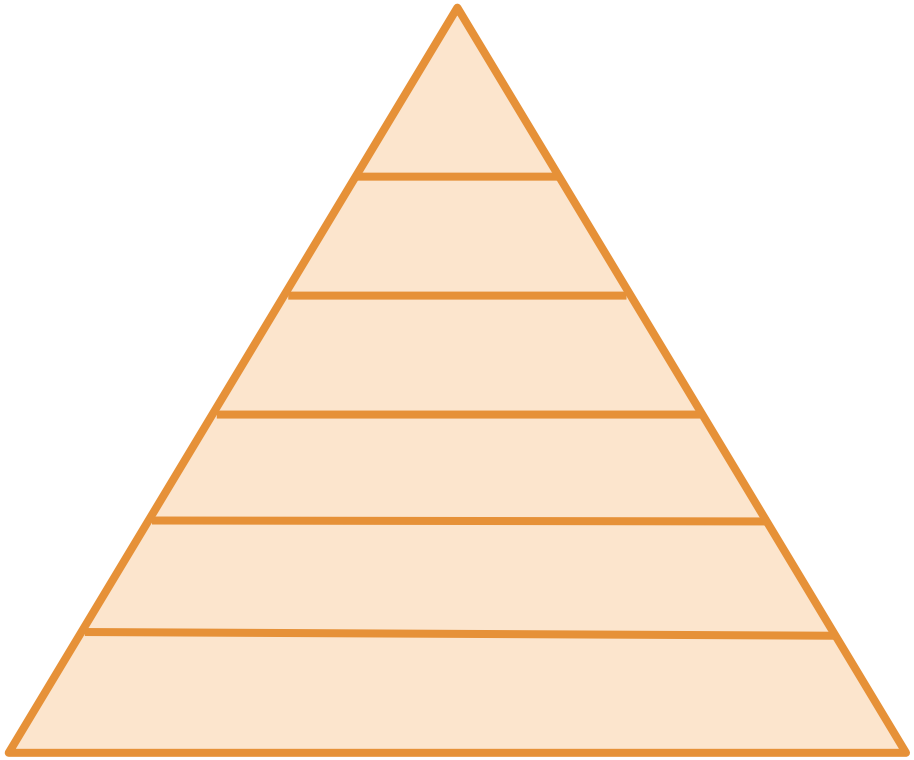
psql=# select (λ  . ???);

Functional Programming … in SQL ?

λ  • ???

Maslow's Hierarchy of Lambda Jam



Idris / Agda / Coq

Haskell / OCaml

Scala / Clojure

...

JavaScript / **SQL**

C / C++ / Assembly

λ 🛢 • ???

~~Functional~~ Programming … in SQL ?

λ 🛢 • ???

general purpose
programming environment

specialised tool for
data munging

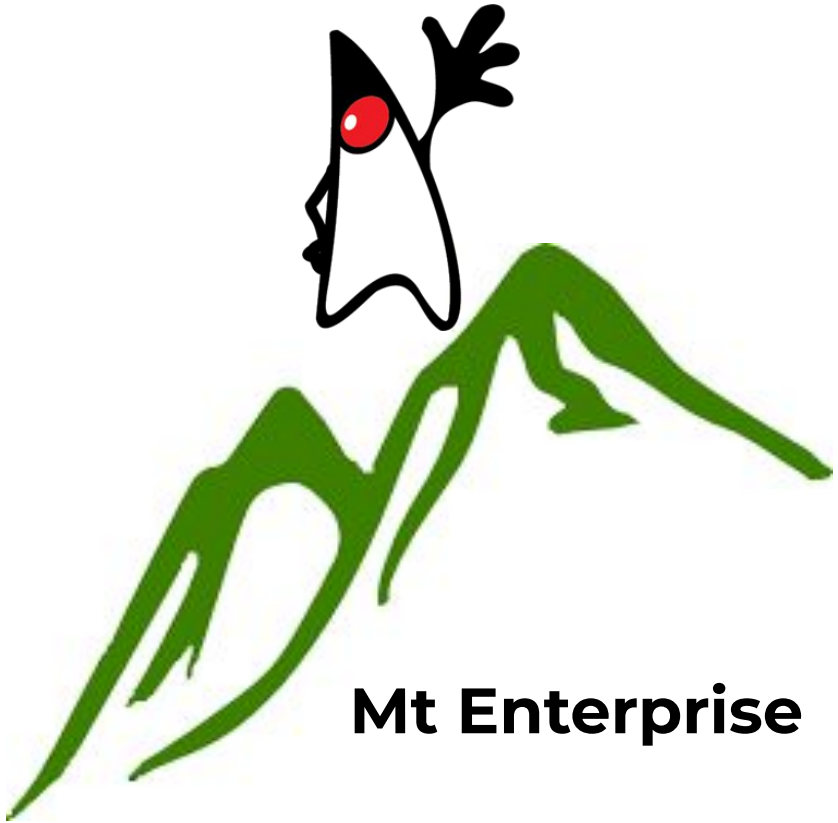
"I think I'll do all my complex data munging over here…"

"… that way I can just treat this like a simpleton manager of flat files."

λ • ???

“thou shalt have
n tiers,

and
in the middle one
doest thou all
the *real* work”



Mt Enterprise

λ 🛢 • ???

Functional Programming
… in SQL ?

