expressions ("guards"):

```
collatz :: Integer -> Integer
collatz n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise      = 3*n + 1
```

Anonymous functions

But it's annoying to give isPositive a name, since we are probably never going to use it again. Instead, we can use an anonymous function, also known as a lambda abstraction:

```
keepOnlyPositive2 :: [Integer] -> [Integer]
keepOnlyPositive2 xs = filter (\x -> x > 0) xs
```

`\x -> x > 0` is the function which takes a single argument x and outputs whether x is greater than 0.

(the backslash is supposed to look kind of like a lambda with the short leg missing)