A cheatsheet on Discourse.

The three gates of speech

Before you speak, let your words pass through three gates.

- At the first gate, ask yourself, is it true.
- At the second gate ask, is it necessary.
- At the third gate ask, is it kind.

Rogerian rhetoric

1. You should attempt to re-express your target's position so clearly, vividly, and fairly that your target says, "Thanks, I wish I'd thought of putting it that way."
2. You should list any points of agreement (especially if they are not matters of general or widespread agreement).
3. You should mention anything you have learned from your target.
4. Only then are you permitted to say so much as a word of rebuttal or criticism.

— Dennett's version of Rapoport's Rules

Argument Ranking

- - High-level generators - Disagreements that remain when everyone understands exactly what's being argued, and agrees on what all the evidence says, but have vague and hard-to-define reasons for disagreeing.
- - Operationalizing - Where both parties understand they're in a cooperative effort to fix exactly what they're arguing about.
- - Survey of evidence - Not trying to devastate the other person with a mountain of facts and start looking at the studies and arguments on both sides and figuring out what kind of complex picture they paint.
- - Disputing definitions - Argument hinges on the meaning of words, or whether something counts as a member of a category or not.
- - Single Studies - Better than scattered facts, proving they at least looked into the issue formally.
- - Demands for rigor - Attempts to demand that an opposing argument be held to such strict standards that nothing could possibly clear the bar.
- - Single Facts- One fact, which admittedly does support their argument, but presented as if it solves the debate in and of itself.

- - Gotchas - Short claims that purport to be devastating proof that one side can't possibly be right.
- - Social shaming- A demand for listeners to place someone outside the boundary of whom deserve to be heard.

"How to apologize: Quickly, specifically, sincerely."

— Kevin Kelly

Arguments

- **Ad baculum** : Argument relying on an appeal to fear or a threat.
- **Ad ignorantiam** : Argument relying on people's ignorance.
- **Ad populum** : Argument relying on sentimental weakness.
- **Ad verecundiam** : Argument relying on the the words of an "expert", or authority.
- **Ex silentio** : Argument relying on ignorance.
- **Ex nihilo** : An argument that bears no relation to the previous topic of discussion.
- **Non sequitur** : An inference that does not follow from established premises or evidence.

Responses

- **Akrasia** : State of acting against one's better judgment.
- **Connotation** : Emotional association with a word.
- **Intransigence** : Refusal to change one's views or to agree about something.
- **Inferential distance** : Gap between the background knowledge and epistemology of a person trying to explain an idea, and the background knowledge and epistemology of the person trying to understand it.
- **Straw man** : Creating a false or made up scenario and then attacking it. Painting your opponent with false colors only deflects the purpose of the argument.
- **Steel man** : To steelman is to address the strongest possible variant or the most charitable interpretation of an idea, rather than the most available phrasings.
- **Red herring** : A diversion from the active topic.
- **Rationalization** : Starts from a conclusion, and then works backward to arrive at arguments apparently favouring that conclusion. Rationalization argues for a side already selected.
- **Dogpiling** : A disagreement wherein one person says something wrong or offensive, and a large number of people comment in response to tell them how wrong they are, and continue to disparage the original commenter beyond any reasonable time limit.
- **Grandstanding** : An action that is intended to make people notice and admire you, behaving in a way that makes people pay attention to you

instead of thinking about more important matters.
- **Whataboutism** : An attempt to discredit an opponent's position by charging them with hypocrisy without directly refuting or disproving their argument.
- **Dissensus** : The deliberate avoidance of consensus.

Beliefs

- **Belief** : The mental state in which an individual holds a proposition to be true.
- **Priors** : The beliefs an agent holds regarding a fact, hypothesis or consequence, before being presented with evidence.
- **Alief** : An independent source of emotional reaction which can coexist with a contradictory belief. Example The fear felt when a monster jumps out of the darkness in a scary movie is based on the alief that the monster is about to attack you, even though you believe that it cannot.
- **Proper belief** : Requires observations, gets updated upon encountering new evidence, and provides practical benefit in anticipated experience.
- **Improper belief** : Is a belief that isn't concerned with describing the territory. Note that the fact that a belief just happens to be true doesn't mean you're right to have it. If you buy a lottery ticket, certain that it's a winning ticket (for no reason), and it happens to be, believing that was still a mistake.
- **Belief in belief** : Where it is difficult to believe a thing, it is often much easier to believe that you ought to believe it. Were you to really believe and not just believe in belief, the consequences of error would be much more severe. When someone makes up excuses in advance, it would seem to require that belief, and belief in belief, have become unsynchronized.
- **A Priori** : Knowledge which we can be sure of without any empirical evidence(evidence from our senses). So, knowledge that you could realize if you were just a mind floating in a void unconnected to a body.

  "A leader is best when people barely know they exists, when their work is done, their aim fulfilled, people will say: we did it ourselves."

  — (Lao Tse), (Dao De Jing)

The first principle of Wikipedia etiquette has been said to be **Assume Good Faith**, also they **Be Bold, but not Reckless**.

Wrong discourse

- Answer: Jumping into a conversation with endless unapplicable, unrealistic or unrelated answers to the question.

- Question: Spouting accusations while cowardly hiding behind the claim of just asking questions, and ignoring the answers. Asking loaded questions.

Good discourse

- Answer: A clear and honest response to the central point of a question without an aggressive attempt to convince.
- Question: question asked with the intention to be fair, open, and honest, regardless of the outcome of the interaction.

Social rules are expected to be broken from time to time, in that regard they are different from a code of conduct.

Response Ranking

- - Central point - Commit to refute explicitly the central point.
- - Refutation - Argue a conflicting passage, explain why it's mistaken.
- - Counterargument - Contradict with added reasoning or evidence.
- - Contradiction - State the opposing case, what.
- - Responding to Tone - Responding to the author's tone, how.
- - Ad Hominem - Attacking the author directly, who.

Interaction Ranking

Discussion

- - Release - Initiating a discussion on the lessons learnt from a project.
- - Update - Presenting the recent development of a personal experience, ongoing event or work in progress.
- - Soapbox - Spontaneous and or enthusiastic posts about a general topic of interest or finding.

Low-Effort

- - Rant- Venting frustration publicly without explicitly looking to have a conversation about the matter.
- - Shitpost - Aggressively or ironically looking for the biggest reaction with the least effort possible. Includes subtoots and vague-posting.

Emotional Reaction

- **Seduction** - You are led to feel that the fulfillment of your dreams depends on your doing what the other is encouraging you to do.
- **Alignment** - The interests of the system are presented as fulfilling your emotional needs. You are led to feel that your survival, your viability

in society or your very identity depends on your doing what the other is requiring of you.

- **Reduction** - Complex subjects are reduced to a single, emotionally charged issue.
- **Polarization** - Issues are presented in such a way that you are either right or wrong. You are told that any dialogue between different perspectives is suspect, dangerous or simply not permissible.
- **Marginalization** - You are made to feel that your own interests (or interests that run counter to the interests of the other) are inconsequential.
- **Framing** - The terms of a debate are set so that issues that threaten the system cannot be articulated or discussed. You are led to ignore aspects of the issue that may be vitally important to your own interests but are contrary to the interests of the other that is seeking to make you act in their interests.

Quotes

> "Kings speak for the realm, governors for the state, popes for the church. Indeed, the titled, as titled, cannot speak **with** annyone." — James P. Carse, Finite and Infinite Games

> "Instead of trying to prove your opponent wrong, try to see in what sense he might be right." — Robert Nozick, Anarchy, State, and Utopia

> "I don't argue: I just say what I know or what I believe, as the case may be." — John W. Cohan

> "You should mention anything you have learned from your target."

LICENSE

2012-06-01 - Applications intelligentes: catégorisation et recommandations de textes courts

Cet article est à propos de la catégorisation de tweet, j'espère que les non-anglophone arriverons à suivre avec les schémas

Préface

Last year I was conducting some research about ways to improve the usability of Twitter. At the same time I was told about work about machine learning similar to what StumbleUpon does. I though that a similar feature in the context of Twitter or any reader apps would be awesome. So I started digging the problem and find out that even without strong machine-learning knowledge it was possible to come up with solutions that in theory could give good results. There might be better, more deeper solutions of the problem, what I want is to outline the algorithm used to achieve such application. But I did not implement it because I believe that without GraphDBs the solution will a) not be scalable b) not be flexible enough to add new data c) not flexible enough to implement (new) algorithms, plus I wanted to have my own application (suite) to implement this features in.

Figure 1: De Lapide Philosophico.

Application intelligente

What I call a smart application is an application that goes forward user needs and guess or at least try to guess and learn from users actions/inputs and environnement. Given the context of twitter application, I am thinking about users and tweets categorization and users and tweets recommendation. I put the features in this order because recommendation use categorization to take its decisions, but it is not the only information it uses. Best smart features are seamlessly integrated to the current use of the application, that's what I try to do in the following post. First how does it look like ?

Figure 2: The sidebar displays the user's labels, shows the list of category, the gray labels in messages are generated

Catégorisation de tweet et d'utilisateurs

Given a tweet, I wanted to put it in a category, so that at the end of the day I have properly organised list of tweets #LifeHacks kind of tweets, #Music tweets, #RandomSmartLinks and #Python stuff properly organized. While at the same time, given a tweet being able to know what it is talking about without having to click on it or read it. This are really interesting information and is, if doable and applicable, promising for longer texts, and it is. Given a Twitter user there is a number of tweets that are tagged plus the network of followings which have also tagged their tweet, we have the first level of tweet features, that can be used to build category with. Each user is associated with the labels he or she used in her or his tweets and the tweets of her or his network.

Figure 3: Label network of level one

Every red-purple edge has a weight but it's not represented. With this data design we can already add most interesting labels in user profile, but tweet hashtags are not always interesting as features of users and tweets, for this matter we can use a white-list or black-list that can be manually or automatically generated with most popular hashtags but we can also use semantic expansion, based on a corpus of hierarchical labels that can be easily retrieved from Wikipedia and well organized websites like github, bitbucket, delicious, Amazon, newsgroups, forums, dmoz and the like. It's true that the extracted data might need to be cleaned but a simple blacklist based on Wiktionary will make the manual work way much easier. Similarly labeling texts aka. features extraction from random websites can be made easier using this blacklist method. By semantic expansion, I mean retrieving the most probable generic word that is linked to a hashtag, using a hierarchical taxonomy make this easier, a networked taxonomy make the algorithm a bit more complex but doable. Given this data/knowledge we can build the second level and third level of labels. The second level of labels are the labels extracted from content of tweets or links. The third level is labels we find with semantic expansion which are represented in green in the following graphic, they are just like any other labels:

Figure 4: labels of level two and three discovered by smart algorithms

With this knowledge we can compute tags for both user and tweets and if there is some trouble to find the right label, for instance given the word «Python», we need to distinguish the animal from the language, we can use the other labels to find out using a collocation weight edges in the network of labels. The fourth level of labels, is given a tweet with hastags stripped, perform with it a full-text search on a database of Wikipedia extended with linked articles and link tweets with the best wikipedia article results. Tweet categorization is then just a matter of matching the user interests with the labels associated with tweets. Recommandation d'utilisateur et de tweet The easiest way to recommend users is by looking up an user's network of followings and retrieve the most common users. Similarly it is also possible to recommend tweets based on the «+fav» and «RT» of your network. But this is bound to the first degree of your network, you can't discover a tweet or an user at 6 degrees of separation. To discover tweets and user you can also use the item-item algorithm used to power Amazon recommendation algorithm, but you might hit a dimension problem. If you have categorized your user and tweets, this can also be achieved to do user and tweet recommendation by matching user or tweets with similar labels. The following give an example of a possible implementation to recommend users:

Tweets recommendation is similar.

More like this

- FIXME
- FIXME
- FIXME

2015-01-01 - Debuter avec la base de donnée clef-valeur bsddb

Remarque je préfère aujourd'hui wiredtiger.

Berkeley database est la base de donnée la plus utilisé dans le monde d'après ses créateurs. Pourquoi? Car elle est très flexible. Ici je vais pas m'étaler sur les différentes fonctionnalités. Je défriche la création d'une base de donnée et la création d'index.

Basics

Le backend btree est très bien pour créer un index ordonnée.

```
import os
import shutil

from bsddb3.db import *

from json import dumps
from json import loads

# reset the database if it already exists
if os.path.exists('/tmp/bsddb'):
    shutil.rmtree('/tmp/bsddb')
os.makedirs('/tmp/bsddb')

# initialize the database
env = DBEnv()
env.open(
    '/tmp/bsddb',
    DB_CREATE | DB_INIT_MPOOL,
    0
)


def compare(a, b):
    # at initialisation time a & b are empty strings
    # those can't be deserialized by json
    if a and b:
        # a and b are string keys
```

```
        # In this case comparing them as is, is non-sens
        # they must be deserialized
        a = loads(a.decode('ascii'))
        b = loads(b.decode('ascii'))
        if a < b:
            return -1
        elif a == b:
            return 0
        else:
            return 1
    return 0


index = DB(env)
# set the function to compare keys
index.set_bt_compare(compare)
index.open('index', None, DB_BTREE, DB_CREATE, 0)


# populate the database

# keep track of all the values that are in the database
# in order of insertion
values = list()

i = 0  # keep track of insertion order
def populate(*key):
    global i
    values.append(list(key))
    key = dumps(key).encode('ascii')
    value = dumps(i).encode('ascii')
    index.put(key, value)
    i += 1

populate(1, 0, 0)
populate(3, 0, 0)
populate(0, 2, 1)
populate(2, 0, 0)
populate(0, 2, 0)

# fetch all index values in order
all = list()
cursor = index.cursor()
next = cursor.first()
while next:
    key, value = next
```

```
    key = loads(key.decode('ascii'))
    value = loads(value.decode('ascii'))
    all.append(key)
    next = cursor.next()

print('initial keys\t', sorted(values))
print('cursor keys\t', all)
assert sorted(values) == all
```

Un autre exemple plus parlant qui fait intervenir deux bases de données:

```
import os
import shutil

from bsddb3.db import *

from json import dumps as json_dumps
from json import loads as json_loads


def dumps(value):
    return json_dumps(value).encode('ascii')


def loads(value):
    return json_loads(value.decode('ascii'))


# reset the database if it already exists
if os.path.exists('/tmp/bsddb'):
    shutil.rmtree('/tmp/bsddb')
os.makedirs('/tmp/bsddb')

# initialize the database
env = DBEnv()
env.open(
    '/tmp/bsddb',
    DB_CREATE | DB_INIT_MPOOL,
    0
)


# create articles database
articles = DB(env)
# DB_HASH is recommanded for database
# that can not fit fully in memory
articles.open('articles', None, DB_HASH, DB_CREATE, 0)
```

```
# create index database
def compare(a, b):
    if a and b:
        a = loads(a)
        b = loads(b)
        if a < b:
            return -1
        elif a == b:
            return 0
        else:
            return 1
    return 0


def duplicate(a, b):
    # this compares ascii bytes values of the index
    # this is supposed to be the default comparaison
    # but the bsddb fails to do so
    if a < b:
        return -1
    elif a == b:
        return 0
    else:
        return 1

index = DB(env)

index.set_bt_compare(compare)
index.set_dup_compare(duplicate)
index.open('index', None, DB_BTREE, DB_CREATE, 0)


# populate the database
def populate(title, body, published_at, modified_at):
    value = dict(
        title=title,
        body=body,
        published_at=published_at,
        modified_at=modified_at,
    )
    value = dumps(value)
    # save article in articles database
    # here title is used as a key but it can
    # be anything memorable.
```

```
    key = title.encode('ascii')
    articles.put(key, value)

    # index the article
    key = (published_at, modified_at)
    key = dumps(key)
    value = title.encode('ascii')
    index.put(key, value)


body = 'a k/v store is a dictionary a set of key/value associations'
populate('Getting started with kv store (1/2)', body, 1, 5)
body = 'the gist of the practice of using kv stores is to build'
body += ' a schema on top of it using string patterns'
populate('Getting started with kv store (2/2)', body, 2, 2)

# for some reason the following article is put in the database
# before the followings even if it is published later
body = 'Wiretiger is kind of the successor of bsddb'
populate('Behold wiredtiger database (1/2)', body, 6, 10)

body = 'bsddb has still room to be put to good use.'
populate('Almighty bsddb (1/2)', body, 4, 3)
body = 'bsddb is stable!'
populate('Almighty bsddb (2/2)', body, 5, 2)

# the following articles will have the same index key
body = 'Working with wiredtiger is similar. Take advantage of its'
body += 'own features'
populate('Behold wiredtiger database (2/2)', body, 7, 0)
body = 'Good question'
populate('Is it worth the trouble?', body, 7, 0)


print('* All articles in chronological order')
cursor = index.cursor()
next = cursor.first()
while next:
    key, value = next
    key = loads(key)
    # the value is the title, this can also be used
    # to fetch the associated article in articles database
    title = value.decode('ascii')
    published_at, modified_at = key
    print('**', published_at, modified_at, title)
    next = cursor.next()
```

```
print('\n* All articles published between 4 and 6 inclusive in chronological order')

cursor = index.cursor()
next = cursor.set_range(dumps((4, 0)))
while next:
    key, value = next
    key = loads(key)
    if 4 <= key[0] <= 6:
        title = value.decode('ascii')
        published_at, modified_at = key
        print('**', published_at, modified_at, title)
        next = cursor.next()
    else:
        break


print('\n* Last three articles in anti-chronological order with body')
cursor = index.cursor()
previous = cursor.last()  # browse the index in reverse order
for i in range(3):
    # we don't need to deserialize the key
    _, value = previous
    # the value is the title, it is used to fetch the full article
    # datastructure. This is a join.
    article = articles.get(value)
    article = loads(article)
    title = value.decode('ascii')
    body = article['body']
    print('**', '%s: %s' % (title, body))
    previous = cursor.prev()
```

Comme c'est un peu très souvent qu'il faut construire des indexes bsddb fournis des routines pour aider:

```
import os
import shutil

from bsddb3.db import *

from json import dumps as json_dumps
from json import loads as json_loads


def dumps(value):
    return json_dumps(value).encode('ascii')
```

```
def loads(value):
    return json_loads(value.decode('ascii'))


# reset the database if it already exists
if os.path.exists('/tmp/bsddb'):
    shutil.rmtree('/tmp/bsddb')
os.makedirs('/tmp/bsddb')

# initialize the database
env = DBEnv()
env.open(
    '/tmp/bsddb',
    DB_CREATE | DB_INIT_MPOOL,
    0
)


# create articles database
articles = DB(env)
# DB_HASH is recommanded for database
# that can not fit fully in memory
articles.open('articles', None, DB_HASH, DB_CREATE, 0)


# create index database
def compare(a, b):
    if a and b:
        a = loads(a)
        b = loads(b)
        if a < b:
            return -1
        elif a == b:
            return 0
        else:
            return 1
    return 0


def duplicate(a, b):
    # this compares ascii bytes values of the index
    if a and b:
        if a < b:
            return -1
```

```
        elif a == b:
            return 0
        else:
            return 1
    return 0

index = DB(env)

index.set_bt_compare(compare)
index.set_dup_compare(duplicate)
index.open('index', None, DB_BTREE, DB_CREATE, 0)


# XXX: this is the main change to the code
def callback(key, value):
    # this will keep the index automatically up-to-date
    value = loads(value)
    published_at = value['published_at']
    modified_at = value['modified_at']
    key = (published_at, modified_at)
    key = dumps(key)
    return key
articles.associate(index, callback)


def populate(title, body, published_at, modified_at):
    value = dict(
        title=title,
        body=body,
        published_at=published_at,
        modified_at=modified_at,
    )
    value = dumps(value)
    # save article in articles database
    key = title.encode('ascii')
    articles.put(key, value)
    # no need to update the index database
    # it's done by bsddb


body = 'a k/v store is a dictionary a set of key/value associations'
populate('Getting started with kv store (1/2)', body, 1, 5)
body = 'the gist of the practice of using kv stores is to build'
body += ' a schema on top of it using string patterns'
populate('Getting started with kv store (2/2)', body, 2, 2)
body = 'Wiretiger is kind of the successor of bsddb'
```

```
populate('Behold wiredtiger database (1/2)', body, 6, 10)
body = 'bsddb has still room to be put to good use.'
populate('Almighty bsddb (1/2)', body, 4, 3)
body = 'bsddb is stable!'
populate('Almighty bsddb (2/2)', body, 5, 2)
body = 'Working with wiredtiger is similar. Take advantage of its'
body += 'own features'
populate('Behold wiredtiger database (2/2)', body, 7, 0)
body = 'Good question'
populate('Is it worth the trouble?', body, 7, 0)


# XXX: index cursor return the primary database value,
# a json serialized article dictionary, no need
# to do the join manually
print('* All articles in chronological order')
cursor = index.cursor()
next = cursor.first()
while next:
    # XXX: just like before key is the **index key**
    # for that article
    key, value = next
    # XXX: but the value is the serialized article value of
    # instead of the primary key of the article
    # which means that the join was done by bsddb
    article = loads(value)
    title = article['title']
    published_at = article['published_at']
    modified_at = article['modified_at']
    print('**', published_at, modified_at, title)
    next = cursor.next()


print('\n* All articles published between 4 and 6 inclusive in chronological order')

cursor = index.cursor()
next = cursor.set_range(dumps((4, 0)))
while next:
    key, value = next
    key = loads(key)
    if 4 <= key[0] <= 6:
        article = loads(value)
        title = article['title']
        published_at = article['published_at']
        modified_at = article['modified_at']
        print('**', published_at, modified_at, title)
```

```
        next = cursor.next()
    else:
        break


print('\n* Last three articles in anti-chronological order with body')
cursor = index.cursor()
previous = cursor.last()  # browse the index in reverse order
for i in range(3):
    key, value = previous
    article = loads(value)
    title = value.decode('ascii')
    body = article['body']
    print('**', '%s: %s' % (title, body))
    previous = cursor.prev()


# XXX: The get method of the secondary database ie. the index
# also returns the primary data instead of the primary key
# of the article
print('\n* Retrieve the article published the 6/10')
# XXX: Using index.pget will return (primary_key, primary_data)
# here it's not required to retrieve the primary key
# since it's also available as part of primary_data
value = index.get(dumps((6, 10)))
article = loads(value)
title = article['title']
published_at = article['published_at']
modified_at = article['modified_at']
print('**', published_at, modified_at, title)
value = index.pget(dumps((6, 10)))

# XXX: Let's delete that article
key = title.encode('ascii')
articles.delete(key)

# XXX: Try to retrieve it from the index just like before
value = index.get(dumps((6, 10)))
assert value is None  # it's not anymore in the secondary index
```

Multi processus

Apparament on peux faire du multiprocess avec BerkeleyDB, voilà un exemple pas très concluant d'après mes tests:

```
#!/usr/bin/env python3
import os
from time import sleep
from datetime import datetime
from hashlib import md5
from json import dumps as _dumps
from json import loads as _loads
from shutil import rmtree
from bsddb3.db import *
from sys import exit
from multiprocessing import Process


def dumps(v):
    return _dumps(v).encode('ascii')


def loads(v):
    return _loads(v.decode('ascii'))


# reset database
path = '/tmp/bsddb-mutliprocess'
if os.path.exists(path):
    rmtree(path)
os.makedirs(path)


def opendb():
    # create and open database environment
    env = DBEnv()
    env.set_cachesize(1, 0)
    env.set_tx_max(4)
    flags = (DB_INIT_LOG |
             DB_INIT_LOCK |
             DB_INIT_TXN |
             DB_CREATE |
             DB_INIT_MPOOL
    )
    env.open(
        path,
        flags,
        0
    )
    # create database
    txn = env.txn_begin(flags=DB_TXN_SNAPSHOT)
```

```
    flags = DB_CREATE | DB_MULTIVERSION
    counter = DB(env)
    counter.open('counter', None, DB_BTREE, flags, 0, txn=txn)
    txn.commit()
    return env, counter


def writer(identifier):
    print('writer', identifier, 'running')
    env, counter = opendb()
    # open database and update it
    txn = env.txn_begin(flags=DB_TXN_SNAPSHOT)
    counter.put(identifier, dumps(0), txn=txn)
    txn.commit()

    for i in range(100):
        # print('writer', identifier, '@ iteration', i)
        txn = env.txn_begin(flags=DB_TXN_SNAPSHOT)
        count = counter.get(identifier, txn=txn)
        count = loads(count)
        count += i
        counter.put(identifier, dumps(count), txn=txn)
        txn.commit()
        sleep(0.2)
    print('writer', identifier, 'finished')
    counter.close()
    env.close()
    exit(0)


def reader(identifiers):
    # print('reader running')
    env, counter = opendb()
    for i in range(100):
        for identifier in identifiers:
            txn = env.txn_begin(flags=DB_TXN_SNAPSHOT)
            count = counter.get(identifier, txn=txn)
            txn.commit()
            if count:
                count = loads(count)
                print('time:', i, 'read', identifier, count)
        sleep(1)
    print('reader finished')
    counter.close()
    env.close()
    exit(0)
```

```
def uuid():
    now = datetime.now()
    now = now.isoformat()
    data = now.encode('ascii')
    id = md5(data)
    id = id.hexdigest()
    return id.encode('ascii')


if __name__ == '__main__':
    jobs = list()
    # spawn (or fork?) two writer
    identifiers = list()
    for i in range(2):
        identifier = uuid()
        identifiers.append(identifier)
        p = Process(target=writer, args=(identifier,))
        p.start()
        jobs.append(p)

    # spawn (or fork?) one writer
    p = Process(target=reader, args=(identifiers,))
    p.start()
    jobs.append(p)

    # check for deadlocks...
    env, counter = opendb()
    while True:
        dead = 0
        for job in jobs:
            if not job.is_alive():
                dead += 1
        if dead == 3:
            break
        r = env.lock_detect(DB_LOCK_DEFAULT)
        if r != 0:
            print('deadlock')

    # print result
    txn = env.txn_begin(flags=DB_TXN_SNAPSHOT)
    for identifier in identifiers:
        count = counter.get(identifier, txn=txn)
        if count:
            count = loads(count)
```

```
            print(identifier, count)
    txn.commit()
    counter.close()
    env.close()
```

Astuce des entrées en double (ou composition de clef)

Avec le backend Btree il est possible d'avoir plusieurs valeurs pour une meme clef. A chaque db.put('super-dupper-high-mojo-key', value) une nouvelle entrée est crée dans la base de donnée avec la clef et cette valeur. Si la clef existe déjà elle est ajouté à la fin (par défaut).