λ 🛢 • ???

```
select to_json(
         ( username,
           given_name,
           family_name,
           user_type
         ) :: api.user
       )
  from app_user;
```

```
{
   "username": "bruce",
   "givenName": "Bruce",
   "familyName": "Lee",
   "userType": "power"
}
```

λ • ???

```
select to_json(api.user('sam'))
           as user_details,
       api.whitelabel_theme('unbranded')
           as brand_theme,
       api.account_unread_messages('customer1')
           as unread_messages;
```

λ 🗄 • ???

```
select p.name,
       p.dob,
       a.reason,
       a.date
  from patient p
       left join lateral
         last_appointment(
           p.patient_id
         ) a
         on true;
```

```
create function
    last_appointment(
        pid bigint
    ) returns table(...)
as $$
    select *
      from appointment a
     where a.patient_id
               = pid
  order by a.date desc
     limit 1;
$$ language sql stable;
```

λ • ???

Approaching SQL functionally

1. “functional core, imperative shell”
   - heavy lifting at the outermost level …
   - … **and inside that, referentially transparent functions**

2. **functions of one SQL statement**
   - the SQL equivalent of a single Lisp / lambda calculus expression
   - `'language sql'`, not `'language plpgsql'`

λ 🛢 • ???



So, FP == map / reduce, yeah?



λ 🗄 • ???

```
map :: (a -> b) -> [a] -> [b]
```

```
select f(a)
  from source a;
```

λ 🛢 • ???

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
select a
  from source a
 where f(a);
```

λ 🗄 • ???

```
reduce :: ((a -> b' -> b'), b', (b' -> b))
          -> [a] -> b
```

```
select avg(a)
  from source a;
```

λ 🗄 • ???

```
(++) :: [a] -> [a] -> [a]
```

```
select a
  from source_1 a
 union all
select b
  from source_2 b;
```

λ 🛢 • ???

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

```
select b
  from source a
       join lateral (
            select f(a)
       ) b on true;
```

λ 🛢 • ???

`concatMap` lateral join example

```
select o.entrant
  from competition c
       join lateral (
         select repeat(
                  c.entrant,
                  c.count
                );
       ) o on true;
```

```
entrant | count
--------|------
  john  |   3
   bob  |   0
 alice  |   1
```

```
entrant
-------
  john
  john
  john
 alice
```

λ • ???

How to do FP in SQL

1. **write (mostly) pure functions**
   - pure functions for data transformation (`'select'`)
   - pure functions for predicates (`'where'`)
   - impure functions for dataset-returning functions (`'from'`)

2. **compose them**

λ 🗄 • ???

```
create function full_name(
    given_name text,
    family_name text
) returns text
as $$
    select given_name
           || ' '
           || family_name;


$$ language sql immutable;
```

```
create function full_name(
    pid bigint
) returns text
as $$
    select given_name
           || ' '
           || family_name
      from person
     where person_id
           = pid;
$$ language sql stable;
```

λ 🗄 • ???

```
select to_person_json(p) from person p;
```

```
create function to_person_json(
    p person
) returns jsonb
as $$
    select jsonb_build_object(
              'id',   p.id,
              'name', full_name( p.given_name,
                                 p.family_name
                               ),
              ...
           );
$$ language sql immutable;
```

λ 🗄 • ???

```
create function
  early_adopter(
    id bigint
  ) returns boolean
as $$
    select id < 100;
$$ language sql immutable;
```

```
explain analyze verbose
 select *
   from se_user
  where id < 100;
```

```
explain analyze verbose
 select *
   from se_user
  where early_adopter(
          se_user_id
        );
```

λ 🗄 • ???

```
map g . map f == map (g . f)
```

```
select g(b)
  from ( select f(a)
           from a
       ) b;
```

```
select g( f(a) );
```

λ  • ???

```
filter g . filter f == filter (f &&& g)
```

```
select a2
  from ( select a1
           from src a1
          where f(a1)
       ) a2
 where g(a2);
```

```
select a
  from src a
 where f(a)
       and g(a);
```

λ • ???

```
concatMap f . map g == concatMap (f . g)
```

```
select c
  from ( select g(a)
           from src a
       ) b
       join lateral (
           select f(a)
       ) c on true;
```

```
select c
  from src a
       join lateral (
        select f(g(a))
       ) c on true;
```

λ 🛢 • ???

Doing it right: organise like real code

- for re-use, separation-of-concerns, abstraction, etc
  - extract functions
  - … including table-returning functions
  - use views (including updateable views)
  - use common table expressions (CTEs – `'with'` clauses)

- with an API

- **with tests ! ! !**

λ 🛢 • ???

Beyond: model effects & change as data

- model time explicitly
- try to model **irrefutable** (therefore immutable) facts
- if the business process is that user creation sends an email, insert into an `email_out` table as part of the transaction
  - `listen / notify` is your friend

λ 🛢 • ???

Taking SQL-as-the-app-layer to extremes

How much app layer
does a CRUD app really need
when the database can already
munge data and do JSON?

λ 🗄 • ???

pg_crud_ops

```
[{
  "school": {
    "get_school": {}
  },
  "classes": {
    "get_classes": {}
  },
  "stats": {
    "get_statistics": {
      "start_date": "2018-12-01",
      "end_date": "2018-12-08"
    }
  }
}]
```

```
select api_student.get_school();
```

```
select api_student.get_classes();
```

```
select api_student.get_statistics (
    "start_date" => "2018-12-01",
    "end_date"   => "2018-12-08"
);
```

λ  • ???

```
psql=# \?
```

Questions ?

@sroberton / sam@samroberton.com

λ 🛢 • ???