```
\as -> length as>3 && (all even as || as !! 2 == 99)
PLen (pgt 3) * (PForAll peven + Plx 2 0 (peq 99))
```

DSL for Predicates

Grant Weyburne

What is a predicate?

A predicate is a function that takes a value and returns a boolean

```
type Predicate a = a -> Bool
```

here are some sample predicates:

```
p a = a == "hello"
p a = length a < 3
p a = True
```

What is a predicate?

Here is a more complex one:

```
p a = length a>3 && (all even a || a !! 2 == 99)

sample inputs and outputs:
p [] == False
p [10,12,2] == False
p [10,12,99] == False
p [10,12,99,1] == True
p [10,12,8,22] == True.
```

Purpose of the DSL for Predicates

Aim of this library is to elevate predicates:

- from a simple function that only returns True/False
- to a first class value

Here is what it will do:

- display a predicate
- compare predicates for equality
- compose predicates
- trace the entire process of running a predicate

Predicates in this DSL are represented by expression trees which enables

- navigation through the tree of results
- debugging: see each step in the process
- validation of data structures eg json configurations
- ability to choose to validate as little or as much you want

some useful predicates

```
data Pred a where
  PConst  :: BoolP -> Pred a
  PCmp    :: Ord a => CmpOperator -> a -> Pred a
  PLift   :: String -> (a -> Bool) -> Pred a
  PAnd    :: Pred a -> Pred a -> Pred a
  POr     :: Pred a -> Pred a -> Pred a
  PNot    :: Pred a -> Pred a
  PFst    :: Pred a -> Pred (a,x)
  PSnd    :: Pred a -> Pred (x,a)
  PRange  :: Ord a => a -> a -> Pred a
  PLen    :: Pred Int -> Pred [a]
  PNull   :: Pred [a]
  PForAll :: Pred a -> Pred [a]
  PExists :: Pred a -> Pred [a]
  PHead   :: Pred () -> Pred a -> Pred [a]
  PIx     :: Int -> Pred () -> Pred a -> Pred [a]
  POne    :: Pred () -> Pred (a,[a]) -> Pred a -> Pred [a]
  PFn     :: String -> (a -> b) -> (Pred b) -> Pred a
  PString :: SConv s => Case -> StringOperator -> s -> Pred s
```

Simplest predicate

```
data Pred a where
  PConst  :: BoolP -> Pred a
  ...

data BoolP = TrueP | FalseP | FailP String

ptrue = PConst TrueP
pfalse = PConst FalseP
pfail e = PConst (FailP e)
```

pp* shows the predicate
pe* runs the predicate with the given value and prints

A less simple predicate

```
data Pred a where
  PNull :: Pred [a]
  ...

peu2 PNull []
peu2 PNull [5,7,10]
```

An even less simple predicate

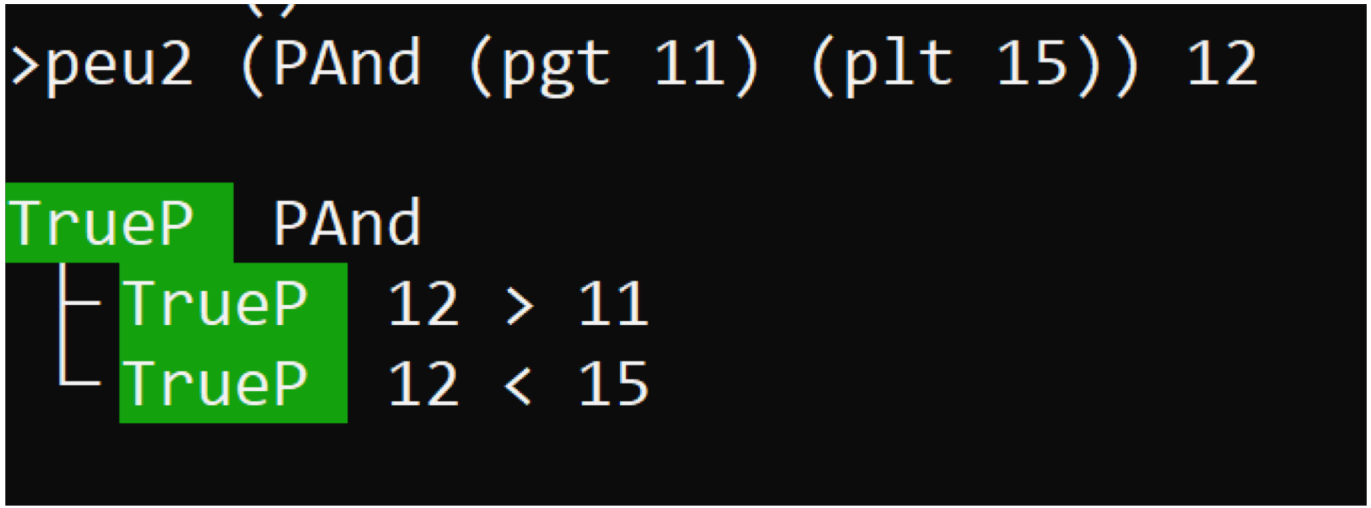
```
data Pred a where
  PCmp :: Ord a => CmpOperator -> a -> Pred a
  ...

-- comparison operators: less than / less than or equal / etc
data CmpOperator = Lt | Le | Ge | Gt | Ne | Eq

-- greater than predicate
pgt :: Ord a => a -> Pred a
pgt = PCmp Gt

-- equal to predicate
peq :: Ord a => a -> Pred a
peq = PCmp Eq

...
```