Le problème c'est que lorsqu'on supprime la clef db.delete('super-dupper-high-mojo-key'), la base va supprimer toutes les entrées. C'est pas forcement ce que l'on veux. Une façon de contourner ce problème est de mettre la valeur dans la clef et utilisé une chaine vide comme valeur.

Par exemple, si on index les propriétés des documents avec leur valeur de façon à pourvoir retrouver tous les documents qui ont une valeur donnée pour un champs donnée. Il faut utiliser un curseur, le placer à l'aide de cursor.range correctement:

```
from collections import namedtuple


cursor = index.cursor()

# Keys have the following format:
#
#  (attribute_name, value, identifier) -> ''
#
# Where identifier is the identifier of a document
# with ``attribute_name`` set to ``value``.
#
KeyIndex = namedtuple('KeyIndex', ('attribute', 'value', 'identifier'))


def lookup_documents_identifiers(attribute, value):
    # The idenfier placeholder is set to the empty
    # string so that the cursor will be positioned at the first
    # key found for the combination of ``attribute``
    # and ``value`` if any, because the empty strings is the
    # smallest string value and the index is sorted.
    lookup = KeyIndex(attribute, value, '')
    next = cursor.set_range(dumps(lookup))

    while next:
```

```
key, _ = next  # value is a useless empty string
key = KeyIndex(*loads(key))
# check that key is within bound of the lookup
if (key.attribute == lookup.attribute
    and key.value == lookup.value):
    yield key.identifier
    next = cursor.next()
else:
    # it can mean that:
    #
    #   key.attribute != lookup.attribute
    #
    # which means there is no more document indexed
    # with this attribute, no need to iterate over more
    # keys
    #
    # or that:
    #
    #   key.value != lookup.value
    #
    # They are some document that have a value for
    # ``lookup.attribute`` but ``key.value`` is not
    # what we look for and will never be anymore since
    # the index is ordered.
    #
    # In both case, there is no more matching documents.
    break
```

En utilisant ce schema il est possible de mettre à jour l'index quand la valeur d'un attribut change sans impacter les autres documents ayant la meme valeur pour un attribut.

Y a d'autres chose dans bsddb évidemment. Si le sujet vous interesse je vous conseille de lire les documents fournis par Oracle.

2015-06-01 - Je fais nimp

Quand je me retourne et je regarde les projets informatiques que j'ai fait ces 5 dernières années je me dis vraiment, what the fuck. J'ai négligé mes amis, ma famille et ma copine *hum* mon ex pour autant de trucs que j'ai jetés ou jetables. C'est en général du code qui m'a pris du temps, plus d'une soirée ou un week end par mois.

Si je ne l'avais pas fait, est ce que je serais aussi à l'aise en informatique?

En tout cas je suis persuadé que j'aurai pu mieux faire. Déjà en passant moins de temps sur chaque sujet. Sans essayer de passer par la case «je vais faire un

projet complet qui va sauver le monde, me rendre empereur de la Terre et ses environs». J'en rajoute mais bon.

Le pire, ça doit être le sujet des graphdbs. Au final, il me semble que ce n'est pas possible de remplacer une base de donnée relationnel completement avec ce type de base de donnée. Pour certains problèmes ça reste interessant. C'est hyper élégant, simple, puissant mais pas suffisament je pense pour se séparer de postgres. Le pire, c'est que j'ai dérivé pour reinventer la roue. J'avais plein de raisons que je pensais raisonnable, mais au final j'ai juste fait mumuse avec des APIs.

Les bénéfices que j'en tire me semble maigres.

Je pense aussi que ce n'est pas la meilleur façon de faire progresser le logiciel libre et la cause du libre. Contribuer à des projets logiciel ou associatif existant c'est beaucoup plus interessant (et ceci est vrai aussi pour trouver un (bon) travail (même si certains s'en sorte s'en faire de libre)). C'est aussi plus difficile pas seulement parce qu'il y a déjà du code à lire ou parce qu'il faut interagir avec des gens mais aussi parce que ça change de mes habitudes.

Je pense aussi qu'on apprend plus, plus rapidement. Primo en lisant du code plus complexe et plus travaillé. Deuxio, on a des retours sur son code. Tertio rencontrer des gens et discuter ce qui est plus facile pour transmettre des connaissances que passer par du code.

À l'avenir, je reflechirai à deux fois avant de me lancer dans des projets perso pour privilegier les contributions.

J'ai encore incoming dans le pipe que je pense se demarque de mes précédents projets dans le sens où c'est un truc dont j'ai vraiment besoin et qui peux interesser d'autres personnes que des dev. Je dois encore mieux étudier les alternatives pour decider comment je vais continuer, si je continue. Je vais m'efforcer de trouver un moyen de l'integrer à projet existant. Sinon j'ai déjà une petite liste projets auxquelles j'aimerai contribuer.

2015-01-01 - Lazy will continue

Cette citation de Bob Marlisp est complètement à propos de continuation de séquences paresseuses en scheme.

Dans cette article je vais presenter deux constructions:

1. Les séquences paresseuses similaires aux iterables comme xrange ou aux générateurs simples.
2. Les coroutines, l'équivalent des generateurs améliorés.

Sequence paraisseuse

Une sequence est dites paresseuses, si elle ne calcule pas tous les elements qui la compose à l'avance. L'interet est double, d'une part on economise la memoire,

d'autre part le calcul se fait en plusieurs fois ce qui repartie la charge CPU dans le temps.

Il existe bien les streams scheme pour faire cela, seulement je veux explorer ça.

Il existe une autre approche emprunté a clojure nommé lazy-seq. Je n'est est pas besoin de la mise en cache des resultats (cela peux consommer beaucoup de memoire (Surtout quand on a pas besoin de cette memoization)).

La méthode simple est inspiré de lazy-seq, le principe est d'utiliser une recurrence et une closure qui va retarder le calcul de la prochaine valeur: Il aussi est possible d'implementer les sequences paresseuses à l'aide de routines de controle du flow des programmes comme yield en Python et call/cc en Scheme que j'essaye d'aborder dans la seconde partie.

```
(define multiples-of-three
  (let next ((n 3))
    (lambda ()
      (values n (next (+ n 3))))))
```

Ligne par ligne cela donne:

1. Definition d'une variable multiples-of-three qui va contenir la définition de la séquence.
2. La deuxieme ligne definie une lambda à l'aide de la forme let nommé qui encapsule le code de la séquence. Le let nommé est très utile pour définir des procédures recurrentes sans utiliser un autre define ou let plus lambda. Le let nommé est bien en dehors de la lambda definissant la séquence.
3. La lambda definissant le comportement de la séquence.
4. Elle retourne deux valeurs grace à values: la valeur courante n *et* à la lambda retourner par next, qui va permettre de continuer l'iteration.

La procédure multiples-of-three retourne toujours 3 et la lambda de la deuxième iteration. Après le premier appel, elle n'est plus jamais appellé. C'est la lambda qui définit la procédure qui est appellé mais avec un contexte différent.

L'utilisation du let nommé complique les choses en un sens. Voici une version qui ne l'utilise pas:

```
(define (multiples-of-three-rec n)
  (values n (lambda () (multiple-of-three-rec (+ n 3)))))


(define (multiples-of-three)
  (multiples-of-three-rec 3))
```

Voici comment cela s'utilise:

```
(use-modules (ice-9 receive))
```

```
(define multiples-of-three
  (let next ((n 3))
    (lambda ()
      (values n (next (+ n 3))))))


(receive (value next) (multiples-of-three)
  (format #t "~a\n" value)  ;; => 3
  (receive (value next) (next)
    (format #t "~a\n" value)  ;; => 6
    (receive (value next) (next)
      (format #t "~a\n" value))))  ;; => 9
```

On peux utiliser le même principe en Javascript ou Python. Dans le code suivant je présente une implementation de multiples-of-three en Javascript:

```
function multiplesOfThree() {

    function next(n) {
    // wrap next call to delay its execution.
    function wrapper () {
        return next(n + 3);
    };
    return {value: n, next:wrapper};
    }

    return next(3);
}

iter = multiplesOfThree()
console.log(iter)
iter = iter.next()
console.log(iter)
iter = iter.next()
console.log(iter)
```

Et en Python:

```
def multiple_of_three():

    def next(n):
    return [n, lambda: next(n+3)]

    return next(3);

value, next = multiple_of_three()
print(value)
```

```
value, next = next()
print(value)
value, next = next()
print(value)
```

Remarque: les deux langages ont déjà un moyen beaucoup plus simple de faire ce genre de chose à l'aide de leur yield respectif, exemple en Python:

```
def multiple_of_three():
    n = 3
    while True:
        yield n
        n += 3


generator = multiple_of_three()

print(generator.next())
print(generator.next())
print(generator.next())
```

En javascript, avec une version recente de node et le flag –harmony cela donne:

```
function* multiplesOfThree(){
    var n = 3;
    while (true) {
    yield n;
    n = n + 3;
    }
}


iterator = multiplesOfThree()

console.log(iterator.next())
console.log(iterator.next())
console.log(iterator.next())
```

Cette méthode resoud en scheme le problème de la construction de la liste paresseuse de façon plus élégante. Ceci dit, il est encore necessaire de construire des procedures map, fold, for-each, filter spécifiques.

Continuations

Au cours de mes lectures il m'a semblé que call-with-continuation (call/cc pour les intimes) était la procedure (?) qui cristalise l'identité minimaliste de scheme dans le sens où il s'agit d'une construction très puissante et simple . Elle n'exsite pas ailleurs, on lui préfére des constructions spécialisés comme yield ou goto. En

effet, call/cc peux emuler la plus part si ce n'est pas toutes les constructions de controles. Le support de call/cc se fait au prix de compilateurs/interpreteur plus compliqués.

Sans autres formes de procès voilà une procédure permettant d'implementer des coroutines:

```
(define (coroutine routine)
  (let ((current routine)
    (status 'new))
    (lambda* (#:optional value)
      (let ((continuation-and-value
         (call/cc (lambda (return)
            (let ((returner
                   (lambda (value)
                (call/cc (lambda (next)
                       (return (cons next value)))))))
              (if (equal? status 'new)
                  (begin
                (set! status 'running)
                (current returner))
                  (current (cons value returner)))
              (set! status 'dead))))))
    (if (pair? continuation-and-value)
        (begin (set! current (car continuation-and-value))
           (cdr continuation-and-value))
        continuation-and-value)))))
```

Dans les grandes lignes, un appel call/cc va créer un "label" dynamiquement referencé par la variable passé à la callback de call/cc. La callback est appellé immediatement. Si l'envie lui prend de "quitter/revenir" mais plus precisement the continuer sa vie avec la continuation elle va l'appeller (avec un argument si sa lui chante). Ce comportement de base est illustré dans le code suivant:

```
(define why (call/cc (lambda (return)
               (format #t "love me or leave me!")
               (return "I leave!")
               ;; the program never reach this part
               (format #t "it probably left :("))))
(format #t "return actually populates WHY variable\n")
(format #t "WHY: ~a\n" why)
```

Avec cette exemple, on dirait que c'est rien de plus qu'un return. En fait c'est bien plus que ça. La continuation est une variable et pas un keyword, elle peux etre gardé en mémoire, passé à une autre procedure. Elle est dynamique contrairement à goto, qui attend un label qui peut-être resolue par le compilateur.

Mon implementation est loin d'être aussi facile à utiliser que le yield Python. En effet chaque yield crée une nouvelle continuation et donc un nouveau yield

cf. second-yield:

```
(define example-coroutine
  (coroutine (lambda (yield)
           (display "coroutine says: HELLO!")
           (newline)
           (let ((second-yield (cdr (yield 1))))
         (display "coroutine says: WORLD!")
         (second-yield 2)
         (newline)
         (display "coroutine says: SORRY, I'M OUT")))))


(display (example-coroutine))
(newline)
(display (example-coroutine))
(newline)
```

Clairement cela necessite un peu plus de travail. J'ai tourné en rond un certain temps pour résoudre le problème de la creation des yield pour avoir une syntax moins imbriqué et plus lineaire.

Les *delimited continuations* et la procédure *amb* sont deux formes de controles qui peuvent être implementé à l'aide de *call/cc*.

2015-01-01 - Les bases du langage Python 3

Les bases qui permettent de se servir de Python comme d'une calculette pour faire les courses et plus si affinités.

Cette note est basée sur learn X in Y minutes sous license Creative Commons by-sa. La license est maintenue à l'identique.

```
#!/usr/bin/env python3
# Une ligne de commentaire commence par un croisillon

# La PEP8 recommende de laisser un espace après le croisillion
# cf. http://legacy.python.org/dev/peps/pep-0008/

# La PEP8 recommende de ne pas écrire de lignes de plus de 80
# caractères. Si le commentaire ne tiens pas en une ligne
# il est possible d'enchainer les lignes de commentaires.


# 1. Types de bases et leurs opérateurs

# Il existe deux types de nombres:
# - les nombres décimaux dit flottants
```

```
# - les nombres entiers

# Les opérations mathématiques ressemblent à des opérations mathématiques
1 + 1  # => 2
8 - 1  # => 7
10 * 2  # => 20

# Les commentaires peuvent aussi se trouvrer à la fin d'une ligne.
# La PEP8 recommende de laisser deux espaces à la fin du code
# avant le croisillon

# Par défaut la division retourne un nombre flottant
35 / 5  # => 7.0

# Il existe un opérateur de division entière (ie. euclidienne)
5 // 3  # => 1
-5 // 3  # => -2

# Si un flottant apparait dans une opération arithmétique
# le resultat sera un flottant
3 * 2.0  # => 6.0

# La division entière de deux flottants va retourner
# la "version" flottant du resultat de la division entière
5.0 // 3.0  # => 1.0 au lieu de 1
-5.0 // 3.0  # => -2.0 au lieu de 2

# Pour réalisé un modulo il faut utiliser le caractère ``%``
7 % 3  # => 1

# Il est possible d'utiliser des parenthèses pour grouper
# les opérations arithmétiques
(1 + 3) * 2  # => 8

# Il existe un type booléen
True
False

# Il est possible de nier une valeur booléene à l'aide de
# l'opérateur ``not``
not True  # => False
not False  # => True

# L'opération d'égalité se fait à l'aide d'un double signe égale
1 == 1  # => True
2 == 1  # => False
```

```
# Il existe aussi un opérateur d'inégalité
1 != 1  # => False
2 != 1  # => True

# Ainsi que les opérateurs classiques de comparaisons
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# Il est possible d'enchainer les comparaisons
1 < 2 < 3  # => True
2 < 3 < 2  # => False

# Il y a trois façons de rentrer des chaines de caractères
string = 'This is also a string.'
# Dans la chaine suivante le caractère ``'`` apparait
string = "Lil' did you know."
# Dans la chaine suivante les caractères ``'`` et ``"``
# apparaisent dans la chaine. Aussi la chaine peut s'étaler
# sur plusieurs lignes.
string = """This isn't a "mishmash

this pure Python!"""

# Il est possible d'acceder aux élèments de la chaine à l'aide
# de l'opérateur ``spam[]``
"This is a string"[0]  # => 'T'

# Il est possible d'ajouter des chaines de caractères
chaine = "Hello " + "world!"  # => "Hello world!"

# Cela dit ce n'est pas la méthode recommendé, il faut lui preferer
# l'interpolation
"%s can be %s" % ('string', 'interpolated')


# ``None`` est un objet unique, il correspond à ``null`` dans
# d'autres langages
None  # => None

# Il ne faut pas utiliser l'opérateur d'égalité ``==`` pour comparer
# un objet à ``None``. ``is`` est l'opérateur qu'y conviens.
"etc" is None  # => False
None is None  # => True
```

```
# 2. Variables et Collections

# Pour créer une variable, il suffit de lui assigner une valeur
some_var = 5
some_var  # => 5

# La PEP8 recommende d'utiliser des lettres_minuscules_separe_par_des_espaces
# pour nommer les variables

# Il est possible d'assigner une nouvelle valeur à une variable sans
# autre forme de procès
some_var = 3.14
some_var = True

# Acceder à une variable qui n'a pas été définit précédemment va
# lever une erreur dites "NameError"
some_unknown_var
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# NameError: name 'some_unknown_var' is not defined


# Créer une liste ordonnée vide
some_list = list()
# Créer une liste ordonnée avec des élèments
some_other_list = [3, 4, 5, 6]

# Pour acceder à un élèment de la liste, la meme syntaxe vue
# por les chaines peut-etre utilisé
some_other_list[0]  # => 3

# En cas de dépassement, une erreur dite "IndexError" est levé
some_other_list[4]
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# IndexError: list index out of range


# Il est possible de parcourir la liste en sens
# inverse en utilisant des index négatifs
some_other_list[-1]  # => 6
some_other_list[-2]  # => 5

# Il est possible de retirer des élèments de la liste à l'aide du mot-clef ``del``
```

```
del some_other_list[2]  # [3, 4, 6]


# Il est possible d'ajouter une liste à une autre liste
a = [1, 2, 3]
b = [4, 5, 6]
ab = a + b  # [1, 2, 3, 4, 5, 6]

# Il est possible d'ajouter un element à une liste de la façon suivante:
yet_another_list = ab + [7]  # [1, 2, 3, 4, 5, 6, 7]

# Pour savoir si un élèment est inclus dans une liste
# il faut utiliser le mot-clef ``in``
1 in yet_another_list   # => True


# Il est possible assigner les élèments d'une liste
# à des variables à l'aide de la syntaxe suivante
a, b, c = [1, 2, 3]     # a == 1, b == 2 et c == 3


# Un dictionnaire est un ensemble d'association clef -> valeur.
# Dans un certains sens, il est similaire à une liste, sauf que le plus souvent
# pour acceder aux valeur il faut passer par des chaines.
#
# Il est possible de créer un dictionnaire vide à l'aide de la syntaxe suivante
empty_dict = dict()
# Un dictionnaire pre-remplis peut-etre créer de la façon suivante:
filled_dict = {"one": 1, "two": 2, "three": 3}

# Pour acceder aux valeur utiliser la syntaxe ``some_dict[clef]``
filled_dict["one"]   # => 1
# un code équivalent:
key = 'one'
filled_dict[key]

# Pour verifier l'existence d'une clef il faut utiliser
# le mot-clef ``in``
"one" in filled_dict  # => True
1 in filled_dict  # => False

# Acceder à une clef qui n'existe pas va lever une erreur
# dite "KeyError"
filled_dict["four"]
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
```

```
# KeyError: 'four'

# Il est possible de supprimer une entrée dans le dictionnaire
# à l'aide de la syntaxe suivante
del filled_dict["one"]


# Un ensemble peux-etre créer à l'aide du code suivant:
empty_set = set()
# Pour créer un ensemble pré-remplit, le code suivant est suffisant:
some_set = {1, 1, 2, 2, 3, 4}
# un ensemble contiens chaque valeur une fois
some_set == {1, 2, 3, 4}  # => True

# Pour connaitre l'intersection entre deux ensembles,
# l'opérateur esperluete ``&`` peut etre utiliser
other_set = {3, 4, 5, 6}
some_set & other_set   # => {3, 4}

# Pour réaliser l'opération d'union il faut utiliser
# L'opérateur trait vertical ``|``
some_set | other_set   # => {1, 2, 3, 4, 5, 6}

# Pour réaliser l'opération de différence, il faut utiliser
# l'opérateur moins ``-``
{1, 2, 3, 4} - {2, 3, 5, 7}   # => {1, 4, 7}

# Le mot-clef ``in`` permet de savoir si un élèment est dans
# un ensemble
2 in some_set   # => True
10 in some_set   # => False

# Il est aussi possible de verifier l'existence d'un objet
# dans une liste a l'aide du meme operateur ``in``
2 in [1, 2, 3]
2 in [4, 5, 6]

# Enfin...
# None, 0, les chaines vides, les listes vides et les dictionnaires vides
# et les ensembles vides sont considérés comme ayant une valeur ``False``
bool(0)  # => False
bool("")  # => False
bool([])  # => False
bool({})  # => False

# Toutes les autres valeurs sont considérées comme équivalentes à ``True``
```

Il y a plein de ressources en français pour apprendre le Python. Voici une liste qui peux aider:

- Débuter avec Python au lycée
- Introduction à Python 3 sans oublier le mémento qui va avec

Maintenant quelques exercices:

- Etant donnée un livre qui a 250 pages, 60 lignes par page et 15 mots par ligne en moyenne. Combien de mot y a t-il a peu pres dans le livre?
- Etant donnee la phrase “Quelle belle arbre” ainsi que “Quelle belle fleur”, calculer le nombre de mots qui apparaissent dans les deux phrases.
- On considère deux dictionnaires, le premier est former par l’association des chiffres de 0 a 9 avec leur écriture en francais, le second est former par les chiffres de 0 a 9 en francais associés aux nombres de 0 a 9 en anglais. Mettez en place une suite d’operation qui permet de retrouver l’ecriture d’un nombre en anglais a partir d’un chiffre.

