Study Sheet

Saturday, December 6, 2014 4:44 PM

Ocaml Programming:

Map:

List.map (fun element of list -> do something to each element) input list
Returns a list of the function applied to all the elements

Filter:

List.filter (fun element of list -> return true or false) input list
Returns a list of only the elements where the function evaluated to true

Fold Right

List.fold_right (fun element of list accumulator -> do something to the accumulator based on the value of the element) input list value of acc if the input is empty
Returns the accumulator after applying the function to each element of the list and seeing what happens to the accumulator

Context-Free Languages

PDA

$M = (Q,\Sigma,\tau,\delta,s,\perp,F)$
Q: all the states
Σ: inupt alphabet
τ: stack alphabet (including input alphabet)
δ: transition function (state, input,pop) (state, push)
s: start state
$\perp$: start of stack symbol
F: accept states

Configurations

A configuration is all the information that you need to know at any one point in time to accurately describe what's happening at that point of the PDA time to accurately describe what's happening at that point of the PDA.

To know this, we need to know
(state, string left to be read, stack content)

Definition of context free languages

Languages that you can create a PDA for

Context free grammars

G = (N,Σ,P,S)
N: all the non terminals
Σ: the terminal symbols
P: set of rules
S: start symbols

Chomsky Normal Form

It's basically a rule such that S never appears on the right hand side of the rule (so that if we have a string of length n and we're at > n non-terminals we know we can't generate the string

Examples of context-free languages

$a^n b^n$ is CFL

Examples of non context-free languages

$a^n b^n a^{mn}$ is not cfl. If we need to remember things "twice" then it's non context free

Important Math Things

Ø: the empty set
ε: the empty string
+: or
▪ : concatenation
* : all possible combinations
∑*: all possible strings over the language
~A: all the elements of ∑* that aren't in A

Turing Machines

Definition of TMS

M = (Q,∑,Γ,⊢,_,δ,s,t,r)
Q: list of states
∑: input alphabet
Γ: tape alphabet
⊢: start of tape symbol
_: blank symbol
δ: transition function (State, on tape) -> (Sate, push tape, move tape head left or right)
s: start state
t: accept state
r: reject state

Now we have tapes!

Configurations

To simulate a TM we need to know

(u,q_a,v)
u: a string of everything that is on the before the tape head
q_a: the curent state
v: a string starting from the tape head to the end of the tape

Definition of recognizable (semidecidable)

Something is semidecidable if we can create a TM whose language is what we want, but it does necessarily reject everything else

Definition of decidable language

Can build a TM that accepts the language and rejects everything else

Church-Turing Thesis

There are many equivalent models of computation. Lambda calculus, post systems, data flow programming, ect are all equivalent

Multitape turning Machines

Same as a normal TM but with two tapes. Equivalent in power of computation.

sn't

Nondeterminisitc Turing Machines

Can have multiple transitions. Equivalent to TMs in power of computation

The halting problem

It is impossible to build a TM that tells if a given TM halts on an input
To do this:

Proof by contradiction:
We can construct a total TM K that halts and accepts an input <M>#x if M halts on x and halts a
rejects otherwise

For any x ∈ {0,1}*, let M_x be the TM s.t. <M_x> = x (the encoding of the TM M = x) if there is one
otherwise let M_x be some fixed TM M* that rejects any input immediately

Now, we construct a TM Ƞ as follows:
On input x:

1. Write <M_x>#x on the tape
2. Simulate K on <M_x>#x
   a. Since K always halts, let's
      i. Loop on K accepting
      ii. Accept if we reach a reject state
3. Is <Ƞ>#<Ƞ> in HP?
   a. Yes:
      i. Then Ƞ halts on <Ƞ>, which means that K rejected <M_{η}>#<Ƞ>, which means t
      <Ƞ>#<Ƞ> would not have been in HP
   b. No:
      i. Then Ƞ loops on <Ƞ>, which means that K accepted <M_{η}>#<Ƞ>, which means
      <Ƞ>#<Ƞ> would have been in HP