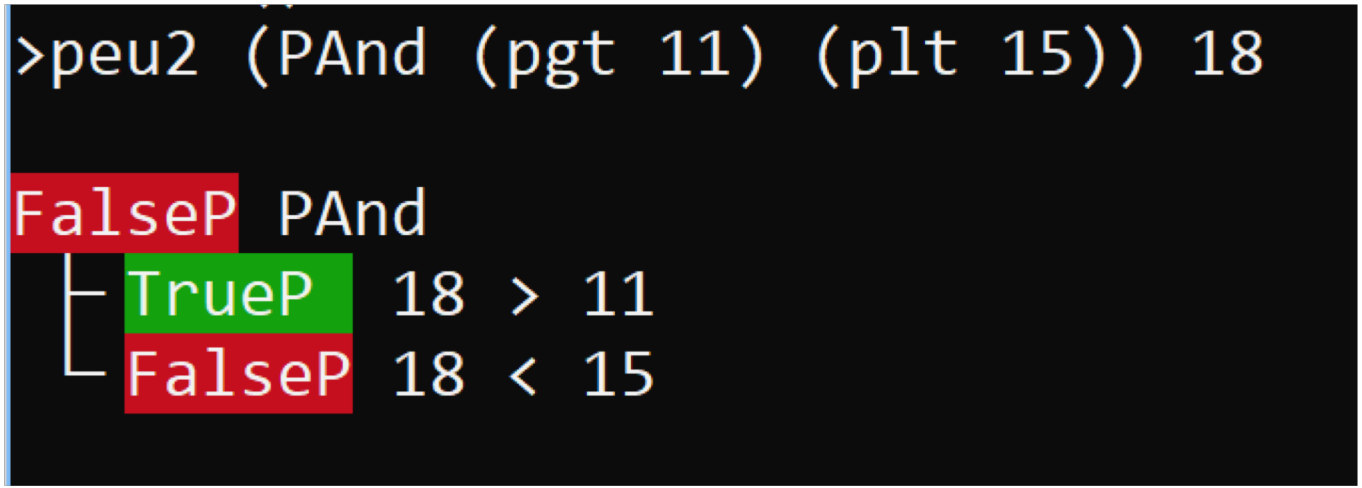
Create your own predicate!

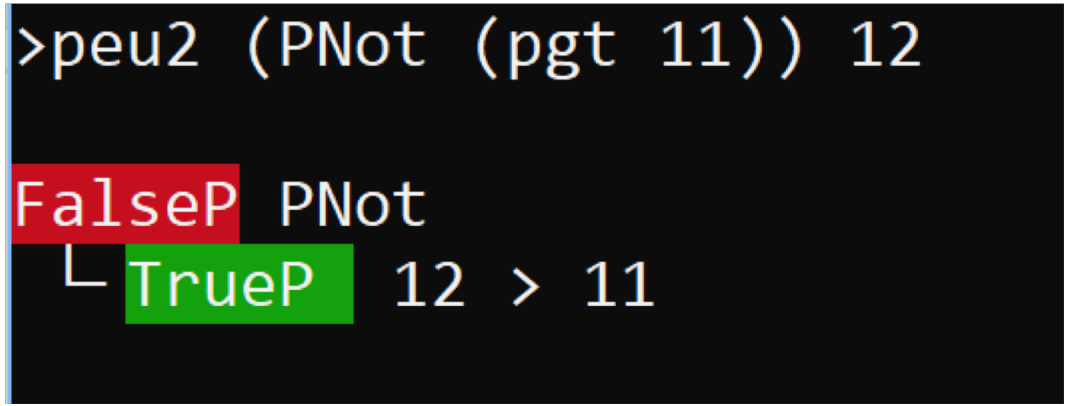
```
data Pred a where
  PLift :: String -> (a -> Bool) -> Pred a
  ...

peven :: Pred Int
peven = PLift "even" even
```

Boolean combinators

```
data Pred a where
  PAnd    :: Pred a -> Pred a -> Pred a
  POr     :: Pred a -> Pred a -> Pred a
  PNot    :: Pred a -> Pred a
  ...
-- show peu2 pe2 pev2
peu2 (PAnd (pgt 11) (plt 15)) 12
peu2 (PAnd (pgt 11) (plt 15)) 18
peu2 (pgt 11) 12
peu2 (PNot (pgt 11)) 12
```


>peu2 (PAnd (pgt 11) (plt 15)) 12
TrueP PAnd
TrueP 12 > 11
TrueP 12 < 15


>peu2 (PAnd (pgt 11) (plt 15)) 18
FalseP PAnd
TrueP 18 > 11
FalseP 18 < 15


>peu2 (PNot (pgt 11)) 12
FalseP PNot
└ TrueP 12 > 11

predicates on tuples

```
data Pred a where
  PFst :: Pred a -> Pred (a,x)
  PSnd :: Pred b -> Pred (x,b)
  ...



peu2 (PFst (peq 'z')) ('z',99)
peu2 (PSnd (plt 12)) ('z',99)
```

Num instance

```
instance Num (Pred a) where
  (+) = POr
  (*) = PAnd
  negate = PNot
  p - q = p `POr` PNot q
  fromInteger n | n == 0 = pfalse
                | n == 1 = ptrue
                | otherwise = pfail ""
```

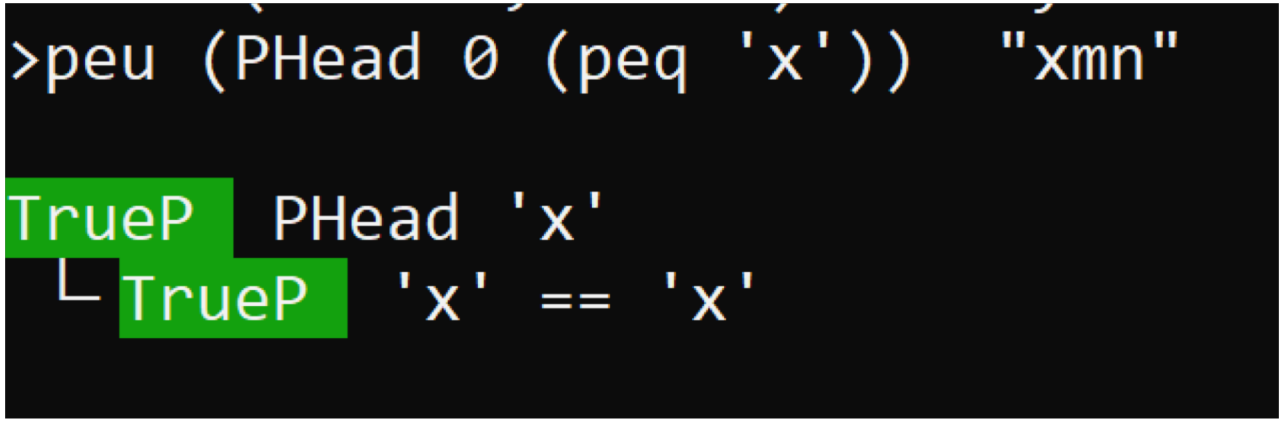
```
peu2 (PAnd (pgt 11) (plt 15)) 12  ==> peu2 (pgt 11 * plt 15) 12
peu2 (PNot (pgt 11)) 12           ==> peu2 (-pgt 11) 12
```