Good luck!

2015-01-01 - Python: subscript rencontre un générateur

Python a un certains nombre de méthodes dites «magiques», notamment **getitem**. C’est la méthode qui est appellé lorsque on fait quelque chose comme make[some_fun], c’est l’opérateur «subscript».

Voyons voir:

```
class SubScript:

    def __getitem__(self, value):
        return 'value est %s' % str(value)


subscript = SubScript()
print(subscript[42])  # value est 42

# mais aussi
print(subscript[42:52])  # value est slice(42, 52, None)

# et encore
print(subscript[42:52:102])  # value est slice(42, 52, 102)
```

Vous allez me dire mais à quoi ça sert? Et bien c’est simple: implémenter une classe qui ressemble à une liste. Comme range en Python 3, qui est générateur qui accepte de se faire découper [to slice].

Le code suivant montre comment crée ce type de générateur:

```
class SubscriptableGenerator:

    def __init__(self, n=None, slice=None):
        self.n = n
        self.slice = slice

    def __getitem__(self, i):
        if isinstance(i, slice):
            return GeneratorSubscriptable(None, i)
        else:
            return self._compute_fibonacci(i)

    def __iter__(self):
        if self.slice:
            if self.slice.step:
                step = self.slice.step
            else:
                step = 1
            n = step
            while n <= self.slice.stop:
                yield self._compute_fibonacci(n)
                n += step
        else:
            n = 0
            while n <= self.n:
                yield self._compute_fibonacci(n)
                n += 1

    def _compute_fibonacci(self, n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            a = self._compute_fibonacci(n-1)
            b = self._compute_fibonacci(n-2)
            return a + b

fibo15 = SubscriptableGenerator(15)

print(list(fibo15))

print(fibo15[5])

print(list(fibo15[0:10]))
```

```
print(list(fibo15[0:10:2]))
```

Cette construction me laisse bouche béh (!) mais je sais pas trop à quoi cela pourrait servir.

2016-01-01 - Code Crafting Codex

At the very core of programming, there is an obvious need for *a divide and conquer strategy*. And that is at the personal level of reflexion. Of course it happens to be true in other domains.

This idea is also expressed as *Separation of Concerns* which wikipedia nails its value in the following wishful paragraph without mentioning its evil extrema:

> The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well separated, individual sections can be developed and updated independently. Of special value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections.

Otherwise said two components satisfy "separation of concerns" if their implementation details don't leak.

Drawing lines is not easy, especially when you consider the infinite entangledments of human understandings.

I think that **Zen of Python** by *Tim Peters* provides useful abstract guiding principles:

> Beautiful is better than ugly.
>
> Explicit is better than implicit.
>
> Simple is better than complex.
>
> Complex is better than complicated.
>
> Flat is better than nested.
>
> Sparse is better than dense.
>
> Readability counts.
>
> Special cases aren't special enough to break the rules.
>
> Although practicality beats purity.
>
> Errors should never pass silently.
>
> Unless explicitly silenced.
>
> In the face of ambiguity, refuse the temptation to guess.

> There should be one – and preferably only one – obvious way to do it.
>
> Although that way may not be obvious at first unless you're Dutch.
>
> Now is better than never.
>
> Although never is often better than *right* now.
>
> If the implementation is hard to explain, it's a bad idea.
>
> If the implementation is easy to explain, it may be a good idea.
>
> Namespaces are one honking great idea – let's do more of those!

Those quotes are also helpful:

> Simple things should be simple, complex things should be possible

Alan Kay

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupery

> To design something really well, you have to get it. You have to really grok what it's all about. It takes a passionate commitment to really thoroughly understand something, chew it up, not just quickly swallow it. Most people don't take the time to do that.

Steve Jobs

> Programs are meant to be read by humans and only incidentally for computers to execute

Donald Knuth

The following quote must be studied in the light of broad sens that can take *the word before the last*:

> There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

And this one provides more food for thought:

> The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free? The Web, in comparison, is a joke. The Web was done by amateurs.

Alan Kay

If sounds, grounds, ideas and understandings are echoing, don't overestimate their infinite reflection that might be aiming an unforeseen dubious equilibrium, find you way out, escape the invisible tie knot, think on your own.

2016-01-01 - Do It Yourself: a graph database in Python

You maybe already know that I am crazy about graph databases. I am trying to build a graph database in Python. Which graphdb? Well this is a moving target. The following design is a good for small databases that don't include a lot of big fields.

One must know that there is much much less code in this graphdb implementation than in the previous iteration on ajgudb. I think it a good fit for graph exploration and analysis but anything that is bigger than a personnal website will not work, for sure. It doesn't provide transactions anyway.

It's main advantage is that it's simple and that everything is indexed.

It is inspired from datomic Entity-Attribute-Value pattern. It based on a schema I call the TupleSpace.

What is the Tuple Space

The idea behind tuple space is to store a set of tuples inside a single table that look like the following:

```
(1, "title", "Building a python graphdb in one night")
(1, "body", "You maybe already know that I am...")
(1, "publishedat", "2015-08-23")

(2, "name", "database")

(3, "start", 1)
(3, "end", 2)

...

(4, "title", "key/value store key composition")

...

(42, "title", "building a graphdb with HappyBase")
```

The tuple can be described as (identifier, key, value).

If you study the above example you will discover that both edges (aka. ManyToMany relations) are represented using the same tuple schema.

Representing ForeignKey is possible but is left as an exercice to the reader ;)

Implementation of a Tuple Space

The first section describe the API of the TupleSpace, following section describe how the TupleSpace is implemented on the storage.

API

The API provided by the tuple space is document oriented instead of tuple oriented. It looks like the following:

```
class TupleSpace:

    def get(self, uid):
        """Return all tuples with the given uid"""
        pass

    def add(self, uid, **properties):
        """Add the tuples formed with key/value pairs
           taken from `properties` and `uid`"""
       pass

    def delete(self, uid):
        """Delete tuples with `uid`"""
        pass

    def query(self, key, value=''):
        """Iterate the index for tuples having `key` and
           optionaly `value`"""
        pass
```

Schema

To implement the above schema inside an ordered key/value store we have to find a relevant key. That key I think, is the composition of the identifier and name. This leads to definition of the following table Key:

`Key(identifier, name) -> Value(value)`

Every Key is unique and is associated with a Value. Given the fact that the store is ordered one can easily retrieve every (key, value) tuple associated with a given identifier by going through the the ordered key space.

The above tuple space will look like the following in the key/value database; using a high level view ie. not bytes view:

```
*******|           Key            |                 Value
-------+------------+-------------+-----------------------------------------
Colunms| identifier |  name       |                 value
-------+------------+-------------+-----------------------------------------
```

```
|     1      |   title     | "Building a python graphdb in one night"
|     1      |   body      | "You maybe already know that I am..."
|     1      | publishedat |              "2015-08-23"
|     2      |   name      |                database

|     3      |   start     |                  1
|     3      |    end      |                  2

     ...          ...                       ...

|     4      |   title     |     "key/value store key composition"

     ...          ...                       ...

|     42     |   title     |    "building a graphdb with HappyBase"

     ...          ...                       ...
```

Key composition

To keep the database ordered you need to pack correctly the components of the Key. You can not simply convert string to bytes, how will you distinguish the string from the other components of the key? You can't use the string representation of number ie. "42" for "42". Remember "2" is bigger than "10".

In a complete TupleSpace implementation one must also take into acccount that values can be of many types.

The simple case of positive integers is solved by struct.pack('>Q', number).

The solution to support all numbers is to always use the same packing schema whatever the sign and whether they are float or not.

Here is a naive packing function that support every Python objects, keeps the ordering of strings and positive integers where integers comes before strings which come before other kind of Python values:

```python
def pack(*values):
    def __pack(value):
        if type(value) is int:
            return '1' + struct.pack('>q', value)
        elif type(value) is unicode:
            return '2' + value.encode('utf-8') + '\0'
        elif type(value) is str:
            return '3' + value + '\0'
        else:
            data = dumps(value, encoding='utf-8')
            return '4' + struct.pack('>q', len(data)) + data
```

```
    return ''.join(map(__pack, values))
```

In database that is column aware it's not always required to build such packing function as the database already has way to compose key as columns.

GraphDB

At this point, the TupleSpace provides documents and some relational paradigm as you can work with references. AjguDB provides a layer on top of TupleSpace to easily work with a graph database.

Data model

The first aspect is building the graph data model:

Vertex are simple TupleSpace documents which identifieres come from document 0, a counter which is incremented everytime a new vertex or edge is created. Moreover it stores in _meta_type key that the document represent a Vertex.

Edge are also simple TupleSpace documents with their identifier come from the counter document with 0. Same as Vertex, Edge document store as _meta_type the fact that they are edge. Moreover start and end attributes are also stores in the TupleSpace document.

Given the fact that every tuples are indexed, it's easy to retrieve all incomings and outgoings edges of a given Vertex so it's not required to cache them in the Vertex document (as it is done in ajgu).

Better schema

A better schema will use one row per document and use the same row to store all edge information. That said tuple spaces is an existing pattern in use in distributed databases.

DIY.

2016-01-01 - Feedback client sur la création de la Terre

Bonjour Dieu,

Merci beaucoup pour tes efforts. Il y a vraiment quelque chose qui se passe. J'ai quelques points à remonter:

1 – J'aime beaucoup toute cette idée sur la lumière, mais j'ai un doute quant à la nomenclature. «Jour» et «nuit», ça va, mais j'ai le sentiment qu'on pourrait améliorer ça. Des idées ? Il faut vraiment qu'on trouve quelque chose dès que possible.

2 – Re: Le «ciel»… Ça manque de mojo. Il faudrais un truc plus «pop». Vous auriez pas d'autres idées?

3 – J'apprecie le travaille qui a été réalisé au niveau de la terre et des océans, mais en ce moment il y a beaucoup trop d'océan. Il faut s'y mettre et chercher pour trouver de la terre. De façon général, l'océan ne résonne pas vraiment bien avec nos utilisateurs. En discutant de ce sujet, l'équipe est venu avec une idée: pourquoi ne pas supprimer les océans. Des idées?

4 – On a remarqué que vous aviez recouvert la terre de végétation qui se reproduisent uniquement entre eux. De meme pour les arbres fruitiers. C'est fait exprès? S'il vous plaît donner votre avis.

5 – Actuellement, nous avons deux grandes sources de lumière dans le ciel… une grosse le jour et une plus petite la nuit? Nous craignons de ne pas avoir été clair lors de l'exposé initiale. Nous avons réellement besoin de plus que ça. Il faut que se soit mémorable, une experience à grande valeur ajoutée pour nos utilisateurs. Veuillez vous référer aux slides page 13 et 14. Donnez de la voix si nécessaire.

6 – Les océans remplit de vie ça passe, mais encore une fois, il faut reduire sa superficie. C'est un no-go pour nous.

7 – Est ce que c'est la dernière version des oiseaux? On a un mauvais feeling avec eux. On souhaiterais des clarifications avant de nous exprimer plus à ce sujet.

8 – Est ce qu'on pourrait avoir plus de bétail et d'animaux sauvages qui se baladent ensemble par éspèce? Encore une fois, chez notre utilisateur finale (slide 18) la terre et les animaux qui trainent dessus c'est hyper important. De fait, tout ce qui va permettre d'augmenter la quantité de terre permettra de transformer les utilisateurs passif en évangéliste.

9 – Re: «L'humanité.» Point interessant suite au brieffing. Ce qui est douleureux, c'est que l'humanité soit largement inspiré par vous. Heureusement, il y a plusieurs catégories d'utilisateurs (slide 27) et on veux vraiment qu'il se sente représenté (slide 28). Avoir des mamiphères bipèdes qui «dominent» la terre et les océan (si il y a toujours des océans) risque de démontiver nos utilisateurs, ce qui les frainera dans leur conversation en évangeliste. Pour faire court, une version alternative de l'humanité est à proscrire. C'est possible?

10 – Oubliez complétement «Soyez féconds et multipliez vous». Celà ne résonne pas du tout avec notre l'engagement de marque comme une marque familliale (slide 34).

Je comprend que c'est samedi et vous avez peut-etre envisagé de vous mettre une mine demain pour admirer votre création et tout, mais j'espère que vous pourrez reprendre ça demain. Je dois me presenter lundi à la première heure devant l'équipe dirigeante, il serait donc bien qu'on puisse nous aussi faire un point demain à la première heure. Je serait pas loin de mon téléphone toute le week-end pour discuter. Vous et votre équipe approchez une grande victoire.

Merci!

Cette article est une traduction de Client Feedback On the Creation of the Earth

2016-01-01 - Getting started with Guile Parser Combinators

Guile Parser Combinators implement (monadic) parser combinators. What it means in pratice is that you create procedures that parse most of the time strings and output a structured data, most likely s-expr. They are said to be *combinators* because you can compose parser procedures. The monadic part is an implementation detail mostly.

Say you want to parse some input text something like markdown and turn it into sxml.

Be aware that guile-parser-combinators (unlike guile-log) doesn't have an error handling machinery but since you compose small parser to make a bigger parser it's easy to debug the small units and have a working parser.

This blog use a markdown parser implemented using guile-parser-combinator; it ain't perfect, they are bugs with workarounds.

The first part of this article introduce guile-parser-combinators concepts and then apply them to build a csv parser.

Getting started

Download guile-parser-combinator:

```
$ git clone git://dthompson.us/guile-parser-combinators
```

Fire an REPL inside the created directory:

```
$ cd guile-parser-combinators
guile-parser-combinators $ guile -L `pwd`
scheme@(guile-user)> (use-modules (parser-combinators))
```

We will heavily rely on streams aka. srfi 41, don't forget to import it:

```
scheme@(guile-user)> (use-modules (srfi srfi-41))
```

You are ready!

Three kinds of parsers

All parsers can return two kind of values of `<parse-result>` a `parse-success` or a `parse-failure`. `parse-success` have a `value` which is the output of the parser and a `stream` which is what remains to be parsed. `parse-failure` results have no value and no stream.

There is three kind of parsers in guile-parser-combinators. In the following I described them in the case where the input is strings and the final output is an s-expr:

- plain parser: `string -> string`. This allows to recognize and split input string into small units. This is the most basic parser. Example plain parser is `(parse-char #\c)` which will succeed if the input stream starts with a `c` char, otherwise it fails. It will return `c` as value and return the input stream queue.
- combinator parser: `parser -> parser`. Those takes as input a parser and output another parser. This might seem strange. I think it's better to call them *control* parsers but the literature says otheriwse. Example combinator parser is `(parse-any (parse-char #\a) (parse-char #\b) (parse-char #\c))` which will succeed if any of its input parser succeed. Which means in this case that the input stream must start with `a` or `b` or `c` char.
- output builder: `s-expr -> s-expr`. Those are not really parser as they never consume the input stream but instead shake around `<parse-result>` value.

All those parsers are further explained in what follows.

Plain parsers

I lied a little previously, I said that plain parser take as input string and output string. Actually it's just a (second hand) view to understand how parser works. The actual implementation takes as input a stream and outputs a `parse-result`.

For instance the following is a valid parser, that will return a `parse-result` that succeed when the first char in the stream is a `c`:

```
(define (parse-c-char stream)
    (if (eqv? (stream-car stream) #\c)
        (parse-result #\c (stream-cdr stream))))
```

You can test it with the following:

```
scheme@(guile-user)> (list->stream '(#\c #\c #\c))
$3 = #<stream ...>
scheme@(guile-user)> (parse-c-char $3)
$4 = #<<parse-result> value: #\c stream: #<stream ...>>
```

The only problem of this parser is that it doesn't handle the case where the input stream is empty. Nonetheless start to get the idea.

To parse a `c` char using guile-parser-combinator, you have to use the `(parse-char char)` procedure which returns a parser that succeed if the first item of the input stream is a `c` and fails otherwise:

```
scheme@(guile-user)> ((parse-char #\c) (list->stream '(#\c #\c #\c)))
$8 = #<<parse-result> value: #\c stream: #<stream ...>>
```

```
scheme@(guile-user)> ((parse-char #\c) (list->stream '(#\a #\b #\c)))
$9 = #<<parse-result> value: #f stream: #f>
```

As you can see in result `$8`, `parse-result`'s value is the `c` char. You can't check easily that `(parse-char #\c)` consumed the input stream, you'll have to believe me for now.

Here's the implementation of `parse-char`, as you can see, it return a `stream -> parse-result` procedure:

```
(define (parse-char char)
  "Create a parser that succeeds when the next character in the stream
   is CHAR."
  (lambda (stream)
    (stream-match stream
      (() %parse-failure)
      ((head . tail)
       (if (equal? head c)
           (parse-result head tail)
           %parse-failure)))))
```

There is various procedure that return plain parsers:

- `(parse-char char)`
- `(parse-char-set char-set)` which parse a char from the provided charset.
- `(parse-string string)` which parse iteratively the list of chars that makes up `string` from the input stream.

Let's try the last procedure:

```
scheme@(guile-user)> (define string->stream (compose list->stream string->list))
scheme@(guile-user)> ((parse-string "ccc") (string->stream "ccc means chaos computer club"))
$12 = #<<parse-result> value: "ccc" stream: #<stream ...>>
```

There is also `parse-any-char` which is not a procedure that returns a parser, but a plain parser. It will consume any char found in the input stream:

```
scheme@(guile-user)> (parse-any-char (string->stream "scheme is awesome"))
$13 = #<<parse-result> value: #\s stream: #<stream ...>>
scheme@(guile-user)> (parse-any-char (string->stream "\nthis starts with a newline"))
$15 = #<<parse-result> value: #\newline stream: #<stream ...>>
```

At this point, if things are still fuzzy, it's ok because you still don't know how given only `(parse-string string)`, `(parse-char char)` and `parse-any-char` plain parsers how you can parse something. Things will get more interesting once you know about *control parser*!

Combinator parsers

Combinator (control?) parsers are procedures similar in principle to `compose` except they take as input other parser and are tailored to compose them using semantic useful in the context of parsing.

Here is the full list of control parsers:

- `(parse-any parser ...)` try to parse input stream with each parser given as argument and succeed as soon as one of the parser succeed. Fails if no parser succeed. (It also called `parse-or`)
- `(parse-each parser ...)` parse input with the parser provided as argument feeding each parser with the stream that previous parser returned. Succeed only if all parser succeed. This allows to parse a sequence of items from the stream otherwise to walk forward in the stream.
- `(parse-zero-or-more parser)` and `(parse-one-or-more parser)` will apply the parser passed argument zero or one or more times to the input stream feeding the result stream of the first iteration to the same parser until `parser` can't parse anything from the input stream. `parse-zero-or-more` succeed all the time but will return the same stream as the input stream if it can't parse even once the intput stream. `parse-one-or-more` must at least succeed once to parse the input stream. This also allows to walk forward the stream.

Let's try `parse-each`:

```
scheme@(guile-user)> ((parse-each parse-any-char parse-any-char parse-any-char) (string->str
$17 = #<<parse-result> value: (#\g #\n #\u) stream: #<stream ...>>
scheme@(guile-user)> (define parse-gnu (parse-each (parse-char #\g) (parse-char #\n) (parse-
scheme@(guile-user)> (parse-gnu (string->stream "gnu"))
$18 = #<<parse-result> value: (#\g #\n #\u) stream: #<stream ...>>
scheme@(guile-user)> (parse-gnu (string->stream "ccc"))
$19 = #<<parse-result> value: #f stream: #f>
```

Let's try `parse-any`:

```
scheme@(guile-user)> (define parse-a-or-b (parse-any (parse-char #\a) (parse-char #\b)))
scheme@(guile-user)> (parse-a-or-b (string->stream "a"))
$20 = #<<parse-result> value: #\a stream: #<stream ...>>
scheme@(guile-user)> (parse-a-or-b (string->stream "b"))
$21 = #<<parse-result> value: #\b stream: #<stream ...>>
scheme@(guile-user)> (parse-a-or-b (string->stream "c"))
$22 = #<<parse-result> value: #f stream: #f>
```

I guess now you get the point.

Output builder

Output builder parsers are special parser that takes a parser as argument and process its output before returning. You can use `(parse-map proc parser)` or

`(parse-match parser matcher ...)` to do so.

For instance, here is an example use of `parse-map`:

```
scheme@(guile-user)> (define parse-gnu* (parse-map (lambda (lst) `(b ,(list->string lst))) p
scheme@(guile-user)> (parse-gnu* (string->stream "gnu is awesome"))
$25 = #<<parse-result> value: (b "gnu") stream: #<stream ...>>
```

As you can see `parse-gnu*` returns `(b "gnu")`.

wrapping with a csv parser

csv is a notoriously difficult text file format to parse because it comes in much different flavor. In what follows we will describe a parser for somekind of csv format that is straightforward.

For our csv parser we want the following two unit tests to pass:

```
(define-syntax test-check
  (syntax-rules ()
    ((_ title tested-expression expected-result)
     (begin
       (format #t "* Checking ~s\n" title)
       (let* ((expected expected-result)
              (produced tested-expression))
         (if (not (equal? expected produced))
             (begin (format #t "Expected: ~s\n" expected)
                    (format #t "Computed: ~s\n" produced))))))))


(when (or (getenv "CHECK") (getenv "CHECK_MARKDOWN"))
  (test-check "single line"
              (csv "a;b;c;")
              (list (list "a" "b" "c")))

  (test-check "multi line"
              (csv "a;b;c;
d;e;f;")
              (list (list "a" "b" "c") (list "d" "e" "f"))))
```

Otherwise said, a csv will be a multiline string with columns separated by semicolon chars.

parse-unless

To be able to parse such a format, we need to introduce another parser combinator called `(parse-unless predicate parser)`. What it does is that it only execute `parser` if `predicate` parser fails. This is useful because a lot of time they are control characters in the input stream that identifies the start or end

of a given text unit. In the case of the csv parser there is two control chars: the semi-colon and newlines.

The implementation of `parse-unless` is simple! Here is it:

```
(define (parse-unless predicate parser)
  (lambda (stream)
    (match (predicate stream)
      ((? parse-failure?) (parser stream))
      (_ %parse-failure))))
```

Basically what it does, is that it checks that `predicate` parser fails on input stream and in that case execute `parser`.

(Otherwise the big picture is that the caller, rewinds the stream until it can find a branch using `parse-any` that succeeds… otherwise said, parser combinators try every parser until it find a parser that succeed or it fails).

Anyway, this allows to check for the presence of control chars before parsing something. There is an similar `(parse-when predicate parser)` and you might think that it's equally useful but in practice when parsing `parse-unless` is much more useful. It has to do with the fact, that negation allows to capture much more logic that plain equality. Negation is very powerful.