The membership problem

It is also impossible to build a TM that tells if a given TM accepts an input

We do a similar proof by contradiction:
We can construct a total TM K that halts and accepts an input <M>#x if M accepts x and halts a
otherwise (even if M loops)

nd

,

that

s that

nd rejects

We can construct from K a TM K' that decides HP, which we know can't happen.
K:
Modify M into M' so that M' accepts when M halts
Write <M'>#x onto the tape of K'

If we get <M>#x ∈ MP, then
M halts on x,
M' accepts x
K halts and accepts
K' halts and accepts

If we get <M>#x ∉ MP, then
M loops on x,
M' loops on x
K halts and rejects on x
K' halts and rejects on x

Arguing Languages undecidable by reduction

I think that this is basically saying that a language is undecidable because if it were decidable, t
halting problem/membership problem, ect. would have to be decidable too. (just a note, decid
accept the empty string is also undecidable)

Streams

Basically like Labview programming

Dataflow diagrams

You have a few base symbols and
then you construct

Implementation of streams

They're basically seen as a list

the
ding if we

(head s)
(tail s)

Fby (first elem)
(fun () -> generate rest of elements)

Stream operations in Ocaml

It's basically just a first element and a promise to compute more if you need to.

Misc:

The pumping lemma

Demon picks k
I pick x,y,z s.t. xzy is in A and |y| = k
Demon picks uvw st y = uvw and v is not empty
I pick i greater than or equal to zero st xuv^iwz is not in A

Logic programming

This is basically finding which values will evaluate so that a given expression is true
NEEDS MORE DETAIL

Lambda Calculus

A model of computation that inspired programming languages like Ocaml. You have these t
lambda expressions
NEEDS MORE DETAIL

Complexity

We might care about
Running time that a program takes.
Space a program takes up
That sort of thing
NEEDS MORE DETAIL

Running time of TMs

TMs are no more than polynomial levels above other programs in running time

Regular Languages:

Languages

A language is a set of strings over an input alphabet Σ such that all things called

A language is a set of strings over an input alphabet Σ, such that all the string are made up of characters in Σ.

Operations on Languages

A ∪ B is the language containing the strings in A and the strings in B
A ∩ B is the language containing the strings that are in both A and B
A* is the language containing any number of the strings in A concatenated together. There are so many strings in this language
A^R is the language containing all the reversed strings in A

DFA

M = (Q,Σ,δ,s,F)
Q: set of all states
Σ: Alphabet
δ: transition function (state, input) -> state
s: Start state
F: Final state (can be more than 1

NFA

Like a DFA except that you don't have to have exactly one transition based on state and input
M = (Q,Σ,Δ,s,F)
Q: set of all states
Σ: Alphabet
Δ: transition function (state, input) -> state
s: Start state
F: Final state (can be more than 1

ε - NFAs that have transitions where you don't read in input symbol

Normalized NFAs - NFAs such that you don't have a transition that points back at the start state

Definition of regular language

If a language is regular, then it can be constructed from regular expressions
A language is regular if we can find a DFA that accepts it

Closure Properties of regular languages

Closure Properties of regular languages

Ø is regular

∑* is regular

If A is finite A is regular

If A is regular then ∑* - A is regular

If A and B are regular then AB is regular

If A and B are regular, then A ∪ B is regular

If A and B are regular, then A ∩ B is regular

If A and B are regular, then A - B is regular

If A is regular , then A* is regular

If A is regular then the A^R is regular

Examples of regular languages

Any string that contains 3 as

Any string that has an odd number of bs

Examples of non-regular languages

$a^n b^n$ is not a regular language if n > 0

Any language that requires some knowledge of prior inputs beyond state is not regular.If we knew what n was it would be a regular language

Regular Expressions

Patterns using only the patterns

$a \in \sum$, ε , Ø, +, ▪ ,*