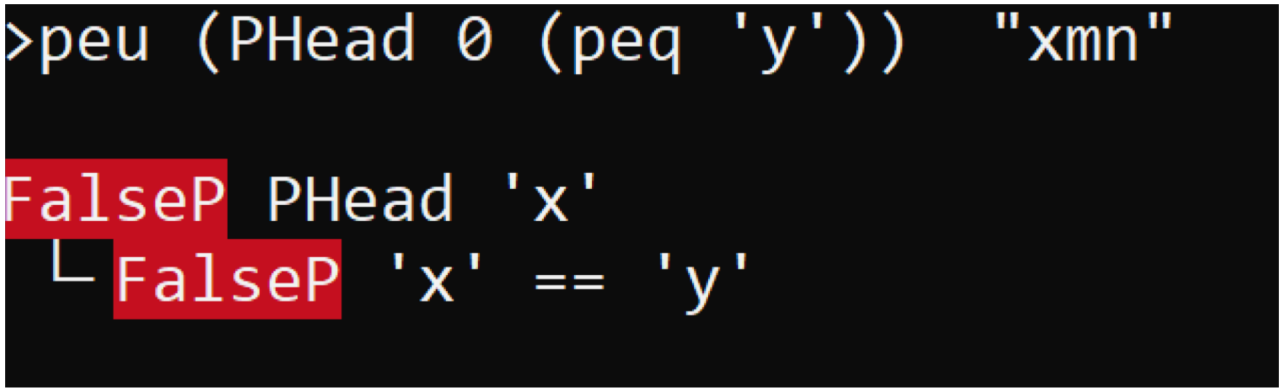
partial predicates

we need to handle the failure cases

```
data Pred a where
  PHead :: Pred () -> Pred a -> Pred [a]
  ...
```

```
peu (PHead 0 (peq 'x'))  "xmn"
peu (PHead 0 (peq 'y'))  "xmn"
peu (PHead 0 (peq 'y'))  ""
```


>peu (PHead 0 (peq 'x')) "xmn"
TrueP PHead 'x'
└TrueP 'x' == 'x'


>peu (PHead 0 (peq 'y')) "xmn"
FalseP PHead 'x'
└FalseP 'x' == 'y'

```
>peu (PHead 0 (peq 'y'))  ""

FalseP PHead empty
```

partial predicates

```
data Pred a where
  PIx :: Int -> Pred () -> Pred a -> Pred [a]
  ...


peu (PIx 3 pfalse (peq 'd')) "abcdef"
peu (PIx 3 pfalse (peq 'x')) "abcdef"
peu (PIx 3 pfalse (peq 'x')) "ab"


peu (PIx 3999 (pfalse' "hey!") (peq 'e')) "abcdef"
```

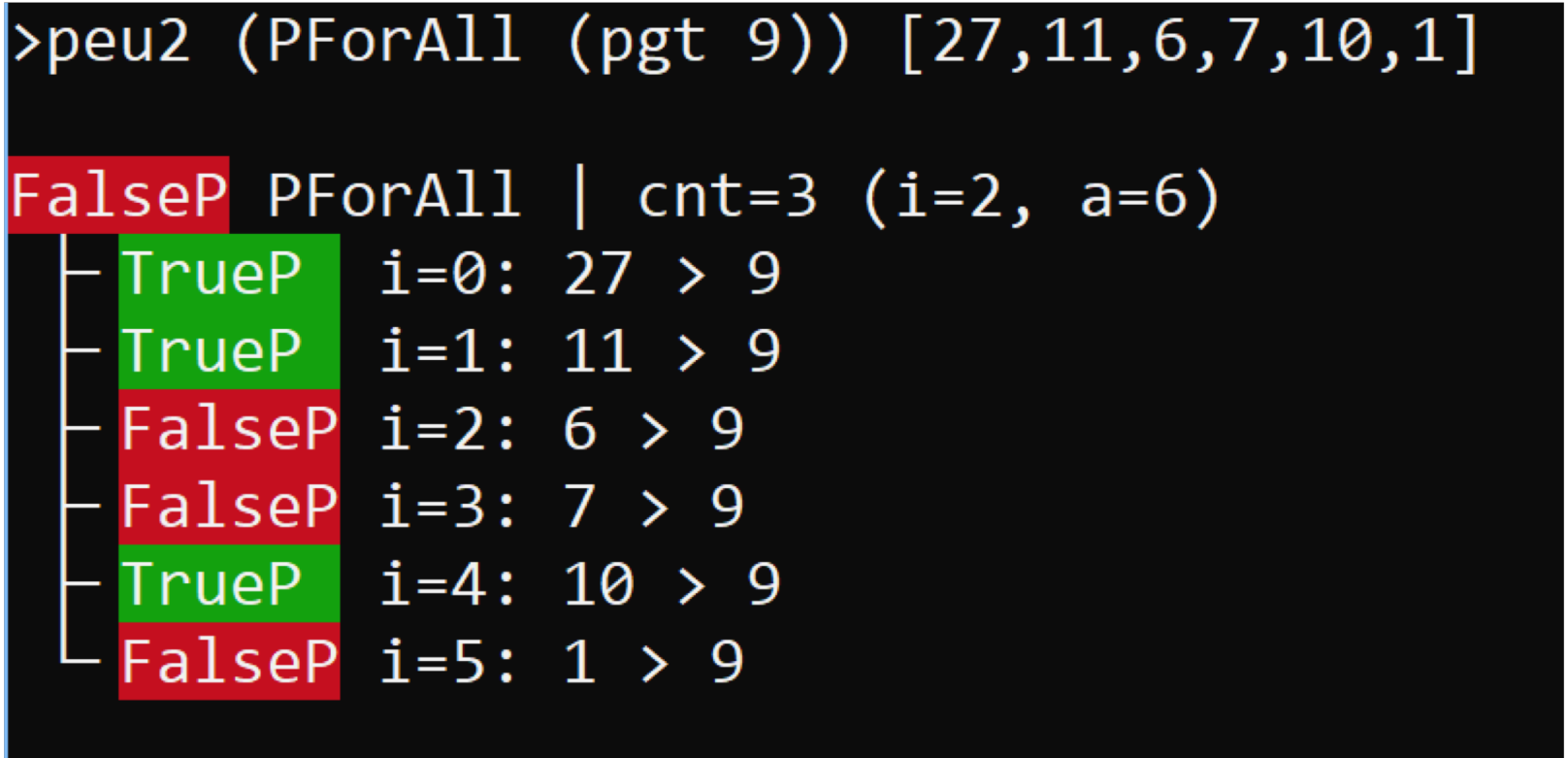
```
>peu (PIx 3 pfalse (peq 'd')) "abcdef"

TrueP  PIx 3 'd'
  └─TrueP  'd' == 'd'
```