Parsing a column

In csv a column looks like `abc;` where the semi-colon `;` marks the end of the column. So a column is made of its value in this case `abc` and its end marker the semi-colon. A such, a column value can be anything but a semi-colon. This is a first parser:

```
scheme@(guile-user)> (define parse-column-value-char (parse-unless (parse-char #\;) parse-an
scheme@(guile-user)> (parse-column-value-char (string->stream "abc;"))
$1 = #<<parse-result> value: #\a stream: #<stream ...>>
scheme@(guile-user)> (parse-column-value-char (string->stream ";abc"))
$2 = #<<parse-result> value: #f stream: #f>
```

But this parser only parse a single char, let's repeat the same parser several times, one or more times using `parse-one-or-more` (we consider that everycolumn has at least one char value).

```
scheme@(guile-user)> ((parse-one-or-more parse-column-value-char) (string->stream "abc;"))
$3 = #<<parse-result> value: (#\a #\b #\c) stream: #<stream #\; ...>>
```

And there we have parsed the first column value "abc" but it's not properly packed as a string, to fix that we can use `(parse-map proc parser)` which will lift the `parse-result`'s value with `list->string`:

```
scheme@(guile-user)> ((parse-map list->string (parse-one-or-more parse-column-value-char)) (
$4 = #<<parse-result> value: "abc" stream: #<stream #\; ...>>
```

We can make of this the first parser unit of our csv parser, as it really parse a single column value and properly packs it:

```
scheme@(guile-user)> (define parse-column-value (parse-map list->string (parse-one-or-more p
scheme@(guile-user)> (parse-column-value (string->stream "abc;"))
$5 = #<<parse-result> value: "abc" stream: #<stream #\; ...>>
```

As you can see, there's still elements in the stream. The semi-colon control character is not parsed yet, but it's really part of the column. So what will do next is parse that control char and remove it from `parse-result` using `parse-map`. Try to guess how before reading what's follows.

So we need to chain our `parse-column-value` with a `parse-column-control-char`. But before that `parse-column-control-char` is a simple `(parse-char #\;)`, we define it to make the parser more readable and redefine `parse-column-value` it terms of it because that's actually the real semantic of parsing a column value:

```
scheme@(guile-user)> (define parse-column-control-char (parse-char #\;))
scheme@(guile-user)> (define parse-column-value-char (parse-unless parse-column-control-char
scheme@(guile-user)> (define parse-column-value (parse-map list->string (parse-one-or-more p
```

Check that `parse-column-value` still works as expected.

Like I said earlier we need to chain `parse-column-value` with `parse-column-control-char` to build `parse-column`. For that we can use `(parse-each parser ...)`:

```
scheme@(guile-user)> ((parse-each parse-column-value parse-column-control-char) (string->str
$7 = #<<parse-result> value: ("abc" #\;) stream: #<stream ...>>
```

As you can see, now we have a list with two values in `parse-result`. We don't care about the semi-colon in the output. It's only present in the input to mark the end of column. So we `parse-map` away:

```
scheme@(guile-user)> ((parse-map car (parse-each parse-column-value parse-column-control-cha
$9 = #<<parse-result> value: "abc" stream: #<stream ...>>
```

That's is it, we have the definition of `parse-column`:

```
scheme@(guile-user)> (define parse-column (parse-map car (parse-each parse-column-value pars
scheme@(guile-user)> (parse-column (string->stream "abc;def;gnu;"))
$10 = #<<parse-result> value: "abc" stream: #<stream ...>>
```

As you might have guess we can repeat one or more time `parse-column` to parse a single row:

```
scheme@(guile-user)> ((parse-one-or-more parse-column) (string->stream "abc;def;gnu;"))
$12 = #<<parse-result> value: ("abc" "def" "gnu") stream: #<stream>>
```

Neat isn't it?

But this doesn't really every kind of row. Because there is two kind of rows:

- The last row that ends with an `eof`

- The other rows that ends with a newline

Let's define a `parse-eol`:

```
scheme@(guile-user)> (define parse-eol (parse-any parse-end (parse-char #\newline)))
```

Now we can (almost) define `parse-row`:

```
scheme@(guile-user)> (define parse-row (parse-each (parse-one-or-more parse-column) parse-ec
scheme@(guile-user)> (parse-row (string->stream "abc;def;gnu;"))
$15 = #<<parse-result> value: (("abc" "def" "gnu") #t) stream: #<stream>>
```

As you can see, there is some #t garbage in the `parse-result`'s value. let's get rid of it using `parse-map`:

```
scheme@(guile-user)> (define parse-row (parse-map car (parse-each (parse-one-or-more parse-c
scheme@(guile-user)> (parse-row (string->stream "abc;def;gnu;"))
$16 = #<<parse-result> value: ("abc" "def" "gnu") stream: #<stream>>
```

Wee!!! We have complete row! Guess what parser combinator you have to use to parse several rows…

Time is up:

```
scheme@(guile-user)> (define parse-csv (parse-one-or-more parse-row))
```

Let's define a small csv document and try to parse it:

```
scheme@(guile-user)> (define document "abc;def;gnu;\n123;456;789;")
scheme@(guile-user)> (parse-csv (string->stream document))
$21 = #<<parse-result> value: (("abc" "def" "gnu" "\n123" "456" "789")) stream: #<stream>>
```

Ooops! There is a single list in the output; there's a bug in a parser! There is no such thing as `\n123` column. So there must be a bug in the `parse-column` parser… The solution is… to not parse columns starting with a newline! So let's use `parse-unless` again:

```
scheme@(guile-user)> (define parse-column (parse-unless (parse-char #\newline) (parse-map ca
```

Our final csv parser looks like the following:

```
(define parse-column-control-char (parse-char #\;))
(define parse-column-value-char (parse-unless parse-column-control-char parse-any-char))
(define parse-column-value (parse-map list->string (parse-one-or-more parse-column-value-cha
(define parse-column (parse-unless (parse-char #\newline) (parse-map car (parse-each parse-c
(define parse-row (parse-map car (parse-each (parse-one-or-more parse-column) parse-eol)))
(define parse-csv (parse-one-or-more parse-row))
```

There is a handy `parse` procedure in guile-parser-combinators that allows to define the following procedure:

```
scheme@(guile-user)> (define (csv string) (parse parse-csv string))
scheme@(guile-user)> (csv document)
$14 = (("abc" "def" "gnu") ("123" "456" "789"))
```

That's all folks! # 2016-01-01 - Getting started with guile UAV database

UAV database is a tuple space database that is easy to use and easy to grasp. There is also a database server that allows to query the database from multiple processus.

We will get started by using the database directly without the database server illustrated with an music album collection kind of stuff and then will open the road to use the database server.

You can fetch the lasted version of UAV database using git:

```
git clone https://framagit.org/a-guile-mind/guile-wiredtiger.git
```

UAV Tuple Space

What is the tuple space? It's table with three columns.

In what follows we consider the following assoc:

```
(define lagale-lagale
    '((artist . "La Gale")
      (title . "La Gale")
      (year . 2012)))
```

Give me a `U`

The first column `U` stands for *unique identifier*. it's a random string assigned the first time you add an assoc to the database. A single assoc is represented with several rows in the table but share the same identifier.

Give me a `A`

`A` stands for *attribute name*. In the above example it's `title` and `year`. In this case attributes are symbols but can be any scheme value that can be serialized with `write`.

Give me a `V`

The last column is `V` for *value*. It can be any scheme value that can be serialized with `write`.

Wrapping up

If we add `lagale-lagale` to the database, and the assigned unique identifier is `ABCDE`, the database will more or less look like the following:

```
   uid  |  attribute |   value
=======+============+===========
 ABCDE |   artist   | "La Gale"
-------+------------+-----------
```

```
 ABCDE |   title    | "La Gale"
-------+------------+-----------
 ABCDE |    year    |   2012
```

API

Fire an REPL inside the wiredtiger directory using the following command:

```
wiredtiger $ guile -L .
```

Open the database

And load the `uav` module with:

```
(use-modules (uav))
```

To get started you need to open database:

```
(define connexion (uav-open* "/tmp/"))
```

Add a document to the database

To add a document to the database you simply format you data into an assoc and use uav-add!:

```
(define uid (uav-add! '((artist . "La Gale")
                        (title . "La Gale")
                        (year . 2012))))
```

Debug the database

There is a debug procedure that allows to have a pick at the underlying schema. Using `uav-debug` you will get something like the following:

```
scheme@(guile-user)> (uav-debug)

;;; (key ("DZI3P5MU" "artist"))

;;; (value ("\"La Gale\""))

;;; (key ("DZI3P5MU" "title"))

;;; (value ("\"La Gale\""))

;;; (key ("DZI3P5MU" "year"))

;;; (value ("2012"))
```

Referencing a document

Now you can use `uav-ref*` to retrieve the document using its `uid`:

```
scheme@(guile-user)> (uav-ref* uid)
$3 = ((year . 2012) (title . "La Gale") (artist . "La Gale"))
```

Update a document

We assigned previously the identifier of *La Gale* album to `uid`.

We can with this information update the document with `uav-update!`. The thing to keep in mind is that this procedure updates the whole document so you need first to retrieve the original assoc, update the assoc and commit the new version using `uav-update!`.

For instance, let's add the *genre* to the assoc:

```
scheme@(guile-user)> (uav-ref* uid)
$6 = ((year . 2012) (title . "La Gale") (artist . "La Gale"))
scheme@(guile-user)> (acons 'genre "Hip Hop" $3)
$7 = ((genre . "Hip Hop") (year . 2012) (title . "La Gale") (artist . "La Gale"))
scheme@(guile-user)> (uav-update! uid $7)
scheme@(guile-user)> (uav-ref* uid)
$8 = ((year . 2012) (title . "La Gale") (genre . "Hip Hop") (artist . "La Gale"))
```

Let's check the output of `uav-debug`:

```
scheme@(guile-user)> (uav-debug)

;;; (key ("DZI3P5MU" "artist"))

;;; (value ("\"La Gale\""))

;;; (key ("DZI3P5MU" "genre"))

;;; (value ("\"Hip Hop\""))

;;; (key ("DZI3P5MU" "title"))

;;; (value ("\"La Gale\""))

;;; (key ("DZI3P5MU" "year"))

;;; (value ("2012"))
```

There is a new key/value pair with a `genre` attribute associated with `"Hip Hop`.

How to delete a document

To delete a document you simply use `(uav-del! uid)` procedure:

```
scheme@(guile-user)> (uav-del! uid)
scheme@(guile-user)> (uav-debug)
scheme@(guile-user)>
```

As you can see `uav-debug` displays nothing, the database is empty!

attribute-value index reference

You can retrieve document using *attribute-value* association. Otherwise said, you can retrieve every document that has a given attribute/value pair in its assoc.

Let re-add `lagale-lagale` to the database an try to retrieve it by year:

```
scheme@(guile-user)> (uav-add! '((artist . "La Gale")
                                 (title . "La Gale")
                                 (year . 2012)))
```

Let's add another album

```
scheme@(guile-user)> (uav-add! '((artist . "La Gale")
                                 (title . "Salem City Rocker")
                                 (year . 2015)))
```

Let's add the album of another artist:

```
scheme@(guile-user)> (uav-add! '((artist . "Mighz")
                                 (title . "Equilibre")
                                 (year . 2015)))
```

Now it's funny enough to query the database! Let's try `uav-index-ref`:

```
scheme@(guile-user)> (uav-index-ref 'year 2012)
$16 = ("3UTO7NW0")
scheme@(guile-user)> (uav-index-ref 'year 2015)
$17 = ("WCKA2IWH" "PQVA9NBB")
```

That's all folks!

2016-01-01 - Getting started with guile-wiredtiger

guile-wiredtiger is a library binding wiredtiger which is database library. I like it because I don't like/master SQL. Other people use it to build real databases like mongodb. Myself I've done one or two toy projects using it like planetplanet clone.

wiredtiger is low level compared to a SQL database.

I can't convince you to use it, but I can show you how it works.

Ground zero

Download wiredtiger 2.6.1 and install it using the usual cli dance. Then retrieve wiredtiger repository with git:

```
git clone https://git.framasoft.org/a-guile-mind/guile-wiredtiger.git
```

Now go to guile-wiredtiger directory and fire a REPL using the following command:

```
guile -L .
```

Mind the dot.

Use wiredtiger

Inside the REPL import the wiredtiger module:

```
scheme@(guile-user)> (use-modules (wiredtiger))
```

Open a connection

Open a connection against a directory say /tmp/wt:

```
scheme@(guile-user)> (mkdir "/tmp/wt")
scheme@(guile-user)> (define connection (connection-open "/tmp/wt" "create"))
```

It's always safe to open a connection against a directory using the create argument even if there is already a database inside that directory.

a is thread-safe.

Open a session

To jump to using the database you have to open a session:

```
scheme@(guile-user)> (define session (session-open connection))
```

a is not thread-safe.

Create a table

wiredtiger is nosql but has a concept of table. It's a two columns layout with sub-columns. The first master column is the key. The second master column is the value.

This dichotomy is useful because the table is automatically ordered using the what is stored in the key.

Let's create simple table with a single string sub-column key, and a single string sub-column value. Yes, columns are typed. Without further addo:

```
scheme@(guile-user)> (session-create session "table:kv" "key_format=S,value_format=S,columns
```

It's safe to session-create a table even if it already exists in the database.

Here we created a table named kv as specified above. The key sub-column is named k and the value sub-column is named v.

Naming columns is optional but required if you want to build indices. Create an index

To create an index we use the same session-create procedure with another configuration string.

Say, we want to invert k sub-column with v sub-column ie. create an index table where the key single sub-column is v from kv table. We can use the following command:

```
scheme@(guile-user)> (session-create session "index:kv:inverse" "columns=(v)")
```

This instruct wiredtiger to add a row to index:kv:inverse table everytime a row is added to table:kv where the key single column content is the content of the column named v in table:kv row.

The index is always synchronized with the reference table for updates, deletes and inserts.

Getting started with cursors

Insert

A is the way to search, navigate, insert and update a table. This is also used to search and navigate index tables.

Let's open a on table:kv:

```
scheme@(guile-user)> (define cursor (cursor-open session "table:kv"))
```

Let's add a record ie. a key/value pair inside the table:

```
scheme@(guile-user)> (cursor-key-set cursor "key")
$4 = 0
scheme@(guile-user)> (cursor-value-set cursor "value")
$5 = 0
scheme@(guile-user)> (cursor-insert cursor)
$6 = #t
```

This is bit involving for our simple case of single sub-columns configuration. Define a procedure to add a record with a single procedure:

```
(define (cursor-insert* cursor key value)
  (cursor-key-set cursor key)
  (cursor-value-set cursor value)
  (cursor-insert cursor))
```

And put it to good use:

```
scheme@(guile-user)> (cursor-insert* cursor "another" "record"))
$7 = #t
scheme@(guile-user)> (cursor-insert* cursor "something" "else")
$8 = #t
```

Search

To look into the table, you also have to use the wiredtiger primitive cursor-key-set followed by cursor-search or cursor-search-near.

For instance, we can do:

```
scheme@(guile-user)> (cursor-key-set cursor "key")
$10 = 0
scheme@(guile-user)> (cursor-search cursor)
$11 = #t
scheme@(guile-user)> (cursor-value-ref cursor)
$12 = ("value")
```

The return value of cursor-value-ref is a list because there might be several sub-columns in a master column. Similarly cursor-key-ref returns a list:

```
scheme@(guile-user)> (cursor-key-ref cursor)
$13 = ("key")
```

Unsuprisingly this returns key because the cursor was positioned at that key using search. This is not a relevant call to do after cursor-search because cursor-search does an exact match of the key or match nothing. This is not the case of cursor-search-near which use some heuristic to find the nearest key. We will study this, but first let's define cursor-search-near star:

```
(define (cursor-search-near* cursor key)
  (cursor-key-set cursor key)
  (cursor-search-near cursor))
```

And try it:

```
scheme@(guile-user)> (cursor-search-near* cursor "ke")
$14 = 1
scheme@(guile-user)> (cursor-key-ref cursor)
$15 = ("key")
```

As you can see cursor-search-near star ie. cursor-search-near returns 1 instead of true or false like cursor-search. cursor-search-near is very useful. I warmly recommend you have a look at its documentation. Navigate

We can verify that the table is correctly ordered. Remember we inserted the following keys: key, another and something.

So in theory we should see them appearing in the alphabetic aka. the lexicographic order.

Let's reset the cursor position and navigate the database:

```
scheme@(guile-user)> (cursor-reset cursor)
$21 = #t
scheme@(guile-user)> (cursor-next cursor)
$22 = #t
scheme@(guile-user)> (cursor-key-ref cursor)
$23 = ("another")
scheme@(guile-user)> (cursor-next cursor)
$24 = #t
scheme@(guile-user)> (cursor-key-ref cursor)
$25 = ("key")
scheme@(guile-user)> (cursor-next cursor)
$26 = #t
scheme@(guile-user)> (cursor-key-ref cursor)
$27 = ("something")
```

All is good.

You can also mix cursor-search[-near] with cursor-next and cursor-previous. Don't forget to read wiredtiger error output ;)

Remove

To remove a record, just set cursor's key content, an remove it using cursor-remove:

```
scheme@(guile-user) [1]> (cursor-key-set cursor "key")
$28 = 0
scheme@(guile-user) [1]> (cursor-remove cursor)
$29 = #t
```

Let's check that this code, is doing what it's supposed to be doing:

```
scheme@(guile-user)> (cursor-key-set cursor "key")
$30 = 0
scheme@(guile-user)> (cursor-search cursor)
$31 = #f
```

cursor-search returns false, which means the key is not found.

Try again with cursor-search-near star.

2016-01-01 - Somewhat Relational Database Library Using Wiredtiger

This is kind of click bait title, because I'm not going to build a relational (not even remotly) in this article. But write about how one can use wiredtiger database like a rdbms.

People don't seem convinced that wiredtiger is the best solution, as of right now, to create database backed application in Guile. Maybe I drink too much of my cool aid. But let me try to convince you again.

This time no fancy algorithms, no, no, no minikaren. We will build a social blogging application using wiredtiger to explicit the fact that it can be used as an RDBMS. So no fancy tricks outside simple RDBMS like tables and indices.

And by the way, if you need more performance or other features in wiredtiger let me know.

If you did not read Getting Started With Guile Wiredtiger, please do.

wiredtiger can be downloaded using the following command:

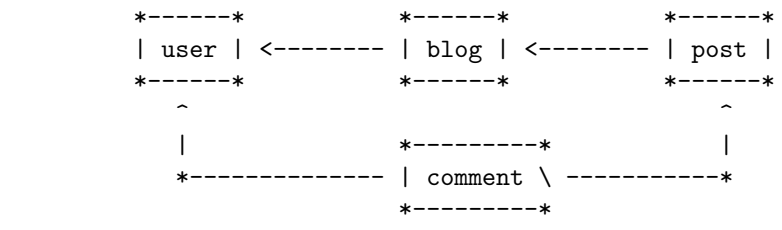
```
git clone https://framagit.org/a-guile-mind/guile-wiredtiger.git
```

To create a guile wiredtiger database you need a Guile REPL.

Ready?!

Schema

I don't recall the precise semantic that must be used to describe a relational database schema so I hope the following will do the trick:



That is all.

Another glimpse into wiredtiger

Let's define using the `wiredtigerz` DSL tables and indices for all the above tables. Remember the language looks like the following:

```
(table-name
 (key assoc as (column-name . column-type))
 (value assoc as (column-name . column-type))
 (indices as (indexed-name (indexed keys) (projection as column names))))
```

Here is the schema of this simple social blogging app platform:

```
(define user '(user
               ((uid . record))
               ((username . string)
```

```
                (bio . string)
                (created-at . unsigned-integer))
               ()))  ;; no index

(define blog '(blog
               ((uid . record))
               ((user-uid . unsigned-integer)
                (title . string)
                (tagline . string))
               ((user-to-blog (user-uid) (uid)))))

(define post '(post
               ((uid . record))
               ((blog-uid . unsigned-integer)
                (title . string)
                (body . string)
                (created-at . unsigned-integer))
               ((blog-to-post (blog-uid) (uid)))))

(define comment '(comment
                  ((uid . record))
                  ((post-uid . unsigned-integer)
                   (user-uid . unsigned-integer)
                   (body . string)
                   (created-at . string))
                  ((post-to-comment (post-uid) (uid))
                   (user-to-comment (user-uid) (uid)))))
```

There is several (!) ways to go on now, I try to make the API simple in the simple case of single threaded applications. So will go on with that API for now.

`wiredtiger-open` is a do-it-all procedure that return two values. You will use that:

```
>>> (use-modules (ice-9 receive) (wiredtiger) (wiredtigerz))
>>> (define cursors (receive (db cursors) (wiredtiger-open "/tmp" user blog post comment) cu
```

`cursors` is an assoc where actual cursor symbols are associated with `<cursor>`. Since all our table have a single record key column there is three kind of cursors for each table:

- `gnu-append` where `gnu` is the name of the table. This kind of cursors allows to insert (or more precisly append) a row in the `gnu` table.
- `gnu` which is the cursor useful for doing something else ie. not append a row.
- `gnu-index` where `gnu` is the name of the table and `index` the name of

the… index. This cursor looks like `gnu` cursor except that it's read-only.

Inserting rows

Now we will create a basic user:

```
>>> (cursor-value-set (assoc-ref cursors 'user-append)
                      "amz3"
                      "Guile hacker 4 ever"
                      (current-time))
>>> (cursor-insert (assoc-ref cursors 'user-append))
>>> (define amz3 (car (cursor-key-ref (assoc-ref cursors 'user-append))))
```

`amz3` variable contains the uid of the created user.

Let's add a blog:

```
>>> (cursor-value-set (assoc-ref cursors 'blog-append)
                      amz3
                      "cryotoptography"
                      "random musing")
>>> (cursor-insert (assoc-ref cursors 'blog-append))
>>> (define cryotoptography (car (cursor-key-ref (assoc-ref cursors 'blog-append))))
```

Wonderful!

Let's define a small procedure to *insert* rows to a table quickly:

```
(define (insert cusors table . args)
    (let* ((cursor-name (symbol-append table '-append))
           (cursor (assoc-ref cursors cursor-name)))
        (apply cursor-value-set (cons cursor args))
        (cursor-insert cursor)
        (car (cursor-key-ref cursor))))
```

Let's add a few data to the database

```
>>> (define hyperdev (insert cursors 'blog amz3 "hyperdev" "guile musing"))
>>> (insert cursors 'post hyperdev "RDBMS in GNU Guile" "start with (use-modules (wiredtiger
>>> (insert cursors 'post hyperdev "GraphDB in GNU Guile" "cf. RDBMS in GNU Guile" (current-
>>> (insert cursors 'post hyperdev "Ahah moment" "Goofing while developping GNU Guile applic
>>> (insert cursors 'post hyperdev "A glimpse into opencog" "opencog is a kitchen sink" (cur
>>> (define abki (insert cursors 'user "abki" "old good things are old" (current-time)))
>>> (define protractile (insert cursors 'blog abki "protractile" "never ending story"))
>>> (insert cursors 'post protractile "Brief introduction to mezangelle" "mezangelle is code
>>> (insert cursors 'post protractile "An Algorithm for poetry" "replace word by definition'
```

Resolving foreign keys

I forgot to add an index to retrieve users by usernames. So let's assume that we know the identifier associated with usernames like `abki` and `amz3`.

So we have usernames, let's lists blogs associated with `amz3`. But first let create a procedure to easily select a single row based on its primary key (or record number).

```
(define (ref cursors table key)
    (let ((cursor (assoc-ref cursors table)))
          (cursor-search* cursor key)
          (cursor-value-ref cursor)))
```

Let's check that is works correctly:

```
>>> (ref cursors 'user amz3)
("amz3" "Guile hacker 4 ever" 1466961863)
>>> (ref cursors 'user abki)
("abki" "old good things are old" 1466964117)
```

Now we can easily resolve primary keys to rows. Indices as defined previously only reference the row primary key. This can be configured otherwise for performance tuning reasons but from a cognitive load point of view it's easier to only introduce primary key as index values.