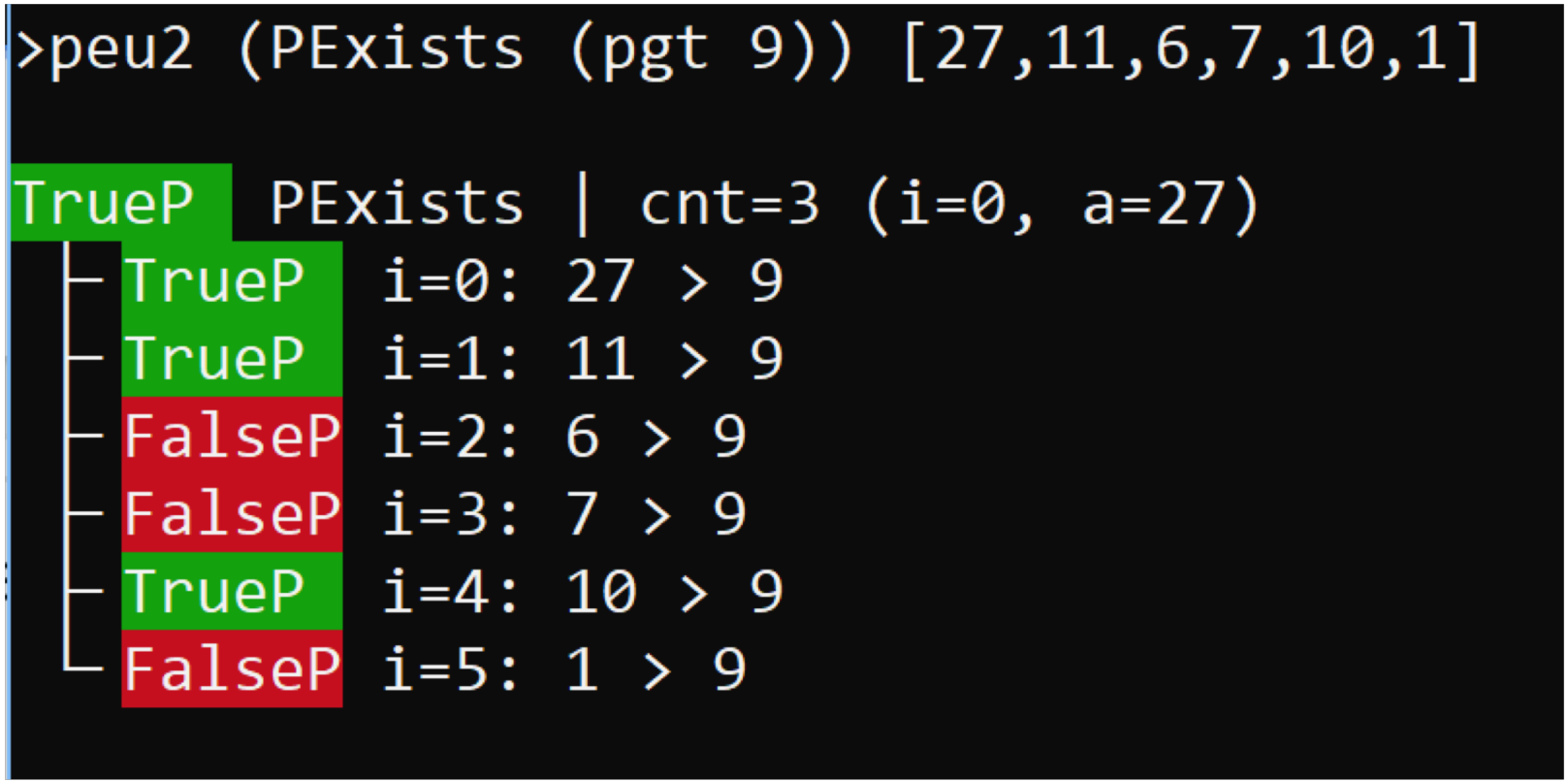
```
>peu (PIx 3 pfalse (peq 'x')) "abcdef"

FalseP PIx 3 'd'
 └─FalseP 'd' == 'x'
```

```
>peu (PIx 3999 (pfalse' "hey!") (peq 'e')) "abcdef"

FalseP PIx 3999 not found | hey!
```

forall and exists predicates

```
data Pred a where
  PForAll :: Pred a -> Pred [a]
  PExists :: Pred a -> Pred [a]
  ...


peu2 (PForAll (pgt 9)) [27,11,6,7,10,1]

peu2 (PExists (pgt 9)) [27,11,6,7,10,1]
```


>peu2 (PForAll (pgt 9)) [27,11,6,7,10,1]
FalseP PForAll | cnt=3 (i=2, a=6)
TrueP i=0: 27 > 9
TrueP i=1: 11 > 9
FalseP i=2: 6 > 9
FalseP i=3: 7 > 9
TrueP i=4: 10 > 9
FalseP i=5: 1 > 9


>peu2 (PExists (pgt 9)) [27,11,6,7,10,1]
TrueP PExists | cnt=3 (i=0, a=27)
TrueP i=0: 27 > 9
TrueP i=1: 11 > 9
FalseP i=2: 6 > 9
FalseP i=3: 7 > 9
TrueP i=4: 10 > 9
FalseP i=5: 1 > 9

```
>peu2 (PForAll (PRange 4 7)) [1..10]

FalseP PForAll | cnt=6 (i=0, a=1)
  ├─ FalseP i=0: 1 < 4 (Under)
  ├─ FalseP i=1: 2 < 4 (Under)
  ├─ FalseP i=2: 3 < 4 (Under)
  ├─ TrueP  i=3: 4 == [4..7]
  ├─ TrueP  i=4: 5 == [4..7]
  ├─ TrueP  i=5: 6 == [4..7]
  ├─ TrueP  i=6: 7 == [4..7]
  ├─ FalseP i=7: 8 > 7 (Over)
  ├─ FalseP i=8: 9 > 7 (Over)
  └─ FalseP i=9: 10 > 7 (Over)
```

predicate from first slide

```
haskell predicate
p as = length as>3 && (all even as || as !! 2 == 99)

predicate from the dsl
p = PLen (pgt 3) * (PForAll peven + PIx 2 0 (peq 99))

-- show predicate expression and ppu
ppv $ PLen (pgt 3) * (PForAll peven + PIx 2 0 (peq 99))

peu2 p [] == False
peu2 p [10,12,2] == False
peu2 p [10,12,99] == False
peu2 p [10,12,99,1] == True
peu2 p [10,12,8,22] == True.
```

```
>ppu $ PLen (pgt 3) * (PForAll peven + PIx 2 0 (peq 99))

PAnd
  ├─PLen
  │  └─a > 3
  └─POr
     ├─PForAll
     │  └─PLift even
     └─PIx 2
        ├─PConst FalseP
        └─a == 99
```

```
>peu2 (PLen (pgt 3) * (PForAll peven + PIx 2 0 (peq 99))) [10,12,99]

FalseP PAnd
  ├─FalseP PLen 3 as=[10,12,99]
  │  └─FalseP 3 > 3
  └─TrueP  POr
     ├─FalseP PForAll | cnt=1 (i=2, a=99)
     │  ├─TrueP  i=0: PLift even | a=10
     │  ├─TrueP  i=1: PLift even | a=12
     │  └─FalseP i=2: PLift even | a=99
     └─TrueP  PIx 2 99
        └─TrueP  99 == 99
```

```
>peu2 (PLen (pgt 3) * (PForAll peven + PIx 2 0 (peq 99))) [10,12,8,22]

TrueP  PAnd
├─TrueP  PLen 4 as=[10,12,8,22]
│ └─TrueP  4 > 3
└─TrueP  POr
  ├─TrueP  PForAll
  │ ├─TrueP  i=0: PLift even | a=10
  │ ├─TrueP  i=1: PLift even | a=12
  │ ├─TrueP  i=2: PLift even | a=8
  │ └─TrueP  i=3: PLift even | a=22
  └─FalseP PIx 2 8
    └─FalseP 8 == 99
```

PString

```
data Pred a where
  PString :: Case -> StringOperator -> s -> Pred z s
  ...

data StringOperator = SNone | SPrefix | SInfix | SSuffix
```

```
sinfix = PString CI  SInfix
sci    = PString CI  SNone
scs    = PString CS SNone  -- default

pe2 (scs "abc") "abC"
pe2 "abc" "abC"
pe2 (sinfix "abc") "XabCYY"
```

PLinear: validate structures & configuration

```
PLinear :: Rigid -> [(Pred Int, Pred a)] -> Pred [a]
```

specify the number of times each predicate should occur

most common are:

- `preq = peq 1`
- `popt = peq 0 + peq 1`
- `pij i j = PRange i j`
- `pnever = peq 0`

fails if two different predicates consume the same value
fails if a predicate isn't used up the right number of times