So let's find out what blog has `amz3` and `abki`:

```
>>> (define user-to-blog (assoc-ref cursors 'blog-user-to-blog))
>>> (for-each (lambda (key) (pk key (ref cursors 'blog key))) (map cadr (cursor-range user-t
>>> (for-each (lambda (key) (pk key (ref cursors 'blog key))) (map cadr (cursor-range user-t
```

Similarly we can retrieve the posts associated with a given blog:

```
>>> (define blog-to-post (assoc-ref cursors 'post-blog-to-post))
>>> (for-each (lambda (key) (pk key (ref cursors 'post key))) (map cadr (cursor-range blog-t
```

Pagination

Pagination is just a matter of slicing the list of primary keys that you retrieve during index lookup.

That's said sometime you don't have uids at all! And you need to slice the table directly. In this case it might be faster to retrieve all primary keys of the table, slice that list of primary key and then `ref` the primary keys. But this is only a performance trick!

2017-01-01 - API cognitive load

A problem can have several solutions. A solution can have different implementations each of which with their own strengths and weakness in terms of space, time and genericity.

The goal of a look'n'feel api design is to optimize the cognitive effort required to understand, learn and use an API. Otherwise said, api look'n'feel aim to improve ease. API look'n'feel has nothing to do with the target domain.

While looking for ease one might trade it over expressive power.

Mastering the look'n'feel of an API involves mastering the cognitive load that it inhertent to it. Cognitive load theory has been designed to provide guidelines intended to assist in the presentation of information in a manner that encourages learner activities that optimize intellectual performance

Here is some of those guidelines:

- People learn more effectively when they can build on what they already understand
- Take advantage of schemata
- Take advantage of visual thinking and other non-verbal thought
- The average person can retain only seven "chunks" of information in (short-term) memory
- Repeat!

2017-01-01 - C'est jamais fini!

Un an et demi à peu près depuis mon dernier post d'humeur. À croire que je n'ai pas d'humeur. C'est peut-être juste que je préfére passer par des couloirs peu emprunté pour expliqué mon moi.

On m'a fait la remarque que je ne terminais jamais ce que je faisait. Au debut je trouvais ça con. Ensuite j'ai compris que j'avais jamais expliqué ce que j'essayais de faire.

En gros, en master j'ai pas pu rejoindre le master Traitement Automatique du Langage (TAL) dans mon université et j'ai un master "bateau" d'intégration logciel. J'ai un peu perdu mon temps. Heureusement il y avais des cours d'IA quand même. Chemin faisant je suis rentrée dans le jeu de l'intégrateur logiciel ou la composition (hierarchique ou pas) d'application à la Django (beaucoup mieux abordé dans aiohttp). J'ai beaucoup écrit et le contexte problement m'a amené à me questionner sur l'utilisabilité du code. Concretement par exemple, comment crée une app Django generique. Ce qui est bien different d'un travail d'integrateur de logiciel. Il s'agit de fournir un outils commun pour travailler ensemble. Quelque part j'ai franchi une frontière. La compétence de se mettre dans la peau de quelqu'un qui lit votre code est aussi utile quand vous integrer des logiciels existant. Cela dit integrer c'est surtout (à mon sens) écouter l'autre donc c'est moins evident de se rendre compte que quelqu'un sera à votre place quand il relira votre code.

A cette epoque j'ai decouvert neo4j sur #django-fr. En creusant je suis tombé sur Tinkperpop Gremlin dont je suis tombé amoureux. J'étais hyper fan des graphdbs.

Sur le chemin du poid cognitif associé à du code je me suis rendu compte que le fait que Python ne puissent pas lancer “naturellement” des chemins de code en parallèle me complexait. Aussi, berkeleydb ne fonctionne juste pas assez bien en mode multiple processus. Et dans le cas d'un processus de base de donnée asyncio était pas une solution. J'ai même participé au projet PyPy STM avant de me rendre compte que c'était pas pour tout de suite.

Je suis passé de Python à GNU Guile, en gros pour ça (c'est aussi une façon facile pour moi de contribuer au projet GNU). Du coup, j'ai ramené mes petits copains wiredtiger, graphdb et idées grandiloquentes

J'ai implementé plusieurs schema pour la graphdb puis j'ai decidé d'utilisé le schema EAV que j'ai porté en Python dans AjguDB (désolé, il n'y a pas de logo).

En GNU Guile, j'appelle ça UAV un acronyme pour Unique-identifier, Attribute, Value.

J'ai continuer mes idées un peu loufoques d'implementer un clone de, et j'ai repris mes recherches.

C'est jours-ci c'est culturia que je vais pas finir.

> Peut-être que quand on franchit une frontière, on oublie tout.

2017-01-01 - Do It Yourself: an artificial intelligence

The main goal of artificial intelligence (AI) is to translate human knowledge into equations.

The problem that arise and still holds is that it can't be solved on a human time scale with current technology.

The problem must be reduced to a problem that can be solved under time constraints pratical for mortals.

Machine learning produce projections of human knowledge over a smaller number of dimensions to make the computation pratical and sometime infer knowledge about how the results was done.

Instead of trying to put order in the mix, my idea is to increase the entropy of knowledge graph using grounded topologies.

Building such structure can be done though personnal note taking applications, croudsource applications and puzzle games.

Not only we leverage Human knowledge and computation skills but append to the already mastered ways of the intelligencewith improved logical and fuzzy reasoning but also push further the use of pattern matching cognition skills that the brain already has.

The opportunities to create cogni-games are manyfolds. People with disabilities can be offered new ways to interact with the world in a meaningful and enjoyable way.

cogni-art and cogni-game are really what should.be+is all about anthroposcene.

```
Anthropo[morphically_s]c[r]e[e]ne[d]
Anthropoc[Sh]e[e]ne[d]
AnthroP[r]o[bos]c[is]ene
--
AnthropO[bs]cene
--
```

Anthropo[S]ceney//AnthropO[bs]cene by Mez Breeze

DIY.

2017-01-01 - Do It Yourself: a search engine in Scheme Guile

Search engines are really funny beast they are a mix of algorithm, architecture and other domain knowledge from databases, linguistic, machine learning and graph theory.

I don't have the prentention to know all of those but with a help of background knowledge, NLP coursera and another description of a search engine I will try to buiolg ahem build and blog about a search engine mocking the different parts and hopefully make a proper release at some point.

Taking orders from space

To get started let's focus and the high level view of the problem. What we want is to be able to search for documents. There is two tasks that must be tackled.

First is the task of return a ranked list of results for a given query. A query is a set of words that we look in a database. We will massively simplify the problem of matching a query to documents by considering that a document is a match if every word from the query appears in it. We will put aside the fact that they might be relevant synonyms or that the query has typos.

The more times words from the query appear in a document higher the score of the document.

The second task is the task of gathering and pre-processsing the documents into the database.

Puting ones and zeros together

The first task, querying, comes backward because in reality we must first store the documents in the database otherwise there is nothing to query. Building the

query engine without having the datastructure of the data written won't work.

That's why we will start with the second task of storing the data inside wiredtiger tables and then implement the querying and then ranking.

It can be built with a RDBMS database with or without LIKE but I don't want to use a RDBMS. If you want you can follow using you prefered SQL ninja foo but at least do not use LIKE to make the exercise more interesting. In wiredtiger there is no LIKE, part of the task is to implement it.

We will implement unit tests, but it's really nice to have a real corpus to test the application. One good candidate are browser bookmarks but html requires preprocessing and more code is required to retrieve the documents… To cut the chase, we will use the bbc news articles dataset as real corpus. You can use whatever plain text documents you have. It's best to use a corpus you know well.

Before reading the solution, think a little while about how you would implement it yourself. Read about wiredtiger (it's basically a hashmap). Imagine a solution in some language whether it's diagrams, scheme or another programming language.

Turning plain text files to scheme data

Going through a set of files

First download the bbc dataset. Extract it somewhere, for instance in ~/src/guile/artafath. artafath means share the light, literaly ar is give back and tafath is light so it reads literally give back the light.

We need a procedure to iterate over all the files, we will use file tree walk procedure to implement a (for-each-file directory extension proc) procedure that will iterate over all files that have EXTENSION as extension inside DIRECTORY and execute a procedure PROC which is passed the path of a matching file:

```
(define (for-each-file directory extension proc)
  (ftw directory (lambda (filename statinfo flag)
                   (match flag
                     ('regular (proc filename)
                               #true)
                     ('directory #true)))))
```

I choosed to do pattern matching.

for-each-file only excutes proc (just like for-each) and doesn't return anything useful (like fold or map) so onward we will only think in terms of file.

Naive parsing of text files

We could simply store the content of the file in a column associated with its filename. The problem with that solution is that we can not easily count the

number of times a word appears in the text without going through the text which has a algorithmic complexity of O(n) instead we will convert the text into a sparse matrix counting the number of times a word a appears in the text. We will call this sparse matrix the bag of words or simply bag.

We must read the content of the file and turn it into a scheme string, reading file as string is done with (ice-9 rdelim) module. For this operation we can simply do:

```
(define (file->string filepath)
  (call-with-input-file filepath read-string))
```

Save this procedure it's useful.

Now we need to 1) lowercase the string so that queries will be case insensitive, words of the query will also be lower case. 2) turn the string into token, which in this case means parsing words. We will adopt a simple approach which works great, most of the time, for english by removing punctuation and splitting by space 3) count the words and store everything in an association.

Let's implement 1) and 2) in (string->tokens text) procedure:

```
(use-modules (srfi srfi-26)) ;; for cut

(define punctuation (string->list "!\"#$%&\\'()*+,-./:;<=>?@[\\]^_`{|}~"))

(define (clean text)
  "Replace punctuation characters from TEXT with a space character"
  (string-map (lambda (char) (if (list-index punctuation char) #\space char)) text))

(define split (cut string-split <> #\space))

(define (sanitize words)
  "Only keep words that have length bigger than one"
  (filter (lambda (word) (< 1 (string-length word))) words))

;; compose must be read from right to left
(define string->tokens (compose sanitize split string-downcase clean))
```

srfi 26 provides the handy cut form.

The third steps is to build the bag of words, a mapping between a word and the number of time it appears. It's a sparse vector. We use this to 1) avoid to store all the text 2) we make the data more computer friendly, kind of… 3) we pre-compute the values that we will need later to match the query against the documents.

The following procedures takes an association BAG and increment a the integer associated with WORD in a persistent way ie. without mutation, it creates a new association (no exclamation mark ! as suffix):

```
(define (bag-increment bag word)
  (let ((count (assoc-ref bag word)))
    (if count
        (let ((bag (alist-delete word bag)))
          (acons word (1+ count) bag))
        (acons word 1 bag))))
```

Now we will have to use recursive loop thanks to a named let to iterate over the words to create the bag of words:

```
(define (words->bag text)
  (let loop ((bag '())
             (text text))
    (if (null? text)
        bag
        (loop (bag-increment bag (car text)) (cdr text)))))
```

Check point. The following program:

```
(define tokens (string->tokens "Janet eat a kiwi! A kiwi!"))
(pk (words->bag tokens))
```

Outputs the following:

```
(("kiwi" . 2) ("eat" . 1) ("janet" . 1))
```

This must be turned into a unit test. Use this template to create one:

```
(use-modules (srfi srfi-64)) ;; unit test framework

(use-modules (artafath))


(test-begin "artafath")

(test-group "string->tokens"
    (test-equal "integration" (string->tokens "Janet eat a kiwi! a kiwy!") '()))
(test-end "artafath")
```

Remains to store the data in wiredtiger and rank a query against the database.

DIY.

2017-01-01 - En avant le web avec Python 3

A l'approche de la pycon france 2017 (à Toulouse (du 21 au 24 Septembre)). Je reflechissait à proposer une présentation de `asyncio` ou aiohttp. La solution proposer par les developpeurs de CPython pour faire du dev asynchrone.

Aussi je me suis dit qu'avant de faire ça se serait bien de faire le point sur papier numerique pour voir si ce que j'avais à dire était assez interessant.

Pour dire vrai, je ne suis pas developpeur CPython, ni `aiohttp` et je pense pas avoir assez de billes pour faire une conf uniquement sur ça. Donc voilà ma reflexion sur le sujet du developpement web en Python.

Kesako la programmation asynchrone?

A vrai dire, on s'en fou! Tout ce qu'on peut garder à l'esprit c'est que c'est plus facile de traiter beaucoup de clients avec.

Kesako `asyncio` ?

`asyncio` est une bibliothèque (de programmation asynchrone) qui vise à tirer partie de la nouvelle syntaxe `async` et `await` introduit dans la PEP 492.

On connaissait déjà la programmation asynchrone en Python 2 grace à twisted et son framework web Nevow utilisant les callbacks. On peut resumer l'experience twisted au points suivants:

- Python c'est bien
- Les callbacks caymal

`asyncio` s'inspire de twisted mais utilise dans son incarnation en Python 3.5 les mots clefs `async` et `await` qui permettent la programmation asynchrones sans callbacks! Et c'est pour ça qu'on peux mettre de coté toutes les subtilités de la programmation asynchrone dans la majorité des cas (et c'est pourquoi on s'en fou au final que se soit asynchrone).

Un code python 3.5+ ressemblera à un code python 2, saupoudré des mots-clefs `async` et `await`.

Kesako `async` et `await` ?

`async` et `await` c'est des nouveaux mots-clefs du langage Python.

Dans `asyncio` si une fonction est declaré comme `async` alors `await` peut apparaitre à l'interieur de cette fonction. `async` est utilisé dans `asyncio` pour declarer une fonction asynchrone. `await` est utilisé pour appeller une fonction asynchrone. Typiquement votre code va ressembler à quelque chose comme ça:

```
async def ma_fonction_asynchrone():
    out = await une_autre_fonction_asynchrone()
    return out
```

Mais j'ai dit qu'on en avait rien à faire de la programmation asynchrone?! C'est là en quelque sorte que l'abstraction au dessus des callbacks **leak** de façon controllé. Mais c'est pas grave, car l'utilisation de ces mot-clefs est clairement identifiés et fait partie de l'API des bibliothèques qui utilisent `asyncio`. Suffit de faire ce qui est marqué dans la doc!

Derrière toussa, il y a un protocole (comme le protocole des itérables utilisé pour faire fonctionner les boucles `for`). Jetez un oeil à la PEP 492 si vous

voulez en savoir plus. Mais sachez que c'est pas la peine de lire et comprendre ce document pour utiliser `asyncio`. J'en suis la preuve vivante.

En fait, `async` et `await` sont uniquement là pour rendre la programmation asynchrone explicite et donc plus facile à maintenir. Cela permet de savoir en un coup d'oeil si une fonction (ou méthode…) peut etre interrompue et où elle pourra etre interrompue… Cela dit c'est pas vraiment la peine de s'en soucier dans une première approche de la programmation asynchrone qui se limite à l'utilisation des bibliothèques existantes.

Pour pas vous laisser desoeuvré fasse à cette nebuleuse asynchrone, je rajouterai juste que ça deviens interessant de savoir quand une fonction peut-etre interrompue quand plusieurs flux d'execution accèdent à la meme ressource. Autrement dit, c'est un usage avancé (sauf si vous creez vous meme des globales (ou autres valeurs partagées) dans votre programme (accessible en ecriture (si les globales sont accessibles en lecture uniquement cela ne pose pas de problème))).

En gros, il suffit de savoir que vous serez obligé de marquer les fonctions asynchrones comme `async` et utiliser `await` pour les appeller comme dit dans la doc. Et c'est tout!

Kesako `aiohttp`

`aiohttp` est un cadre de developpement web qui inclus:

- un serveur et un client HTTP classique
- un serveur et un client WebSocket qui peut participer à la boucle d'évènement que le serveur HTTP classique. C'est à dire qu'un meme executable peut servir les client HTTP et websocket et partager les ressources tel que les connections à la base de donnée. Et comme c'est un seul programme on peut programmer salement!
- un mecanisme de routage à la Django et Flask basé sur les regex qui supporte les applications recursives.

Et c'est tout et c'est déjà pas mal je trouve. D'autre bibliothèques viennent completer ce framework (un peu comme dans l'écosystème Flask) cette base et fournissent connections à la base de donnée à travers SQLAlchemy (ou pas!) ou meme une abstraction aux systèmes de cache REDIS ou memcached

Contrairement à un avis repandu ça ressemble beaucoup à du code python 2 classique à la Django ou Python saupoudrer de `async`/`await`. Ça change des habitudes et c'est plus relou que la fonction `print` en terme de reflexe (surtout que les erreurs sont pas toujours clairs, en tout cas en Python 3.5.3).

Le seul cas qui peux arriver avec aiohttp qui n'arrive jamais en python 2 synchrone c'est l'annulation d'une coroutine et donc du controlleur d'une requête. Ce qui peux poser problème si par exemple on met à jour à la fois la base

de donnée et le cache ou si on **n**'utilise **pas** de transactions ou si on fait de l'autocommit…

`aiohttp` vs. Django vs. Flask

Je vais pas mentir, le gros souci par rapport aux approches synchrones (lire Django et Flask) c'est le manque d'utilisateurs et du coup y a moins de bibliothèques prêtes à etre utiliser ou on trouve encore des bugs… Mais c'est pas comme si les applications Django étaient sans bugs…

`aiohttp` c'est un peu comme flask (avec le support des websocket dans le même thread en plus), il y a pas d'ORM par défault, pas de framework de formulaire par défaut etc… Donc un peu comme en Javascript il faut avoir des idées sur comment monter un cadre logiciel complet.

J'ai commencer le dev avec Django est je trouvai ça super simple, la doc est clair (et maintenant en plus en français) et des centaines d'applications pour démarrer son projet rapidement et à vrai dire je recommanderai encore d'utiliser Django (plutôt que Flask ou `aiohttp`) pour prototyper (sauf si on a besoin de websocket).

Après le stade du prototype, Django et Flask c'est pas le top.

Apparemment les performances sont moins bonnes, à vérifier.

Mais en plus les choix qui sont fait sont pas toujours geniaux:

- Je trouve qu'utiliser un ORM c'est se tirer une balle dans le pied, par exemple le problème des requêtes N+1. J'ai cru au début que l'ORM m'éviterai d'apprendre le SQL. Faux. Déjà il faut apprendre comment fonctionne l'ORM (aurevoir SQLAlchemy). Ensuite il est encore nécessaire de comprendre le SQL pour analyser les requêtes générer par l'ORM. Donc au final, pourquoi ne pas simplement utiliser le SQL qui est enseigné de base dans toutes les formations que je connais.
- L'utilisation massive des globales (apps, url router, template, database connection, requete…) ce qui rend le code imbitable cf. dans Flask cf. `ctx.py` et `globals.py` ou dans Django la gestion de plusieurs bases de données.
- Mélanger la validation des données et le rendu, c'est bien dans le cadre d'un prototype et ça s'arrête là.

Là je me fourvoie peut-être, mais la dernière fois que j'ai regarder il y a deux ans, le support des apps dans Django n'était pas suffisant. Disons que je créer une application pour vendre du pain bio. Je met en place la partie métier spécifique kivabien:

```
painbio = web.Application()

business = web.Application()
```

```
## ajouter les routes kivonbien
business.add_route('POST', '/command', handle_command)
business.add_route('GET', '/receipt/{id}/', handle_receipt)

painbio.add_subapp('/', business)
```

Maintenant, je souhaite ajouter un petit blog à mon application pour faire de la propagande. En Django, ce n'est pas aussi simple que:

```
blog = nerfed.Blog()
painbio.add_subapp('/blog', blog)
```

En Django, il faut ajouter toutes les app dependantes une par une dans le bon ordre pour que ça marche dans le `settings.py` en passant par une indirection sous forme de chaine de caractères. C'est un peu dommage. Alors qu'avec `aiohttp`, on a un début de réponse convenable avec les applications recursives.

Avec toussa on pourrai ajouter que Django channels va ressoudre le problème de riches tel que les websockets, voyez vous meme, c'est pas simple, pas accessible tout sauf la philosphie Python.

CQFD

Je vais continuer à travailler sur mon *nouveau* projet socialite et au final faire quelque chose en me basant sur ce travail l'an prochain, ce qui je pense aura plus d'intérêt.

> Note: 2018/02/11, en fait ça m'étonnerai que je libère du temps pour faire quoi que se soit en Python sur mon temps libre.

Et demain ?

Python dans ton browser?!

Commentaires sur le journal du hacker

2017-01-01 - Getting Started With State of the Art Frontend Development

Nowdays I do a mix of Python system programming and Web UI developements using Javascript (scss, backbone, classy, bootstrap).

I've diven a bit into state of the of the modern way of doing things, prolly synonymous of state of the art. There is in my opinion three main libraries that fight for the spotlight:

- Angular, which is basically a dead project
- React/Redux, the most popular solution, but it's very modular.

- Vue, is kind of inspired from react/redux, similarly it's very modular but the development is mostly centralized.

All three of them have a common point which is `diff+patch` algorithms. That algorithm allows to declare the way the html should look in its entierity and the algorithm make it happen in the rendering graph by updating/remove/adding rendering nodes (which are in the case of webdev, most of the time: DOM objects).

I kept that idea. I created bindings on top of snabbdom.js and built what I could build, first getting inspiration from elm and redux to come up with the most minimald and most versatile framework for building web apps.

The canvas offered by this framework is summed by the following `mount` procedure:

```
(define (mount container init view)
  "Mount in node from the DOM named CONTAINER, the result of the state
returned by INIT passed to VIEW. VIEW must return pseudo sxml where
\"on-fu\" attributes (where fu is DOM event name) are associated with
action lambdas. An action looks like the following:

   (define (button-clicked state spawn)
     (lambda (event)
       (+ 1 state)))

In the above STATE is the current state. SPAWN allows to create
a new green thread. When the action returns the new state, the
VIEW procedure is called again with the new state.

A minimal VIEW procedure looks like the following:

   (define (view state)
     `(button (@ (on-click . ,button-clicked)) ,state))

A minimal INIT procedure looks like the following:

   (define (init) 1)

That's all folks!
"
  (let ((state (init)))  ;; init state
    ;; create a procedure that allows to create new green threads
    (letrec ((spawn (lambda (timeout proc args)
                      (set-timeout (lambda () (apply (proc state spawn) args)) timeout)))
             ;; lambda used to wrap event callback
             (make-action (lambda (action)
                            (lambda args
```

```
                              (let ((new (apply (action state spawn) args)))
                                (set! state new)
                                (render)))))
                    ;; rendering pipeline
                    (render (lambda ()
                              (set! container (patch container
                                                     ((sxml->h* make-action) (view state)))))))
      (render))))
```

Basically, it says that `INIT` procedure produce a seed state passed to `VIEW` which renders the first version of the graph scene. A node from the graph scene fires an event, user specified callbacks are which are called action procedures which basically takes everything they need to:

- do ajax without blocking using an imperative syntax (via call/cc)
- spawn new green thread
- update the state

That's where will land the business code.

The `VIEW` procedure contains the UI code.

If you want to know more about this project head over the website or have a look at this screencast on youtube.

2018-01-01 - A Graph-Based Movie Recommender Engine Using Guile Scheme

The goal of this article is to introduce the `grf3` graphdb library and its graph traversal framework named traversi. To dive into traversi, we will implement a small recommender system over the movielens dataset.

grf3 graphdb library

`grf3` is graph database library built on top of an *assoc space* persisted to disk thanks to wiredtiger.

Basics

Basically you create vertices using `(create-vertex assoc)` and `(create-edge start end assoc)` where `ASSOC` must be an alist with symbol keys and any serializable scheme value as value. `START` and `END` must be `<vertex>` see below. Those procedures will return respectively a `<vertex>` and `<edge>` record.