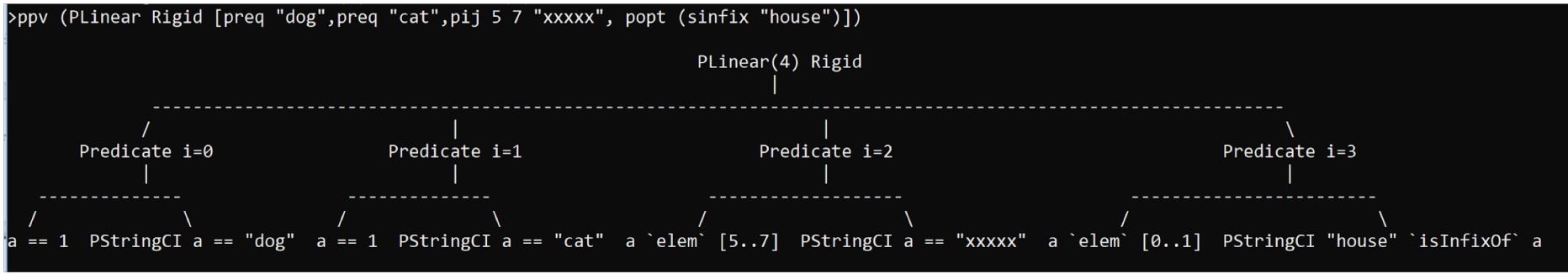
```
Rigid (catches all errors) vs Loose (allows extra values)
peu1 (PLinear Rigid [preq "dog",preq "cat",pij 5 7 "house"]) ["cat","dog","cat","mouse"]
peu1 (PLinear Rigid [preq "dog",preq "cat",preq "house"]) ["cat","dog","cat","mouse"]
peu1 (PLinear Rigid [preq "dog",preq "cat",preq "house"]) ["cat","dog","house"]
```

```
>peu1 (PLinear Rigid [preq "dog",preq "cat",pij 5 7 "house"]) ["cat","dog","cat","mouse"]

FalseP PLinear Failed Pred [Int] | errors(1): NoMatch 3
  ├─FalseP Predicates | PZipAnd | PZipExact | (bad,good)=(2,1)
  │  └─FalseP PLift and | a=[True,False,False]
  │     ├─TrueP  i=0
  │     │  ├─TrueP  1 == 1
  │     │  └─PStringCI a == "dog"
  │     ├─FalseP i=1
  │     │  ├─FalseP 2 > 1 (Over)
  │     │  └─PStringCI a == "cat"
  │     └─FalseP i=2
  │        ├─FalseP 0 < 5 (Under)
  │        └─PStringCI a == "house"
  ├─TrueP  PLinear | OneMatch 0 a="cat" cnt=1 (i=1, a="cat")
  │  ├─FalseP i=0: PStringCI "cat" == "dog"
  │  ├─TrueP  i=1: PStringCI "cat" == "cat"
  │  └─FalseP i=2: PStringCI "cat" == "house"
  ├─TrueP  PLinear | OneMatch 1 a="dog" cnt=1 (i=0, a="dog")
  │  ├─TrueP  i=0: PStringCI "dog" == "dog"
  │  ├─FalseP i=1: PStringCI "dog" == "cat"
  │  └─FalseP i=2: PStringCI "dog" == "house"
  ├─TrueP  PLinear | OneMatch 2 a="cat" cnt=1 (i=1, a="cat")
  │  ├─FalseP i=0: PStringCI "cat" == "dog"
  │  ├─TrueP  i=1: PStringCI "cat" == "cat"
  │  └─FalseP i=2: PStringCI "cat" == "house"
  └─FalseP PLinear NoMatch 3 a="mouse"
     ├─FalseP i=0: PStringCI "mouse" == "dog"
     ├─FalseP i=1: PStringCI "mouse" == "cat"
     └─FalseP i=2: PStringCI "mouse" == "house"
```

```
>peu1 (PLinear Rigid [preq "dog",preq "cat",preq "house"]) ["cat","dog","cat","mouse"]

FalseP PLinear Failed Pred [Int] | errors(1): NoMatch 3
  ├─FalseP Predicates | PZipAnd | PZipExact | (bad,good)=(2,1)
  │  └─FalseP PLift and | a=[True,False,False]
  │     ├─TrueP  i=0
  │     │  ├─TrueP  1 == 1
  │     │  └─PStringCI a == "dog"
  │     ├─FalseP i=1
  │     │  ├─FalseP 2 > 1 (Over)
  │     │  └─PStringCI a == "cat"
  │     └─FalseP i=2
  │        ├─FalseP 0 < 1 (Under)
  │        └─PStringCI a == "house"
  ├─TrueP  PLinear | OneMatch 0 a="cat" cnt=1 (i=1, a="cat")
  │  ├─FalseP i=0: PStringCI "cat" == "dog"
  │  ├─TrueP  i=1: PStringCI "cat" == "cat"
  │  └─FalseP i=2: PStringCI "cat" == "house"
  ├─TrueP  PLinear | OneMatch 1 a="dog" cnt=1 (i=0, a="dog")
  │  ├─TrueP  i=0: PStringCI "dog" == "dog"
  │  ├─FalseP i=1: PStringCI "dog" == "cat"
  │  └─FalseP i=2: PStringCI "dog" == "house"
  ├─TrueP  PLinear | OneMatch 2 a="cat" cnt=1 (i=1, a="cat")
  │  ├─FalseP i=0: PStringCI "cat" == "dog"
  │  ├─TrueP  i=1: PStringCI "cat" == "cat"
  │  └─FalseP i=2: PStringCI "cat" == "house"
  └─FalseP PLinear NoMatch 3 a="mouse"
     ├─FalseP i=0: PStringCI "mouse" == "dog"
     ├─FalseP i=1: PStringCI "mouse" == "cat"
     └─FalseP i=2: PStringCI "mouse" == "house"
```

```
>peu1 (PLinear Rigid [preq "dog",preq "cat",preq "house"]) ["cat","dog","house"]

TrueP  PLinear
  ├─TrueP  Predicates | PZipAnd | PZipExact | (bad,good)=(0,3)
  │  └─TrueP  PLift and | a=[True,True,True]
  │     ├─TrueP  i=0
  │     │  ├─TrueP  1 == 1
  │     │  └─PStringCI a == "dog"
  │     ├─TrueP  i=1
  │     │  ├─TrueP  1 == 1
  │     │  └─PStringCI a == "cat"
  │     └─TrueP  i=2
  │        ├─TrueP  1 == 1
  │        └─PStringCI a == "house"
  ├─TrueP  PLinear | OneMatch 0 a="cat" cnt=1 (i=1, a="cat")
  │  ├─FalseP i=0: PStringCI "cat" == "dog"
  │  ├─TrueP  i=1: PStringCI "cat" == "cat"
  │  └─FalseP i=2: PStringCI "cat" == "house"
  ├─TrueP  PLinear | OneMatch 1 a="dog" cnt=1 (i=0, a="dog")
  │  ├─TrueP  i=0: PStringCI "dog" == "dog"
  │  ├─FalseP i=1: PStringCI "dog" == "cat"
  │  └─FalseP i=2: PStringCI "dog" == "house"
  └─TrueP  PLinear | OneMatch 2 a="house" cnt=1 (i=2, a="house")
     ├─FalseP i=0: PStringCI "house" == "dog"
     ├─FalseP i=1: PStringCI "house" == "cat"
     └─TrueP  i=2: PStringCI "house" == "house"
```

display predicate



```
data Pred a where
  PDist :: Case -> String -> Pred Int -> Pred String
  ...

peu2 (PDist CS "production" (plt 4)) "production"
```

plinearDist

plinearDist :: Int -> [Dist z String] -> Pred z [String]

leverages PLinear but is used when you dont have control on all the values and are looking for cases where someone has mistyped or fat fingered.

eg typed "Production" vs "production" or "produktion" or "PRODUCTION"
ie words that are in close proximity

peu1 (plinearDist 2 [dopt "production"]) ["produczlon"]
peu1 (plinearDist 2 [dopt "production"]) ["PRODUCTION"] -- zero distance but wrong case
peu1 (plinearDist 2 [dopt "production"]) ["notnotproduction"]

```
>peu1 (plinearDist 2 [dopt "production"]) ["produczIon"]

FalseP PLinear Failed Pred [Int]
├─FalseP Predicates | PZipAnd | PZipExact | (bad,good)=(1,2)
│ └─FalseP PLift and | a=[True,False,True]
│   ├─TrueP  i=0
│   │ ├─TrueP  0 == [0..1]
│   │ └─PStringCS a == "production"
│   ├─FalseP i=1
│   │ ├─FalseP 1 > 0 (Over)
│   │ └─PDistCI "production"
│   │   └─a `elem` [1..2]
│   └─TrueP  i=2
│     ├─TrueP  0 == 0
│     └─PAnd
│       ├─PStringCI a == "production"
│       └─PNot
│         └─PStringCS a == "production"
└─TrueP  PLinear | OneMatch 0 a="produczIon" cnt=1 (i=1, a="produczIon")
  ├─FalseP i=0: PStringCS "produczIon" == "production"
  ├─TrueP  i=1: PDistCI | dist=1 | s=production | t=produczIon
  │ └─TrueP  1 == [1..2]
  └─FalseP i=2: PAnd
    ├─FalseP PStringCI "produczIon" == "production"
    └─TrueP  PNot
      └─FalseP PStringCS "produczIon" == "production"
```

```
>peu1 (plinearDist 2 [dopt "production"]) ["PRODUCTION"]

FalseP PLinear Failed Pred [Int]
 ├─FalseP Predicates | PZipAnd | PZipExact | (bad,good)=(1,2)
 │  └─FalseP PLift and | a=[True,True,False]
 │     ├─TrueP  i=0
 │     │  ├─TrueP  0 == [0..1]
 │     │  └─PStringCS a == "production"
 │     ├─TrueP  i=1
 │     │  ├─TrueP  0 == 0
 │     │  └─PDistCI "production"
 │     │     └─a `elem` [1..2]
 │     └─FalseP i=2
 │        ├─FalseP 1 > 0 (Over)
 │        └─PAnd
 │           ├─PStringCI a == "production"
 │           └─PNot
 │              └─PStringCS a == "production"
 └─TrueP  PLinear | OneMatch 0 a="PRODUCTION" cnt=1 (i=2, a="PRODUCTION")
    ├─FalseP i=0: PStringCS "PRODUCTION" == "production"
    ├─FalseP i=1: PDistCI | dist=0 | s=production | t=PRODUCTION
    │  └─FalseP 0 < 1 (Under)
    └─TrueP  i=2: PAnd
       ├─TrueP  PStringCI "PRODUCTION" == "production"
       └─TrueP  PNot
          └─FalseP PStringCS "PRODUCTION" == "production"
```

```
>peu1 (plinearDist 2 [dopt "production"]) ["notnotproduction"]

TrueP  PLinear
 ├─TrueP  Predicates | PZipAnd | PZipExact | (bad,good)=(0,3)
 │ └─TrueP  PLift and | a=[True,True,True]
 │   ├─TrueP  i=0
 │   │ ├─TrueP  0 == [0..1]
 │   │ └─PStringCS a == "production"
 │   ├─TrueP  i=1
 │   │ ├─TrueP  0 == 0
 │   │ └─PDistCI "production"
 │   │   └─a `elem` [1..2]
 │   └─TrueP  i=2
 │     ├─TrueP  0 == 0
 │     └─PAnd
 │       ├─PStringCI a == "production"
 │       └─PNot
 │         └─PStringCS a == "production"
 └─TrueP  PLinear NoMatch 0 a="notnotproduction"
   ├─FalseP i=0: PStringCS "notnotproduction" == "production"
   ├─FalseP i=1: PDistCI | dist=6 | s=production | t=notnotproduction
   │ └─FalseP 6 > 2 (Over)
   └─FalseP i=2: PAnd
     ├─FalseP PStringCI "notnotproduction" == "production"
     └─TrueP  PNot
       └─FalseP PStringCS "notnotproduction" == "production"
```

Json

```
Object  -- dictionary indexed by a string key
Array   -- list indexed by integer
String  -- primitive values
Number
Bool
Null

data JPath =
     JPKey String   -- Object has a String as the key
   | JPIndex Int    -- Array has an Integer as the index into the array
   | JPValue Value  -- any json value
```