Both `<vertex>` and `<edge>` have an assoc and unique identifier named `uid`. They have sugar syntax to interact with the assoc via `(vertex-set vertex key value)`, `(vertex-ref vertex key)`, `(edge-set edge key value)` and `(edge-ref edge key)`. Mind the fact that `-set` procedures have no `!` at the end which means they return a new record.

`<edge>` has two specific procedure called `edge-start` and `edge-end` which will return the unique identifier the start vertex and respectively the end vertex.

`(get uid)` will return a `<vertex>` or `<edge>` with `UID` as unique identifier.

`(save vertex-or-edge)` will persist changes done to a vertex or edge.

That is all for the basics of the `grf3` library. If you experiment a little with the library in the REPL you will notice that the `assoc` associated with `<vertex>` and `<edge>` contains more than what you put in it. NEVER change the keys that starts with `%` char or the database will break.

traversi stream framework

traversi is a port of Tinkerop's Gremlin to scheme using streams implemented using delayed lambda evaluation. This does not use srfi-41 because srfi-41 is slower.

The API is similar to srfi-41:

- `(list->traversi lst)`
- `(traversi->list lst)`
- `(traversi-car traversi)`
- `(traversi-cdr traversi)`
- `(traversi-map proc traversi)`
- `(traversi-for-each proc traversi)`
- `(traversi-filter proc traversi)`
- `(traversi-backtrack traversi)` will backtrack to the value that produce the current value by the previous call to `traversi-map`.
- `(traversi-take count traversi)`
- `(traversi-drop count traversi)`
- `(traversi-length traversi)`
- `(traversi-scatter traversi)` takes a traversi of lists (where each value of the stream is a list) and convert all the thing to single traversi composed of the value containes in the lists of the stream.
- `(traversi-unique traversi)` keep only on occurence of each value.
- `(traversi-group-count traversi)` return an alist with the count of each value.

helpers

Here is a few helpers:

- `(vertices)` return a traversi made of all the vertex **uids** found in the database
- `(edges)` return a traversi made of all the edge **uids** found in the database
- `(from key value)` return elements which the value associated with `KEY` is `equal?` to `VALUE`

- `((where? key value) uid)` predicate procedure which will return `#true` if the element `UID` has `VALUE` as `KEY`.
- `((key name) uid)` return the value of `KEY` for the element which has `UID` as unique identifier.
- `((key? name value) uid)` predicate procedure which will return `#true` if the element which has `UID` as unique identifier has a `(name . value)` pair in its assoc.
- `(incomings uid)` return a list made of the *incoming edges* of the element `UID`. `UID` must be the identifier of a vertex. If `UID` is the identifier of an edge it will return an empty list (except if the database is broken).
- `(outgoings uid)` return a list made of the *outgoings edges* of the element `UID`. `UID` must be the identifier of a vertex. If `UID` is the identifier of an edge it will return an empty list (except if the database is broken).
- `(start uid)` if `UID` is the identifier of an edge return the edge's *start node*. Otherwise the behavior is not specified.
- `(end uid)` if `UID` is the identifier of an edge return the edge's *end node*. Otherwise the behavior is not specified.

Last but not least, there is `(get-or-create-vertex key value)` which will create a vertex with `((key . value))` if there is not vertex with such a pair or return the existing vertex otherwise.

Using a graphdb to do recommendation

Now we will put this all together to movie recommendations.

Getting started

First you need a bit of setup.

Clone the culturia repository using the following command:

```
> git clone https://github.com/amirouche/Culturia
```

Rendez vous inside the `src` directory, create a `data` directory and inside that directory download the movielens small dataset and decompress the archive:

```
> cd Culturia/src
> mkdir data
> wget http://files.grouplens.org/datasets/movielens/ml-latest-small.zip
> unzip ml-latest-small.zip
```

Go back the `src` directory, create a `/tmp/wt` directory, study a little the `movielens-step00-load.scm` file using your favorite editor emacs and fire guile to load the dataset as `grf3` database:

```
> cd ..
> mkdir /tmp/wt
> emacs movielens-step00-load.scm &
```

```
> guile -L . movielens-step00-load.scm
....
```

Loading will take a few minutes.

To understand the graph traversal you need to know how the graph is drawn. There is three kinds of vertices:

- movie vertices have a `'movie` label, `'movie/id` and a `'movie/title`
- genre vertices have a `'genre` label and a `genre/title`.
- user vertices have a `user` label and a `'user/id`

There is two kinds of edges:

- edges starting at movies, which connects to a genre vertex labeled as `'part-of`
- edges starting at users, which connects to a movie vertex labeled as `rating`. They also have a `rating/value` integer value.

Here is a schema of the graph mapping:

Figure 5: graphdb mapping

Fire guile REPL and import all the things:

```
> guile -L .
GNU Guile 2.1.3
Copyright (C) 1995-2016 Free Software Foundation, Inc.

Guile comes with ABSOLUTELY NO WARRANTY; for details type `,show w'.
This program is free software, and you are welcome to redistribute it
under certain conditions; type `,show c' for details.

Enter `,help' for help.
scheme@(guile-user)> (use-modules (ukv) (grf3) (wiredtigerz) (wiredtiger) (ice-9 receive) (s
```

Open an environment over the `/tmp/wt` directory:

```
scheme@(guile-user)> (define env (env-open* "/tmp/wt" (list *ukv*)))
```

You are ready!

Forward

First we would like to know which movie has the id `1`:

```
scheme@(guile-user)> (with-context env (traversi-for-each pk (traversi-map get (from 'movie/

;;; (#<<vertex> uid: "V6HM7CEX" assoc: ((movie/title . "Toy Story (1995)") (movie/id . 1) (
```

Remember that every time you access the database in the REPL you need to wrap the call using `with-context`. This is also the prefered way to work with `grf3` in modules.

Now let's see what genres is *Toy Story*:

```
scheme@(guile-user)> (with-context env (traversi-for-each pk (traversi-map (key 'genre) (tra

;;; ("Fantasy")

;;; ("Animation")

;;; ("Children")

;;; ("Comedy")

;;; ("Adventure")
```

Seems good to me. Mind the `traversi-scatter` it used just after `(traversi-map outgoings ...)` this is pattern that you will see very often.

Now let's do something more complex. We would like to know which movies are liked by the same people that liked *Toy Story*.

Remember that compose must be read from right to left, which means in this case bottom-up:

```
(define (users-who-scored-high movie/id)
  (with-context env
   (let ((query (compose
          ;; fetch the start vertex ie. users
          (cut traversi-map start <>)
          ;; backtrack to edge uids
          (cut traversi-backtrack <>)
          ;; keep values that are <= 5.0
          (cut traversi-filter (lambda (x) (<= 4.0 x)) <>)
          ;; fetch the 'rating/value of each edge
          (cut traversi-map (key 'rating/value) <>)
          ;; keep edges which have 'rating as 'label
          (cut traversi-filter (key? 'label 'rating) <>)
          ;; scatter...
          (cut traversi-scatter <>)
          ;; fetch all incomings edges
          (cut traversi-map incomings <>))))
     (traversi->list (query (from 'movie/id 1))))))
```

This is not very useful since we don't know the users, but the final result might be more interesting because we might now the movies (or at least we can STFW).

As an exercise check that the uids that were fetched are really user vertex.

Now will check the users likes and *group-count* to see if it makes sens:

```
(define (users-top-likes users)
  (with-context env
    (let ((query (compose
                  (cut traversi-group-count <>)
                  ;; retrieve movie vertex's title
                  (cut traversi-map (key 'movie/title) <>)
                  (cut traversi-map end <>)
                  ;; retrieve rating edges with a score of at least 4.0
                  (cut traversi-backtrack <>)
                  (cut traversi-filter (lambda (x) (<= 4.0 x)) <>)
                  (cut traversi-map (key 'rating/value) <>)
                  (cut traversi-filter (key? 'label 'rating) <>)
                  (cut traversi-scatter <>)
                  (cut traversi-map outgoings <>)
                  (cut list->traversi <>))))
      (list-head (query users) 10))))
```

The above algorithm is available in the `culturia/src/movielens-step01-recommend.scm` file.

See by yourself!

2018-01-01 - a work in progress

A fight a lot a painful thought when I am publishing something on the Internet: Am I toxic? I sure this won't stop today, nor tomorrow.

I have dozens of repositories online, I feel somewhat shameful about it because most of them don't work anymore. I have the feeling that it gives a bad impression. That said, my belief that those will be useful for me and for other people is stronger.

I repeat myself that a) I do the right thing by sharing my *experience* b) It's good to talk about it. This is in this spirit that I blog, share code and do other things of life.

I have the right to write a software trying to understand how opencog works and call it culturIA. Thinking in terms of words Intelligence Augmentation instead of Artificial Intelligence. Using that very same name for a versioned hypergraph, and then mixing it with my grandiloquent blog engine (read a blog that is not static html) that I call a personal knowledge base. Forget about Natural Language Processing because it's too difficult or too soon or because I am not skilled enough. And go back to it. I should feel free to say *cyberspace transderivational search* without feeling ashamed. I should feel free to reinvent the wheel (on my free time (at least)). To create a search engine and fail and

try again later. To think that I can do better than others. I should feel free to be a creative mind.

I have the right to dream big.

I should feel free to do all those things that are not toxic, but I don't feel that way.

There should be a place for everyone, I hope one day I will be confortable at mine.

2018-09-24 - Comment on justifie un ORM ?

Reponse a l'article de Sam Les critiques des ORM sont à côté de la plaque.

Ma position actuel est que les ORM c'est inutile voir dangereux par rapport a l'utilisation de SQL. Et que cela ne scale pas du tout au gros projet qui doivent etre maintenable et performant.

Mais d'abord faisont une etude du texte originale.

Etude de la plaque

> Rappel: qu'est-ce qu'un ORM ?

Dans cette partie Sam essaye d'expliquer ce qu'est un ORM. Il oublie que le principale ORM Python, l'ORM de Django fait un pot pourrit de differents patrons:

- Object-Relational Mapper a proprement dit qui fait le lien entre la base de donnee et le code Python. A vrai dire, dans l'usage, c'est surtout le code Python qui indique a la base quelle table, quelle colonne avec un type.
- Observable: le model Django a des methodes qui sont appellees au cours du cycle de vie du modele qui permettent en pratique de modifier le model pour une raison x ou y. On verra plus tard que c'est pas utile, pire c'est dangeureux.
- Validation: Et oui, il y a une petite astuce, c'est que le $_{\text{form}}$, je veux dire le model sait se valider mais juste un peu pas trop, car sinon les `Form` il saurait pas quoi faire.
- Facade: il abstrait le langage de requete SQL a travers le tandem `QuerySet` et bien sur son manager.

Aussi, ce qu'on appelle vulgairement un ORM, gere les migrations automagiquement.

> Tout est ecrit en Python

Je suis completement d'accord avec cette idee. Je veux ecrire un maximum de code en Python. Mais je suis oblige de reconnaitre que ce qu'on appelle ORM apporte plus de probleme que de solution.

> SQLAlchemy (pour les gros projets)

Voila un petit commentaire facile qui moutonne encore un peu plus le public. Mon adage prefere c'est les problemes simples doivent etre simple a resoudre et les problemes compliques possible. Ce qui n'est pas le cas de SQAlechemy la derniere fois que je l'ai utilise. En effet, le cout de mise en place fait que vous voulez pas utiliser cela dans un petit projet. Une API qui part dans tous les sens fait que vous vous y retrouvez jamais ni dans la documentation, ni dans les blogs qui ventent ses merites.

> Que reproche-t-on aux ORM ?

Je suis d'accord avec les elements de reponses apportes par Sam dans cette section.

> Heu?

Demarre la critique qui visent a convaincre que les ORM c'est bien.

L'argumentation est mal construire a mon sens mais les arguments sont interessants.

> Les ORM ne servent pas à éviter d'écrire du SQL. Ça, c'est vaguement un effet de bord.

Malheureusement, c'est comme cela comme presente au premier abord les ORM. Et c'est principalement pour cela qu'on les utilisent. C'est une tentative de moutonage.

> Ils servent à créer une API unifiée inspectable, une expérience homogène, un point d'entrée unique, un socle de référence explicite et central, pour le modèle de données.

L'abstraction universelle en un seul mot pour gerer vos data.

Prenons un part un les arguments:

- API unifiée: entre plusieurs bases de donnee, mais comme il en parle pas plus les gens ne saurons pas que personne ou presque n'utilise cette fonctionalite des ORM qui permet de passer d'une base a une autre avec moins de douleurs. D'ailleurs, il ecrit plus tot que SQL fait deja le taf de ce point de vue la…
- inspectable: je ne connais pas mysql, mais postgresql est completement inspectable en SQL.
- une experience homogene: il repete le premier argument
- un point d'entree unique, un socle de référence explicite et central, pour le modèle de données: Ici, il dit deux fois la meme chose et glisse un mot de passe python bien connu "explicite". En quoi le SQL n'est pas explicite? En quoi le fichier de schema n'est pas un point d'entree unique?

Au contraire avec cette abstraction on a l'information duplique dans le code Python et dans la base.

Une fois qu'on a passé pas mal de temps à faire des projets…

Ici il met un coup de guitare a la Django. DRY. Application re-utilisables.

quelqu'un dans l'équipe va commencer à écrire une abstraction…

C'est ca le truc, l'abstraction, selon moi dont on a besoin est tellement simple quelle defit le sens commun.

Pas besoin d'abstraction complique.

On separe la vue du model ainsi que le controller. On valide les entrees dans le controller. On decrit l'acces au model a l'aide du SQL dans une petite fonction ou methode.

Et voila!

L'exemple de Django

Apres Sam va continuer a justifier l'interet d'un ORM parce que Django c'est bien! Donc l'argument c'est Django == bien donc ORM == bien. Wat! Un coup de moutonage au passage, il y a plein de projet Django donc c'est bien Django.

Django et son ecosysteme merite un article a proprement parler.

C'est le moment ou je baisse les bras car l'article est mal construit. Au lieu de faire une serie d'argument, couvrir l'argument completement et donner un exemple. Ici, il repasse sur certains de ses arguments pour faire semblant de les demonter, c'est rigolo.

Pas d'ORM, c'est possible

Je vais pas vous moutoner en disant un truc du genre regardez comment les autres font pour scaler leur Systeme d'Information. C'est uniquement des indices. Et encore c'est parfois des fausses pistes.

Essayez de prendre du recule. Etudier votre code. Oubliez ce que vous croyez savoir. Essayez de comprendre les difficultees liees a votre base de code. Tenez un journal des incidents et bugs. Essayez de faire des rapprochements entre ces elements.

Vous arriverez a une meilleur solution qui resoud le probleme que vous avez et pas celui que croyais avoir les developpeurs de Django.

Indice: quand on tourne toujours a gauche et qu'on se rend compte qu'on tourne en rond. Qu'est ce qu'on doit faire?

2018-01-01 - Getting Started With Guile Dynamic Foreign Function Interface

I created a dynamic-link star procedure to help during the creation of dynamic ffi bindings (which is the prefered way to go now on).

It's helpful for several reasons:

- no need to explicitly call dynamic-func and pointer->procedure, so it's two less procedures to know about.
- It mimicks the C signature, so it's easier to read.

This documentation is meant to be self sufficient which means that you should be able to fully bind your favorite function based C library just by reading this.

The code of the procedure is at the bottom.

```
((dynamic-link* [library-name]) return-type function-name . arguments)
```

Return a lambda that returns a scheme procedure linked against FUNCTION-NAME found in LIBRARY-NAME. If LIBRARY-NAME is not provided this links against the C standard library.

The returned procedure takes the signature of the function that you want to link against.

Also RETURN-TYPE and ARGUMENTS must be foreign types. They can be found in (system foreign): int8, uint8, uint16, int16, uint32, int32, uint64, int64, float, double.

In addition, platform-dependent types variables exists: int, unsigned-int, long, unsigned-long, size_t, ssize_t, ptrdiff_t.

There is also a void variable that must be used to wrap function that returns nothing.

Last but not least, the star symbol is used by convention to denote pointer types.

More documentation about foreign types.

Example

Here is a REPL run, showing how it works:

```
(define stdlib (dynamic-link*))  ;; link against stdlib
(define strlen (stdlib int "strlen" '*))  ;; retrieve a procedure associated to "strlen"
(strlen (string->pointer "abc"))
```

Since you probably don't want to expose the pointer api to the dev. You might define the following strlen procedure:

```
(define stdlib (dynamic-link*))  ;; link against stdlib

(define (strlen string)
   (let ((function (stdlib (int "strlen" '*))))  ;; retrieve strlen function as a procedure
   (function (string->pointer string))))
```

AFAIK, there is no performance gain in memoizing stdlib or function. Where to go from here?

If you need to bind structures the proper way to go is to use scheme bytestructures. The code

```
(use-modules (system foreign))

(define* (dynamic-link* #:optional library-name)
   (let ((shared-object (if library-name (dynamic-link library-name)
(dynamic-link))))
     (lambda (return-value function-name . arguments)
       (let ((function (dynamic-func function-name shared-object)))
         (pointer->procedure return-value function arguments)))))
```

That's all folks!

2018-01-01 - Je suis une bande éthic à moi tout seul

C'est pas juste que j'ai du cœur à l'ouvrage, sans sens apparat à apparaitre aux après de celles et ceux qui revaient d'une enigme, c'est ma passion.

Ce que j'ai fait comme contribution libre en 2017.

Python

Combinatorix

J'ai crée une bibliothèque pour écrire des parsers par composition de fonctions. Au lieu de parser une grammaire, dans le cadre des parseur par combinaison la grammaire c'est le code! Vive combinatorix! Contrairement à d'autres implementations, il n'est pas monadic… enfin pas à ma conaissance.

BeyondJS

J'ai fait un prototype d'un framework inspiré de feu nevow qui permet d'écrire des applications web sans passer par une ligne de javascript et sans fichier HTML externe à l'aide d'une API dont je suis très fier mais que j'ai completement pompé sur nevow. C'est pas exactement nevow non plus car il utise pas twisted, mais aiohttp. Aussi, il est encore moins utile (bien que possible) d'écrire du javascript pour faire des trucs un peu kikoo. Spoiler, ça utilise un virtual dom. La connaissance de les API web (cf. MDN est encore requise. Lisez le code sur github!

Socialite

J'ai essayé de me lancer dans un gros projet mais j'ai juste integrer deux trois bibliothèques ensemble pour que ça ressemble à quelque chose d'utilisable proche de ce que fourni le monolithe Django. Ça utilise ReactJS. Y a pas grand chose, mais ça m'a value quelques forks et étoiles. Be social, star it!

maji

Je sais pas pourquoi j'ai fait ça. Désolé. https://github.com/amirouche/xp-maji

Scheme

scheme-todomvc

C'est mon plus gros succès de l'année! C'est la fameuse todo app adapté en scheme à l'aide de snabbdom et inspiré de la version de Elm sans les signaux. Il y a d'autres versions du "framework" que j'utilise pour faire ça que je préfére à cette version. Mais cette version est la plus présentable. Le coeur du truc (inspiré de Elm) est plus testable. Ce que je retiens de cette experience, c'est que le Scheme à travers le Scheme XML (aka. SXML) est très compétitif par rapport à JSX.

guile javascript backend

Suite au GSoC de Ian Price qui visaient à adapter le backend JavaScript au nouveau code du compilateur de Guile, j'ai corrigés deux ou trois bugs et j'ai rendu possible d'appeller du JavaScript depuis Guile et du Guile depuis JavaScript. Y a des morceaux du truc du scheme-todomvc la dedans. Malheureusement, les navigateurs modernes (!) ne supportent toujours pas l'optimisation des recurences terminales qui est pourtant dans le standard, donc en gros c'est pas utilisable jusqu'à nouvel ordre (ou rappel l'ordre ;)).

Le code est sur gitlab.com cette fois.

guile-termbox

J'ai crée des bindings pour la superbe lib termbox (je recommende ce fork des vraies couleurs :) et un petit editeur de texte tout rikiki en guise de documentation.

Culturia

J'ai pas un, ni deux mais quatre repository culturia, le dernier se trouve chez framagit. Je sais plus précisement ce que je veux faire. Et le terme pompeux c'est Personal Knowledge Base.

Autres

J'ai passé la barre des 3000 points sur stackoverflow et j'ai écrit un challenge full stack que personne à ma connaissance n'a reussit. Il y a des étoiles qui se perdent… ## 2018-09-28 - SQL is the ORM killer

I think the future of data persistence is FoundationDB but I don't want to fight several battles at the same time. For the time being, let's tackle a pervasive pratice: Object-Relational Mapper. This is so much entrenched in programming practice that people don't think about it. It's somewhat like Object-Oriented Programming in a multi-paradigm language, like, say Python, of course I will use a class! Of course, I must use an ORM.

In this article, I will explain why ORM is a poor pratice.

ORM is not a good abstraction

In Python, most people know ORM through Django. Django's ORM is a **mega** abstraction that try to solve several problems:

1. The ORM wants to be a class-based Python representation of the *underlying* RDBMS tables.
2. It's also throught it's `pre_save` and `post_save` hooks a way to enforce some constraints on the data.
3. It's also somekind of facade on top the SQL query language.
4. Abstract differences between several databases

This seems like an anti-pattern on first sight as it does not follow the single responsability principle. But it's worse than that since I omitted that it also try to validate the schema and data. With an ORM you are on your way for god classes that break all designs that aim for separation of concerns like Model-View-Controller.

We will kill all those preconceptions and see how they solved problems you don't have and worse they hinder progress toward scalable applications.

Class-based representation doesn't help

To get started, but many will see it as minor annoyance, if any, most ORM force you to use classes which adds complexity to the code from the start.

Python representation of the database schema is wishful thinking but it doesn't deliver in practice.

First, because of the Object-relational impedance mismatch.

Simply said, the structure of the database doesn't map with Object-Oriented Programming. In simple cases, for instance tables with foreign keys it works, but as soon as you start building a more complex data layer (many-to-many relations,

table inheritance or polymorphic data) you end up with a complicated python representation that is actually a breeze to deal with in the database native language namely SQL. Not only that, but it is made very easy in Python to shoot yourself in the foot because the ORM hides the actual complexity of a given query throught its numerous layers like the `SELECT` n + 1.

Another tooted advantage of having a Python representation of the schema is that you can do introspection. That is nice. Everything in the confort of Python metaclass complex architecture. But that is not an argument anymore since posgresql has introspection. So the same django-admin you all love (meh) can be built with raw pyscopg or better asyncpg. So, introspection is not a good argument.

Some people say, that Python is more readable than SQL. With that I heartedly agree if you disregard the complexity required to create that readable syntax. That being said, having the schema described in Python is not DRY since the schema is also stored in the database.

You won't need `pre_save` and `post_save`

`pre_save` and `post_save` hooks implement, as far as I can tell, some kind of observable pattern. That is, you keep an eye on what happens on the data and enforce some rules. It is like database triggers but done in Python.

I argue that `pre_save` and `post_save` hooks are an anti-pattern. Most likely, the code that goes into those should be in either a validator or a domain function (if you prefer, read "business function"). If you start using `pre_save` and `post_save` you are hardcoding domain behaviors in the model which leads to god classes and ugly `if`.

Last but not least, remember that Django is a framework that was meant to create an ecosystem of reusable applications. That's why it provides hooks and signals. What I want you to notice is that in real world scenario where you build an application for yourself, especially after the Minimum Viable Product or prototype, you head your way to a monolith where you are fully in control. That is completly different from the Django use-case where you want Joe to extend Jane's application.

Embrace the power of SQL

The abstraction on top of SQL is moot. That is ORM does not help you completly avoid SQL and it gives you more guns to shoot yourself.