Json predicates

```
data Pred a where
  PJsonP :: [JPath] -> Pred z () -> Pred z Value -> Pred z Value
  PJsonKey :: Pred z String -> Pred z [(NonEmpty JPath, Value)] -> Pred z Value
  ...


 jprtPretty json1

 --path to "abbrev" key
 peu1 (pjsonKeyOne "abbrev" 1) json1

 peu (pjsonKeyOne "abbrev" (sinfix "8879")) json1

 peu (PJsonP [JPKey "glossary",JPKey "GlossDiv",JPKey "GlossList",JPKey "GlossEntry",JPKey "Abbrev"] 0 1)
 json1

 -- misspelling shows where you it wrong
 peu (PJsonP [JPKey "glossary",JPKey "GlossDiv",JPKey "GlossList",JPKey "GlossEnxtry",JPKey "Abbrev"] 0 1)
 json1
```

```
>peu1 (pjsonKeyOne "abbrev" 1) json1

TrueP  PJsonKey
 ├─TrueP  POne String "ISO 8879:1986"
 │  └─TrueP  PConst a=String "ISO 8879:1986"
 └─Debugging jpaths
    └─i=0 | key=Abbrev | [JPKey "glossary",JPKey "GlossDiv",JPKey "GlossList",JPKey "GlossEntry",JPKey "Abbrev"]
```

```
>peu (pjsonKeyOne "abbrev" $ sinfix "iso") json1

TrueP  PJsonKey
 ├─TrueP  POne String "ISO 8879:1986"
 │  └─TrueP  PStringCI String "iso" `isInfixOf` String "ISO 8879:1986"
 └─Debugging jpaths
    └─i=0 | key=Abbrev
```

```
>peu (PJsonP [JPKey "glossary",JPKey "GlossDiv",JPKey "GlossList",JPKey "GlossEntry",JPKey "Abbrev"] 0 1) json1

TrueP  PJsonP
 ├─TrueP  matched
 └─key "glossary"
    └─key "GlossDiv"
       └─key "GlossList"
          └─key "GlossEntry"
             └─key "Abbrev"
                └─TrueP  matched complete path
```

```
>peu (PJsonP [JPKey "glossary",JPKey "GlossDiv",JPKey "GlossLisxt",JPKey "GlossEntry",JPKey "Abbrev"] 0 1) json1

FalseP PJsonP
 ├─FalseP valid keys are [title,GlossList]
 └─key "glossary"
   └─key "GlossDiv"
     └─valid keys are [title,GlossList]
```

Vinyl Regex predicate

```
data Pred a where
   PRegexV
  :: Rec (RE Char) rs                       -- heterogeneous list of regex exprs
     -> Pred z String                       -- failure predicate
     -> Pred z (Rec W.Identity rs, String)  -- success predicate: hlist of results and leftovers
     -> Pred z String

runs each regex in turn against a string until failure or completes successfully


  :& delimits each element and RNil ends the list

  string we want to parse
  "9 1233.987   141.213.124.19ZZ"

  list of regular expresssions
  res :: Rec (RE Char) '[Int, (), Rational, (), IP Int]
  res = digit :& spaces1 :& ratio :& spaces1 :& ipaddr :& RNil
```

```
>peu2 (PRegexV (digit :& spaces1 :& ratio :& spaces1 :& ipaddr :& RNil) 0 1)
"9 1233.987   141.213.124.19ZZ"

TrueP  PRegexV (5) | matched all | leftover=ZZ
 ├─TrueP  PConst a=({9, (), 1233987 % 1000, (), IP:141.213.124.19},"ZZ")
 └─matched all | leftover=ZZ
    ├─i=0 | a=9 | used=9 | remaining= 1233.987   141.213.124.19ZZ
    ├─i=1 | a=() | used=  | remaining=1233.987   141.213.124.19ZZ
    ├─i=2 | a=1233987 % 1000 | used=1233.987 | remaining=   141.213.124.19ZZ
    ├─i=3 | a=() | used=    | remaining=141.213.124.19ZZ
    └─i=4 | a=IP:141.213.124.19 | used=141.213.124.19 | remaining=ZZ
```

```
>peu2 (PRegexV (digit :& spaces1 :& ratio :& spaces1 :& ipaddr :& RNil) 0 $
PBoth (prx (pgt 9999)) PNull) "9 1233.987   141.213.124.19ZZ"

FalseP PRegexV (5) | matched all | leftover=ZZ
 ├─FalseP PBoth
 │  ├─FalseP PFn rx | a={9, (), 1233987 % 1000, (), IP:141.213.124.19} | b=9
 │  │  └─FalseP 9 > 9999
 │  └─FalseP PNull length=2 as="ZZ"
 └─matched all | leftover=ZZ
    ├─i=0 | a=9 | used=9 | remaining= 1233.987   141.213.124.19ZZ
    ├─i=1 | a=() | used=  | remaining=1233.987   141.213.124.19ZZ
    ├─i=2 | a=1233987 % 1000 | used=1233.987 | remaining=   141.213.124.19ZZ
    ├─i=3 | a=() | used=    | remaining=141.213.124.19ZZ
    └─i=4 | a=IP:141.213.124.19 | used=141.213.124.19 | remaining=ZZ
```

```
>peu2 (PRegexV (digit :& spaces1 :& ratio :& spaces1 :& ipaddr :& RNil) 0 $
PBoth (pr2 (pgt 99)) (PLen (plt 5))) "9 1233.987   141.213.124.19ZZ"

TrueP  PRegexV (5) | matched all | leftover=ZZ
 ├─TrueP  PBoth
 │  ├─TrueP  PFn regex2 | a={9, (), 1233987 % 1000, (), IP:141.213.124.19} |
 b=1233987 % 1000
 │  │  └─TrueP  1233987 % 1000 > 99 % 1
 │  └─TrueP  PLen 2 as="ZZ"
 │     └─TrueP  2 < 5
 └─matched all | leftover=ZZ
    ├─i=0 | a=9 | used=9 | remaining= 1233.987   141.213.124.19ZZ
    ├─i=1 | a=() | used=  | remaining=1233.987   141.213.124.19ZZ
    ├─i=2 | a=1233987 % 1000 | used=1233.987 | remaining=   141.213.124.19ZZ
    ├─i=3 | a=() | used=    | remaining=141.213.124.19ZZ
    └─i=4 | a=IP:141.213.124.19 | used=141.213.124.19 | remaining=ZZ
```

Summary: What can do with this DSL?

We can:

- compose predicates
  - to handle arbitrarily complex structures
- display predicates
  - as an expression tree
- compare predicates
  - for equality
- trace the entire process of running the predicate
- enable data validation
  - and configuration validation
- supercharge json
  - so we treat the structure as data and use predicates to validate