Of course, SQL has another syntax but the ultimate language doesn't exists, especially if you plan to deliver something this month. You should rather keep thing simple and avoid building too much abstraction in order to keep things in control. Using too much abstraction, is the best way to see the complexity of your project explode.

My point is an ORM helps to write simple queries but it's not easy to write complex queries. Even when you use SQL, you need to fine tune your query to best make use of indices. If you use an abstraction on top of SQL, you will need to think about the correct SQL and then translate that to the correct ORM query incantation.

I won't go into the Django `QuerySet` with its manager shenanigans because again, it leads to god classes with ugly `if` and yet another time it makes the OOP approach the default choice where **a proper use of basic data types and function are good enough.**

The last bit is the same as the first, in the end you will need to know SQL to take full advantage of your RDBMS.

Conclusion

I used to practice (read abuse) a lot Object-Oriented Programming and I loved Django ORM and I was impressed by SQLAlchemy. I grew up (read learned a lot), and I think I know better. OOP can be useful, god objects are not. That's what you will end up with using ORM god objects and bad patterns.

I hope I gave enough elements to answer the question for yourself. I omitted some elements, for instance, when you aim for performance you do not want to `SELECT * FROM table` all the time and that you end up with pieces of data that don't map to your `Model` classes except if you build dozens of classes. Of course an ORM haz a **workaround**! Or you won't use automagic migration tool in big projects.

I also did not dive into the issues related to the use of JSON data types that is now part of modern postgreSQL. That leads to other interesting problem that are partly solved in Object Document Mappers aka. ORM for document stores. I have not extensive experience with those but my choice will be prolly the same. Stick with the basics, see where they are patterns and solve those with carefully choosed abstractions.

Think about it carefully. Known good and somewhat obvious guiding principles like "single responsability principle" or "seperation of concerns", try to be **systematic**: always solve the problem the same way. In particular, ORM are not systematic, simple queries are done some way, complex queries are complicated and not explicit…

Caveat emptor! ## 2019-02-01 - Comment choisir une base de donnee?

Pour choisir une base de donnee vous pouvez vous aidez de StackOverflow et du tag database-design.

Dans votre question il est preferable d'apporter des elements de reponses en decrivant votre besoin:

1. Garanties: Cela ce definit en remettant en question les differents proprietees ACID: Atomic, Consistent, Isolation, Durability. Voir la documentation PostgreSQL sur le controle de la concurrence ou en autre le le chapitre sur les transactions dans wiredtiger. Voir si BASE vous convient.
2. Taille
3. Modele de donnee : A quoi ressemble les donnees. Est ce qu'elles sont de formes heterogenes ou bien il y as des types bien identifies.
4. Workload : Surtout des read, surtout des write, melange ou encore write once, then read-only. Ainsi que les types de requetes qui vont etre executees: recursive / profondes, colonne / lignes ou par proximites. ## 2019-10-07 - FoundationDB

Ceci est un compte rendu approximatif, avec quelque améliorations, au talk que j'ai donné hier chez BackMarket.

A Database To Rule Them All

Derrière ce titre pompeux se cache la volonté a peine voilée de faire de FoundationDB votre principale source de vérité dans votre système d'information.

Mais aujourd'hui je ne veux pas (uniquement) vous vendre FoundationDB et transmettre une idée plus grande. À savoir, comment aborder la programmation des bases de données clef-valeur ordonnées. Contrairement au `dict` Python, ce n'est pas l'ordre d'insertion qui prime mais la comparaison lexicographique entre les octets, autrement dit l'ordre du dictionnaire des langues naturelles.

Kesako FoundationDB

FoundationDB est une base de données ordonnée clef-valeurs scalable horizontalement qui est ACID et respecte les garanties de Cohérence et résiste aux erreurs de Partitionnement dans le cadre du théorème CAP. Les garanties offertes sont similaires a PosgreSQL (c'est à dire CP).

Voir https://apple.github.io/foundationdb/cap-theorem.html

C'est aussi une base de données mieux testée que les bases de données qui ont enduré les tests jespen.io. En fait, l'un des fondateur de FoundationDB a quitté Apple pour se concentrer sur cette méthode de développement (qui est en somme du TDD++) dans une entreprise dédiée à promouvoir cette pratique, dite de la simulation.

Voir https://www.youtube.com/watch?v=fFSPwJFXVlw.

Cela étant dit, l'idée la plus importante pour commencer la programmation avec FoundationDB est l'idée qu'il s'agit d'un *dictionnaire ordonné*. Et cette "ordre" fait de FoundationDB une base très versatile. En préservant les garanties ACID et CP pour les charges temps réel, elle peut s'adapter à n'importe quel(!) modèle de données. D'où l'idée d'en faire la source de vérité principale.

Un dictionnaire ordonné, comme l'arbre rouge et noir, n'est pas uniquement utile pour ranger des entiers dans l'ordre croissant ou décroissant avec une complexité logarithmique. L'ordre permet de créer des structures ou abstraction de plus haut niveau.

Who use FoundationDB

Je vous renvoie vers les FoundationDB Summit de 2018, ainsi que le future summit de 2019 de la Linux Foundation.

TL;DR: Des entreprises qui doivent gérer de gros volume de données.

Note: voir le talk: "Lightning Talk: Entity Store: A FoundationDB Layer for Versioned Entities with Fine Grained Authorization and Lineage" qui discute d'une base de données utilisée dans le cadre de pipeline de data science chez Apple (écrit en Python 2.7 :).

Why use FoundationDB

- Vous avez plus d'un 1TB de donnée
- Vous avez besoin de résistance aux pannes et donc besoin de réplication, YOLO!
- Vous avez de l'argent à investir sur le long terme. Le projet reste jeune et beaucoup de fruits restent à cueillir.
- Vous voulez apprendre quelque chose de nouveau, sans quitter votre zone de confort :)

When not to use FoundationDB

Vous n'avez pas besoin d'utiliser FoundationDB sur vos projet legacy qui tournent bien! Si vous utilisez PosgreSQL pour votre projet perso ou d'entreprise probablement que cela va tenir un certain temps.

Cela dit sachez que FoundationDB, n'est pas la seule base de données clef-valeur ordonnées.

Il y a:

- SQLite LSM extension, voir https://github.com/coleifer/python-lsm-db/
- MongoDB Wiredtiger
- Facebook RocksDB
- Google LevelDB
- Et feu bsddb

Sans oublier TiKV ou Google Spanner (privatif). À ce sujet je recommande la lecture du papier "Spanner: Google's Globally-Distributed Database".

(Et pendant que j'y suis le papier surnommé Large-scale cluster management at Google with **Borg**)

(Note: relire “Fast key-value stores: An idea whose time has come and gone”)

How to program FoundationDB

L’ordre du dictionnaire FoundationDB n’est pas l’ordre d’insertion!

L’ordre du dictionnaire FoundationDB n’est pas l’ordre d’insertion!

L’ordre du dictionnaire FoundationDB n’est pas l’ordre d’insertion!

Grâce à tuple.pack on peut considérer que foundationdb est un dictionnaire de tuples ordonnés selon l’ordre naturel des types de bases `int`, `float`, `byte`, `str`.

Oui vous avez bien lu entre les lignes, il y a un couple de fonctions qui permet de traduire certains types Python vers des bytes qui préservent l’ordre de ces types.

Voir le module `fdb.tuple` dans les bindings Python officiels.

En gros:

```
from fdb import tuple


before = (1, 2, 3)
after = (10, 20, 30)

assert before < after
assert tuple.pack(before) < tuple.pack(after)
```

Et aussi, c’est une operation reversible:

```
from fdb import tuple


expected = (123456789, 3.1415, ("hello", "world"), b'\x13\x37')

assert tuple.unpack(tuple.pack(expected)) == expected
```

L’ordre permet de créer des structures ou abstractions de plus haut niveau.

The End

Avant de commencer, garder en tête que FoundationDB n’est pas très facile à mettre en production, mais c’est facile à tester en dev. Il y a un backend mémoire et un backend ssd. Et un nouveau backend appellé redwood qui va lever certaines des limitations décrites dans la documentation officielle.

2019-06-15 - Functional Database: Versioned Generic Tuple Store

Keywords

- Database
- Knowledge Base
- Reproducible Science
- Scheme programming language
- Version Control System

Summary

Versioning in production systems is a trick everybody knows about whether it is through backup, logging systems and ad-hoc audit trails. It allows to inspect, debug and in worst cases rollback to previous states. There is no need to explain the great importance of versioning in software management as tools like mercurial, git and fossil have shaped modern computing.

Having the power of multiple branch versioning open the door to manyfold applications. It allows to implement a change-request mechanic similar to github's pull requests and gitlab's merge requests in any domains.

The change-request mechanic is explicit about the actual human workflow in entreprise settings in particular when a senior person validates a change by a less senior person.

Versioning tuples in a direct-acyclic-graph make the implementation of such mechanics more systematic and less error prone as the implementation can be shared across various tools and organisations.

Being generic allows downstream applications to fine tune their time-space requirements. By incrementing the number of items in a tuple, it allows to easily represent provenance or licence. Thus, it avoid the need for reification technics as described in Frey:2017 to represent metadata on all tuples.

datae is a software that takes the path of versioning data in a direct-acyclic-graph. It applies the change-request mechanic to cooperation around the making of a knowledge base. It is similar in spirit to wikidata or freebase.

Resource Description Framework (RDF) offers a good canvas for cooperation around open data but there is no solution that is good enough according to Canova:2015. The use of a version control system to store open data is a good thing as it draws a clear path for reproducible science.

In projects like datahub.io or db.nomics, datae aims to replace the use of git.

datae can make practical cooperation around the creation, publication, storage, re-use and maintenance of knowledge base that is possibly bigger than memory.

datae use a novel approach to store tuples that is similar in principle to OSTRICH Ruben:2018 in a key-value store. datae use WiredTiger database storage engine to deliver a pragmatic versatile ACID-compliant versioned generic tuple store.

datae only stores changes between versions. To resolve conflicts, merge commits must copy some changes. datae does not rely on the theory of patches introduced by Darcs Tallinn:2005.

Current status, and plans for the future

This is stil a work-in-progress. You can find the code at source hut.

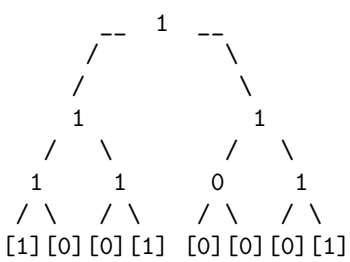
2019-11-12 - FuzzBuzz Hash Algorithm

fuzzbuzz hash is a Locality-sensitive hashing algorithm that has the following properties:

- The length of the longest common prefix of two bit strings is smaller that the length of the longest common prefix of their fuzzbuzz hash.
- The smaller the Hamming distance between two bit strings, the bigger the longest common prefix of their fuzzbuzz hash.

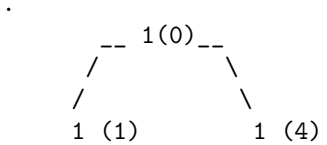
The algorithm rely on a Merkle-tree that use a bitwise `OR` as a hash function and then serialize the tree using a top-down depth-first traversal algorithm.

For instance, let's pick `1001 0001` bit string. We can construct the following Merkle tree:



Then we can serialize it the tree without the leafs, using top-down depth-first traversal.

Here is the same tree, with the index of each bit between parens:



```
   /   \        /   \
  1 (2) 1 (3)  0 (5) 1 (6)
 / \   / \    / \   / \
[1][0][0][1] [0][0][0][1]
```

That result is the following bitstring: `111 1101`

At last, we suffix that bitstring with the original bitstring:

`11 1110 1001 0001`

More experiments in fuzzbuzz repository.

References:

- https://vaibhavsagar.com/blog/2019/09/08/popcount/

2019-11-10 - DIY approximate string matching (fuzzbuzz)

The idea of fuzzbuzz is to do fuzzy search in textual data otherwise said, approximate string matching. This is based on the simhash. Which can be summarized as follow:

```
def features(string):
    """Return a bag of grams of the given STRING."""
    tokens = ['$' + token + '$' for token in string.split()]
    out = Counter()
    for token in tokens:
        iterator = chain(*[ngram(token, n) for n in range(2, len('$CONCEPT$'))])
        for gram in iterator:
            out[hash(gram)] += 1
    return out


def simhash(string):
    """Compute a similary hash called simhash"""
    input = features(string)
    intermediate = [0] * HASH_SIZE
    for feature, count in input.items():
        for index, bit in enumerate(int2bits(feature)):
            intermediate[index] += count if bit == '1' else -count
    # compute simhash
    simhash = ''.join(['1' if v > 0 else '0' for v in intermediate])
    simhash = int(simhash, 2)
    return simhash
```

The astute reader will recognize that `simhash` returns a positive integer based on a bag-of-grams where grams are slices of words between 2 and 7 magic.

The idea is that simhash will capture similarity that exist in small-ish strings:

```
In [1]: import fuzzbuzz

In [2]: fuzzbuzz.hamming2(fuzzbuzz.simhash('obama'), fuzzbuzz.simhash('barack obama'))
Out[2]: 16

In [3]: fuzzbuzz.hamming2(fuzzbuzz.simhash('obama'), fuzzbuzz.simhash('trump'))
Out[3]: 30

In [4]: fuzzbuzz.hamming2(fuzzbuzz.simhash('concept'), fuzzbuzz.simhash('concpet'))
Out[4]: 22

In [5]: fuzzbuzz.hamming2(fuzzbuzz.simhash('concept'), fuzzbuzz.simhash('concept'))
Out[5]: 0

In [6]: fuzzbuzz.hamming2(fuzzbuzz.simhash('concept'), fuzzbuzz.simhash('concept car'))
Out[6]: 11

In [7]: fuzzbuzz.hamming2(fuzzbuzz.simhash('concept'), fuzzbuzz.simhash('quality'))
Out[7]: 30

In [8]: fuzzbuzz.hamming2(fuzzbuzz.simhash('quality assurance'), fuzzbuzz.simhash('quality'
Out[8]: 17
```

That is it gives a clue of how similar two strings are. That said, it requires to compute the Hamming distance of the simhash.

Given a giant set of documents and a new document, figuring which document is the most similar requires to compute the simhash before hand at index time, and then at query time, it requires to compare the simhash of the new input document with the simhash of ALL the known documents the complexity is at least O(n).

In fuzzbuzz, `HASH_SIZE` is not documented magic, but was chosen to be two times bigger than the count of known documents: 2^32 = 4 294 967 296. That is around 4/2 billions documents. That is around 2 billions * 32 / 8 = 8 000 000 bytes otherwise 8GB of memory required just to store the simhashes. That is way too much for my laptop with 12GB of RAM.

It is not possible to pre-compute (aka. index) the Hamming distance of all possible input documents.

What about indexing, in an Ordered Key-Value store, the simhash instead?

That turns out to be is possible.

Given a simhash of 8 bits, one can construct a merkel-tree with binary `OR` operator as a hash function and serialize the resulting tree using a depth-first search back to a bit string called `bbk`.

edit: the merkel-tree hash function is binary `OR`.

`bbk` will have the amazing property that the more similar to two documents are, the longer the common prefix will be.

At query time, the remote distributed dictionary expert system must compute the simhash of the new document, then the `bbk` hash and then search near or query using ranges of `bbk` prefixes of decreasing length.

2019-06-16 - Generic Tuple Store

Keywords

- Database
- Resource Description Framework
- SRFI-168

Kesako a tuple store?

In SRFI-168 a tuple store is defined as follow:

> a tuple store is an ordered set of tuples.

If you prefer, replace "ordered" with "sorted" (I am not sure what is the difference). Anyway, the order is the lexicographic order given by the bytes packing procedure. That can be summarized as: items of the same type are ordered using their natural order. For instance the number `2` is smaller that the number `10` but the string `"2"` is bigger than `"10"`.

The comparison between items of different types is defined but it doesn't matter for what is following.

Why Generic?

As explained in Frey:2017 named graph perform better at metadata representation than other strategies. Similarly, nstore excels at representing metadata about triples or quads. In general, whenever you need to attach some information to all tuples you can add an item to the tuple to do so.

The canonical Resource Description Framework (RDF) naming of tuple items are for triple stores:

```
(subject predicate object)
```

For named graph (ngraph) or quad store you prefix the tuple with `graph`:

```
(graph subject predicate object)
```

If you come from mongodb world, you will have an easier time with the following naming:

```
(collection object-id key value)
```

If you come from the RDBMS world, you will prefer the following naming:

`(table primary-key column-name value)`

It consider that every row as a primiary key, like in google spanner `Corbett:2012` and most relational databases I encountered.

In RDBMS when you need to add information to a row, you add column. That would be equivalent in the RDBMS to RDF mapping to add a **tuple** to every `primary-key` from a `table` with the given `column-name`.

The **strict** equivalence in RDF would be, instead, to add an **item** to every tuple, which is not yet part of the specification.

Based on that reflexion, one might argue that RDBMS is more powerful than RDF or nstore.

n+1 tuple items

To give some food for thought, try to imagine how to represent metadata like:

- provenance / source of row values
- license of row values
- history of row values

Done figuring a good schema for that problems?

This is difficult.

There is some prior art, like the Entity-Attribute-Value model that is used in magento CRM.

That example use of EAV model, a triple store, in magento is interesting. Basically, magento try to be "ready" for every possible schema by choosing the triple representation. The meme is it has too many downsides. I argue that this comes from a time where the problem was different. In particular, hard disk is cheaper nowdays. Also, event-sourcing architecture has proven that it is completly feasible to store a lot of data in a Single-Source-Of-Thruth that can written to very quickly and expose "views" of that data in secondary systems.

You might think, that representing provenance, license for a single value is overkill. But that is problem that wikidata has and almost all data science projects should have the same issue if they consider reproducibility and quality assurance.

Also, regarding keeping track of values history, they are known audit-trails in the wild. They don't allow time-traveling queries. Most of the time, it is not systematic. If you have a new table you have to build a new audit-trail table OR rely on EAV model.

So we are back at triple stores.

What is nice about a triple store, is that they allow to to represent every kind of facts whether it is relational / graphical, tabular or documents. And every fact, has its own row. And if you want you can reify a given triple to add more facts about it. That what `Frey:2017` explains. Most of those technics require to do more **join on every tuple** to be able to query to fetch the particular information (provenance, license, history). nstore avoids those extra joins, as the metadata is stored along the rest of the tuple. ## 2019-06-06 - Je suis une bande éthic à moi tout seul 2018/2019

C'est pas juste que j'ai du cœur à l'ouvrage, sans sens apparat à apparaitre aux après de celles et ceux qui revaient d'une enigme, c'est ma passion.

Ce que j'ai fait comme contribution libre en 2018 / 2019.

Python

hoply

Maintenant hoply (anciennement AjguDB) est une generic tuple store. J'ai completement abandonne mon ancienne abstraction. Je trouve la nouvelle plus versatile mais toujours pas au top. Attendez la suite!

lastho.pe

Il s'agit d'une experience qui utilise ff.js… et une ancienne version de hoply. Autrement dit j'ai pas mis a jour le code et le code a pourri. En gros, au lieu de faire des recherche dans des fenetre separer et d'ouvrir une nouvelle instance fraiche du navigateur. L'utilisateur doit regrouper ses recherche dans une activite.

asyncio-foundation

Comme son nom l'indique cela a un rapport avec asyncio et foundationdb. Voir https://github.com/amirouche/asyncio-foundationdb.

JavaScript

ff.js

J'ai mis au propre mon framework inspire de Scheme. Y a de la documentation. Voir https://github.com/amirouche/ff.js

Scheme

http://scheme-lang.com/cons

C'est un debut de mise a jour du framework Scheme pour faire du frontend qui utilise chibi-scheme compile pour wasm.

datae

C'est la suite de hoply et mon main focus. Voir la doc: https://github.com/awesome-data-distribution/datae

Conlusion

Et d'autres pas interessant ;-)

2019-06-02 - New domain

The new domain is hyper.dev! ## 2019-06-12 - On the road to a multi-model database

To get a good grasp of what graphdb were, I mocked a graph database using Python 2.6's anydbm in a project dubbed ajgu (2010).

I created Graphiti (2012) which was an Object-Relational Model framework for representing graphical data in your favorite Object-Oriented Programming language, namely Python. I promise at some point it was working. And I promise, nowdays I can code better. That project teached the following lesson: embedding the full query language of graphical databases inside Python without relying on string interpolation is not possible.

Given it was not possible to embed nicely another Turing Complete language inside Python, I tried to embed Python inside Tinkerpop in GraphitiDB (2013). I promise it was working at some point. To me, that was mostly a failure because it was not really possible to send Python queries directly from Python code without relying on string interpolation.

also tried to bind Tinkerpop's Blueprints (2013) inside Python. It was very slow.

And I figured something else. Using Tinkerpop, unlike with PostgreSQL, I had to rely on another database to do full-text search and geospatial queries which would break the ACID semantic I was so attached to.

I continued working on ajgu all along. At some point I renamed it AjguDB (2015). This is the first version that rely on Entity-Attribute-Value Model abstraction. That inspired the post: Do-It-Yourself: A Graph Database In Python.

At this point, I knew that building a **database** in Python was bad idea [0]. So, I pivoted into an "graph exploration tool" but never really executed that idea.

Along the way, I got lost a little regarding my goal. My initial idea was to create a general purpose database that support transactions all around.

For other reasons I learned Scheme programming language. GNU Guile did not have a Global Interpreter Lock and performance were better than CPython. I continued experimenting with ordered key-value stores. I have written several posts regarding this work:

- Somewhat Relational Database Library Using Wiredtiger (2016)
- Getting started with guile UAV database (2016)
- Getting started with guile-wiredtiger (2016)
- Do It Yourself: a search engine in Scheme Guile (2016)
- A Graph-Based Movie Recommender Engine Using Guile Scheme (2016)

Since, the beginning I always wrapped inside classes and Python objects the abstractions I was building. Somekind of Object-Oriented Programming made things, very difficult to interop and compose abstractions. To be honest, even the Scheme functional code was not easy to compose.

Even the successor of ajgu, namely hoply is still broken in this regard.

That is when I FoundationDB was open-sourced that I figured that the ordered key-value store doesn't have to be hidden. That is the underlying abstraction can leak on purpose because it helps to build higher level abstractions, compose them and generaly reach somekind of fractal architecture.

I have put in motion that idea in SRFI-167 with an exampe abstraction SRFI-168 [1]. That is, abstractions on top of SRFI-167 shall not hide the ordered key-value store and accept `prefix` to allow to hook it somewhere. Also, it should be possible to nest abstractions to be fractal. This is not possible in the sample implementation because the packing procedure don't support nested datastructures, as of yet.

So, if you wanted you could build a triple store on top a triple store. That is already a thing and some do n-tuples inside triple store. Based, on my analysis this is not great in terms of performance.

Or you could star datae.

[0] edgedb.

[1] This is still a work-in-progress and we very recently released another draft. Chime in! ## 2019-12-10 - nomunofu

nomunofu is database server written in GNU Guile that is powered by WiredTiger ordered key-value store, based on SRFI-167 and SRFI-168.

It allows to store and query triples. The goal is to make it much easier, definitely faster to query as much as possible tuples of three items. To achieve that goal, the server part of the database is made very simple, and it only knows how to do pattern matching. Also, it is possible to swap the storage engine to something that is horizontally scalable and resilient (read: foundationdb).

The idea is to have a thin server and thick client, in order to offload the database server(s) from heavy computations.

I pushed portable binaries built with gnu guix for amd64 with a small database file. You can download it with the following command:

```
$ wget https://hyper.dev/nomunofu-v0.1.3.tar.gz
```

The uncompressed directory is 7GB.

Once you have downloaded the tarball, you can do the following cli dance to run the database server:

```
$ tar xf nomunofu-v0.1.3.tar.gz && cd nomunofu && ./nomunofu serve 8080
```

The database will be available on port 8080. Then you can use the python client to do queries.

Here is example run on a subset of wikidata, that queries for:

> instance of (P31) government (Q3624078)

The python code looks like:

```
In [1]: from nomunofu import Nomunofu
In [2]: from nomunofu import var
In [3]: nomunofu = Nomunofu('http://localhost:8080');
In [4]: nomunofu.query(
(var('uid'),
 'http://www.wikidata.org/prop/direct/P31',
 'http://www.wikidata.org/entity/Q3624078'),
(var('uid'),
 'http://www.w3.org/2000/01/rdf-schema#label',
 var('label')))

Out[4]:
[{'uid': 'http://www.wikidata.org/entity/Q31',
'label': 'Belgium'},
 {'uid': 'http://www.wikidata.org/entity/Q183',
'label': 'Germany'},
 {'uid': 'http://www.wikidata.org/entity/Q148',
'label': 'China'},
 {'uid': 'http://www.wikidata.org/entity/Q148',
'label': "People's Republic of China"},
 {'uid': 'http://www.wikidata.org/entity/Q801',
'label': 'Israel'},
 {'uid': 'http://www.wikidata.org/entity/Q45',
'label': 'Portugal'},
 {'uid': 'http://www.wikidata.org/entity/Q155',
'label': 'Brazil'},
 {'uid': 'http://www.wikidata.org/entity/Q916',
'label': 'Angola'},
 {'uid': 'http://www.wikidata.org/entity/Q233',
'label': 'Malta'},
 {'uid': 'http://www.wikidata.org/entity/Q878',
'label': 'United Arab Emirates'},
 {'uid': 'http://www.wikidata.org/entity/Q686',
```

```
'label': 'Vanuatu'},
 {'uid': 'http://www.wikidata.org/entity/Q869',
'label': 'Thailand'},
 {'uid': 'http://www.wikidata.org/entity/Q863',
'label': 'Tajikistan'},
 {'uid': 'http://www.wikidata.org/entity/Q1049',
'label': 'Sudan'},
 {'uid': 'http://www.wikidata.org/entity/Q1044',
'label': 'Sierra Leone'},
 {'uid': 'http://www.wikidata.org/entity/Q912',
'label': 'Mali'},
 {'uid': 'http://www.wikidata.org/entity/Q819',
'label': 'Laos'},
 {'uid': 'http://www.wikidata.org/entity/Q298',
'label': 'Chile'},
 {'uid': 'http://www.wikidata.org/entity/Q398',
'label': 'Bahrain'},
 {'uid': 'http://www.wikidata.org/entity/Q12560',
'label': 'Ottoman Empire'}]
```

As of right now there is less than 10 000 000 triples that were imported. Blank nodes are included, and only English labels are imported.

You can grab the source code with the following command:

```
$ git clone https://github.com/amirouche/nomunofu
```

I hope you have a good day! ## 2019-11-15 - On the road to guile-babelia

Yeah, I am back in GNU Guile land. With yet another good name for the very same project that boils down to fight boredom, learn new skills and bring back the power to the computers of every lambda users.

Quick flashback: I started with a minimal bulletin board more than 15 years ago. It was all good, and some success, lost all the backup and to repeat the same mistake: poor software pratices. Nevermind, it was mostly copy-pasted LAMP stack code written on Windows 2000. The lost backups will haunt me back for sure. But what does not?! Then I tried with Python, Django, Flask, MongoDB, Neo4J, PosgreSQL, Whoosh, skipped ElasticSearch, read a few hundreds of lines of Lucene documentation and code. Dozens of science papers. Enjoy Ordered Key-Value Store all the way down. Re-discovered fuzzbuzz. Learned Scheme. gnunet is making progress. Dozen of prototypes on various subjects. I have a very good idea of the graphical user interface in terms of web stack (to get the project started). Various good good discussions happening around rdf, AI-KR, GOFAI. And a glimpse into proprietary commercial system that have great success.

Eventually, more or less, a roadmap for the next five decaces.

I have two immediate pain points:

- gmail. I hate that interface. I much prefer inbox. I could upgrade ff.js framework. But meh. And Gambit is missing some library. I tried to help but then lost interest, temporarly, because the next point is more important.
- Search engine overall user experience: simply said I want to own all the data I search and that should not be scatered in various privative or hidden sqlite files! I want an IDE of my research adventures. I do not want to do 3 or more clicks and fail to retrieve the paper I read 3 days or 3 years ago that is now not publicly accessible. I want to ban medium.com from my search results. But still, unhide it in case of boredom or emergency.

You prolly guessed, that I want a personal search engine.

Here is in no particular order pieces that I think are missing in GNU Guile:

- headless firefox driver for crawling Single-Page-Application,
- a smart crawler (see what use Common Crawl),
- news.ycombinator.com (meh) html snapshot,
- stackoverflow html snapshot,
- quora snapshot? not sure it is possible, it is a walled garden,
- wet/warc file parser,
- Common Crawl support,
- Somekind of wet/warc file consumer that will (only) build page/domain ranks.
- A fork of Grammar Link in pure Guile (with minisat bindings) to… check pages grammar.
- a word2vec and paragraph2vec similar to Gensim and Spacy that would allow to finger print the subject of domains / webpages against wikipedia vital articles hierarchy.

Sure all this stuff can be put together in a few days by a junior architect using a franken-assemblage, to quote myself:

> We could argue indefinitly that one or more of those requirements are unnecessary, overkill and YAGNI. We could argue that by relaxing a few of the requirements, a particular software or set of softwares can come close. We could argue endlessly that building yet-another-database [or search engine] is NIH, wheel re-invention that curse the software industry with fragmentation and fatigue. We could invoke UNIX philosophy, entreprise software architectures, experiences, know-how, failed patterns, decades of good services and big communities.

Blue pill: To use a vocabulary that has a lot mindshare: that is my product, isn't?

Red pill:

> "Plans are only good intentions unless they immediately degenerate into hard work"
>
> Peter Drucker

2019-06-30 - Past, Present, and Future

> Plans are worthless, but planning is everything. There is a very great distinction because when you are planning for an emergency you must start with this one thing: the very definition of "emergency" is that it is unexpected, therefore it is not going to happen the way you are planning.

Past

My main focus was working on the sample implementation for SRFI-167 and SRFI-168 along the continued work on a real implementation on top of WiredTiger using (Chez Akku) Arew Scheme. I managed to get together a release dubbed 0.1.1, that you can try using the following command:

> git clone https://github.com/scheme-live/srfi-167-and-168-tutorial

The standardization process is kicking and more people got involved by sharing wishes, bug reports and in general valuable positive and negative feedbacks. My talk on this subject to Scheme Workshop was accepted. I would have liked write and submit a full paper about it, I mostly failed. I started writing a tutorial (book?) about the Ordered Key-Value Store that is called Around Multi-Model Database.

I drew a plan in the sand of github about datae.

I put together some ideas about peer-to-peer networks and functional package manager.

Also, schemedoc launched Awesome Scheme.

I got involved in some W3C mailling list related to RDF and AI.

At last, I got some energy to read things and listen to Noam Chomsky.

I found especially interesting the text entitled: Design Principles Behind Smalltalk.

Here is a first quote about their "scientifical" approach to design systems:

> Our work has followed a two- to four-year cycle that can be seen to parallel the scientific method:
>
> - Build an application program within the current system (make an observation)
> - Based on that experience, redesign the language (formulate a theory)

- Build a new system based on the new design (make a prediction that can be tested)

The Smalltalk-80 system marks our fifth time through this cycle.

Here is another about the relation between tools, individual accomplishments and how to achieve them:

> I'll start with a principle that is more social than technical and that is largely responsible for the particular bias of the Smalltalk project:
>
> Personal Mastery: If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.
>
> The point here is that the human potential manifests itself in individuals. To realize this potential, we must provide a medium that can be mastered by a single individual. Any barrier that exists between the user and some part of the system will eventually be a barrier to creative expression. Any part of the system that cannot be changed or that is not sufficiently general is a likely source of impediment. If one part of the system works differently from all the rest, that part will require additional effort to control. Such an added burden may detract from the final result and will inhibit future endeavors in that area. We can thus infer a general principle of design:
>
> Good Design: A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.

Seems like the spirit of Scheme or worrydream.com.

I also liked the lisp os text. Here is the part that echoed the most with my work:

> Object store based on tags
>
> Instead of a hierarchical file system, we propose an object store which can contain any objects. If a file (i.e. a sequence of bytes) is desired, it would be stored as an array of bytes.
>
> Instead of organizing the objects into a hierarchy, objects in the store can optionally be associated with an arbitrary number of tags. These tags are key/value pairs, such as for example the date of creation of the archive entry, the creator (a user) of the archive entry, and the access permissions for the entry. Notice that tags are not properties of the objects themselves, but only of the archive entry that allows an object to be accessed. Some tags might be derived from the contents of the object being stored such as the sender or the date of an email message. It should be possible to accomplish most searches of the store without accessing the objects themselves, but only the

> tags. Occasionally, contents must be accessed such as when a raw search of the contents of a text is wanted.
>
> It is sometimes desirable to group related objects together as with directories of current operating systems. Should a user want such a group, it would simply be another object (say instances of the class directory) in the store. Users who can not adapt to a non-hierarchical organization can even store such directories as one of the objects inside another directory.

While the idea of building a Scheme operating system is beautiful. I don't have enough time for it. Much inspiration can be taken from existing projects like GNU Guix, Mezzano or Unikernels…

Another interesting read is: Accelerating Science: A Computing Research Agenda:

> Accelerating Science: The Value Proposition
>
> Cognitive tools for acclerating science could lead to dramatic increases in scientific productivity by increasing efficiency of the key steps in scientific process, and in the quality of science that is carried out (by reducing error, enhancing reproducibility), allow scientific treatment of topics that were previously impossible to address, and enable new modes of discovery that leverage large amounts of data, knowledge, and automated inference.

… and allow effective learning.

Present and Future

The sample implementation for SRFI-167 in particular needs more love. It happens that I need it, for some future projects. Among others I use it in emacs-like editor and I might rely on it in the functional package manager. I have still things to figure. It seems like I am trying to shoehorn it, especially in the case of the functional package manager. In the case of the editor, it is less likely to be a mistake because indeed it is helpful and implements separation of concerns like Model-View-Controler. Having the same data-structure everywhere is helpful and it is more powerful than a hash-table or struct as it allow to easily do introspection to ease debugging.

While I was excited by datae as a demonstration of change-request mechanic over structured data that is bigger than memory. I was thinking that it might lead to making a living out-of-it with grants, community support or consulting. Getting together that software with 100% support of SPARQL is not impossible. The question is do I really want to invest my time in this particular project? At some point, "maybe" is not anymore an acceptable answer. In the meantime, I will just put it on "standby" mode.

I am discussing a possible relicencing of Arew into something dubbed business-friendly most likely Apache 2.0. This will lead to a split of the project into two components. The first will remain in Arew and will be dedicated to R7RS support and moar. The second repository will be prolly called babelia and will be licensed in a fork of the Affero GPL licence that is more humane, more sensible to the challenges faced by the anthorposcene.

I got a better idea about my Personal Knowledge Base project or if you prefer my Personal Assistant or Research Assistant. To quote Ronald J. Brachman:

> Can we have realistically useful Knowledge Base that is designed in the absence of specific intended applications?

Otherwise said, I need an application of the application (!) to be able to dogfood the idea. I think studying Artificial Intelligence and its history is a good subject of study, in particular I need to learn more about non-monotoic logic and ordinal number systems.

2019-07-07 - SPARQL to Scheme Generic Tuple Store (nstore)

SPARQL is Resource Description Framework (RDF) query langauge. It allows to query triple stores and quad stores (called named graph).

Scheme Generic Tuple Store is work-in-progress Scheme Request For Implementation (SRFI) dubbed 168. It embeds a triple store or quad store or set of tuples of n items in Scheme programming language.

It rely on a pattern matching query semantic similar to SPARQL upon which minikanren logic language can be bound.

In the following, I will show how to translate some SPARQL queries into Scheme code.

In what follow we consider the following two stores. A triple store:

```
(define triplesotre (nstore #vu8(00) '(subject predicate object)))
```

And a quad store:

```
(define quadstore (nstore (#vu8 01) '(graph subject predicate object)))
```

And `tx` is a transaction object.

Data Types

The supported data types are composition of the follow basic Scheme types:

- boolean
- numbers (big integers, float and double)
- symbol
- string

- bytevector
- list (soon)
- vector (soon)

There is no specific handling of date time objects or URIs.

Simple query

SPARQL

```
SELECT ?title
WHERE
{
  <http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> ?title .
}
```

Scheme

```
(nstore-query
  (nstore-from tx triplestore
               (list 'http://example.org/book/book1
                     'http://purl.org/dc/elements/1.1/title
                     (nstore-var 'title))))
```

SPARQL

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

Scheme

```
(nstore-query (nstore-from tx triplestore
                           (list (nstore-var 'graph)
                                 'http://xmlns.com/foaf/0.1/name
                                 (nstore-var 'name)))
              (nstore-where tx triplestore
                            (list (nstore-var 'graph)
                                  'http://xmlns.com/foaf/0.1/mbox
                                  (nstore-var 'mbox))))
```

If you prefer, you can define a procedure in Scheme:

```
(define (foaf symbol)
  (string->symbol (string-append "http://xmlns.com/foaf/0.1/" (symbol->string symbol))))
```

And then the query becomes:

```
(nstore-query (nstore-from tx triplestore
                           (list (nstore-var 'graph)
                                 (foaf 'name)
                                 (nstore-var 'name)))
              (nstore-where tx triplestore
                            (list (nstore-var 'graph)
                                  (foaf 'mbox)
                                  (nstore-var 'mbox))))
```

FILTER

SPARQL

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { ?x dc:title ?title
          FILTER regex(?title, "^SPARQL")
        }
```

Scheme

```
(gfilter (lambda (binding) (string-prefix? "SPARQL" (hashmap-ref binding 'title)))
  (nstore-query (nstore-from tx triplestore
                             (list (nstore-var 'x)
                                   'http://purl.org/dc/elements/1.1/title
                                   (nstore-var 'title)))))
```

SPARQL

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title ?price
WHERE   { ?x ns:price ?price .
          FILTER (?price < 30.5)
          ?x dc:title ?title . }
```

Scheme

```
(gfilter (lambda (binding) (< (hashmap-ref binding 'price) 30.5))
  (nstore-query (nstore-from tx triplestore
                             (list (nstore-var 'x)
                                   'http://example.org/ns#price
                                   (nstore-var 'price)))
                (nstore-where tx triplestore
                              (list (nstore-var 'x)
                                    'http://purl.org/dc/elements/1.1/title
                                    (nstore-var 'title)))))
```

`OPTIONAL`

SPARQL

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }
```

Scheme

JOKER!

`UNION`

SPARQL

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?title
WHERE  { { ?book dc10:title  ?title } UNION { ?book dc11:title  ?title } }
```

Scheme

```
(gappend
  (nstore-select (nstore-from tx triplestore
                              (list (nstore-var 'book)
                                    'http://purl.org/dc/elements/1.0/title
                                    (nstore-var 'title))))
  (nstore-select (nstore-from tx triplestore
                              (list (nstore-var 'book)
                                    'http://purl.org/dc/elements/1.1/title
                                    (nstore-var 'title)))))
```

You are not obliged to copy-paste and you can factor queries…

`FILTER NOT EXISTS`

JOKER

`FILTER EXISTS`

JOKER

`MINUS`

JOKER

BIND

Use `gmap`

`GROUP BY`

JOKER

`HAVING`

Use `gfilter`

Sub-queries

Trivial.

Graph queries

Trivial.

`ORDER BY`

JOKER

`DISTINCT`

JOKER

`LIMIT ... OFFSET ...`

Use `gtake` and `gdrop`.

2019-10-04 - State of Scheme in the Browser

I made some progress around my experiences in the browser, this lead me to write a Scheme-to-JavaScript compiler that is incomplete but can do some stuff.

Chicken

I still did not try Chicken's Spock egg.

ref: http://wiki.call-cc.org/eggref/4/spock

BiwaScheme

This is an interpreter written in JavaScript that doesn't support tail-call optimization.

- demo: https://hyperdev.fr/defunct-forward.scm/
- repo: https://github.com/amirouche/defunct-forward.scm

Gambit JavaScript backend

It is still a work in progress. ~~It prolly support tail-call optimization but requires some work. With the build I tested, tail-call factorial gives an error.~~

edit: It works well actually. The problem was on my side.

- demo: https://scheme-live.github.io/scheme-fuss/
- repo: https://github.com/scheme-live/scheme-fuss

Schism

This is the most promising stuff. It is a self-hosted Scheme-to-WebAssembly (wasm) compiler. The two things that are missing are some kind of call/cc support and I don't know how to yield the control back to JavaScript.

- repo: https://github.com/google/schism/

Chibi Scheme WebAssembly build

It works most of the time on Firefox but it (used to) crash on Windows Chrome.

- repo: https://github.com/scheme-live/ff.scm
- demo: https://scheme-live.github.io/ff.scm/

Ruse Scheme

This is my work-in-progress compiler that targets JavaScript and not WebAssembly (yet).

- repo: https://github.com/scheme-live/ruse-scheme/
- demo: https://scheme-live.github.io/ruse-scheme/demo/counter/

2020-08-19 - 1001 LOLS

1001 lines or less of Scheme project ideas:

1. fauxtexte
2. zombie: static blog generator
3. biolog (ghost clone)
4. kvcli
5. linkify fdb
6. linkify lite
7. linkify fast
8. bbkh fdb
9. bbkh lite
10. bbkh fast
11. fstore fdb (fuzzystore)
12. fstore lite
13. fstore fast

14. processing.scm
15. text editor (zk)
16. search engine fdb
17. search engine lite
18. search engine fast
19. xzstore fdb (xz-ordering)
20. xzstore lite
21. xzstore fast
22. nstore fdb
23. nstore lite
24. nstore fast
25. vnstore fdb
26. vnstore lite
27. vnstore fast
28. docstore fdb
29. docstore lite
30. docstore fast
31. estore fdb (eavt / datomic clone)
32. estore lite
33. estore fast
34. pubsubbox
35. ohmypalette
36. mined (demineur clone)
37. blockchain (tetris clone)
38. column drop (sega columns clone)
39. convo (isso clone)
40. datex (superset clone)
41. ftyper (ztype clone)
42. squared (magical drop clone)
43. tictac web analytics
44. http log parsers
45. fridge pastebin
46. shorturl
47. status page
48. incoming(email & feed reader)
49. gstore fdb graph store
50. gstore lite
51. gstore fast
52. html2md
53. md2html
54. musicone (bandcamp clone, for single user)
55. hivimix (collaborative video dj)
56. lasthope (meta search engine + history + bookmark)
57. sensimark (requires word2vec, gensim and scikit-learn)
58. c3b2 (peer-to-peer bulletin board)
59. c3qa (peer-to-peer question-answering site)

60. around news (geonews)
61. nightwatch (newspaper)
62. zettlekasten
63. quicoeur (lobsters clone)
64. rstore fdb (rankedset)
65. rstore lite
66. rstore fast
67. space exploration (clicker game)
68. lux grammar
69. simhash lite
70. simhash fast
71. simhash fdb
72. easy-conceptnet (with shortest path, fuzzy search)
73. easy-words (wordnet explorer)
74. html2one https://github.com/Y2Z/monolith
75. nuw (terminal web browser)
76. sosos (stackoverflow sos)
77. html2pdf
78. tootui (mastodon terminal client)
79. html2png
80. html2epub
81. opml2html
82. ccse (commoncrawl search engine)
83. trivial (slack alternative)
84. todomvc (todo list)
85. realworld (medium clone)
86. azka (peer-to-peer ide)
87. babelia (federated search engine)
88. razlebol (rollbar alternative)
89. ruse (scheme to javascript)
90. xinos (nixos clone based on musl)
91. source (git / fossil clone)
92. source hub (onedev / source hut clone)
93. zombie brain wash (game)
94. sql2http
95. tuttyfruity (diamonds clone)
96. sqlrepl
97. sql2graphql
98. jungle (notion.so clone)
99. spaceop (shoot-them-up vertical scrolling)
100. musicism (browser-based music player)
101. voxelart (3d pixel art)

2020-08-19 - Arew Scheme

I started working a shim for Chez Scheme to support R7RS.

Want to try?

On Debian / Ubuntu, you can do the following:

```
# git clone https://github.com/arew-scheme/arew-scheme/ arew
# cd arew
# make init
# ./venv arew eval example.scm hello world
```

2020-08-24 - Common Crawl Search Engine

- ccse will search for keywords in Common Crawl datasets
- 188 + 291 (aho-corasick algorithm) = 479 lines of code.

Getting started

```
# wget https://commoncrawl.s3.amazonaws.com/crawl-data/CC-MAIN-2018-26/segments/15292678597
...
# gunzip CC-MAIN-20180618105733-20180618125538-00026.warc.wet.gz
...
# ccse CC-MAIN-20180618105733-20180618125538-00026.warc.wet keyword search engine
...
# time ./ccse.scm  CC-MAIN-20180618105733-20180618125538-00026.warc.wet keyword search engi

real    0m 12.67s
user    0m 12.54s
sys 0m 0.10s
```

ChangeLog

- 2020/08/24: v0.1.0 - initial release
  - alpine
  - debian buster

2020-10-12 - Do It Yourself: A Search Engine

Prelude

I read more often than ever that people are looking for ways to build their own search engines.

Even if more on more "advanced" features are integrated into search engines. They are mostly based on human grunt work. Semantic search engine, whatever "semantic" does mean for you, is in fact merely a couple, not more than a dozen, set of tricks. I like to say, much of Google's search engine is good old human labor. If you still doubt it, here is again: Google results are not only biased, also they are editorialized. Whether algorithms, and their bugs, party is irrelevant.

My point is: it is human made. Not some necessarly advanced alien tool.

The only thing preventing you to have your own search engine is there is no readily available software, why? because there is no cheap hardware.

This might sound like a crazy idea five or ten years ago, but with the advent of AMD threadripper ie. cost gravity at play getting together a personal search engine is, if not a necessity, at least a possibility.

The most common complain I read about Google is that it yields irrelevant text "that do no even contain my search terms". That is not a bug, that is a feature! It seems to me the subtext is that you can not easily customize Google or whatever search algorithm since it is privative. Even retrieving Google search results for further processing if not forbidden, is at least difficult.

Getting started

Let's start with the beginning: what is a search engine? A search engine is software that will **crawl** the Internet, **index** ie. store the text in a particular format, and that users can **query** and receive in return the most relevant text.

Let's dive into what "relevant text" means. What is relevant text?

1. A text that contains the search term in my query
2. A text that has the same topic as my query
3. A text that gives an answer to my query

The good answer is "it depends". That's why queries have grown from keywords match like a book index, to boolean queries e.g. `"Apple" -bible`, until so called semantic search, which boils down to consider one-way or two-way synonyms and other lexical features.

So far, I failed to build a crawler that works. Also much of the text I am looking for is in wikipedia or StackOverflow for which they are flat releases which are much more easy to get started than putting together your own crawler. Still, they are some crawler around, so you can use that or learn from it. I will not dive into the crawler part because it still hurts when I think about `robots.txt`, throttling, text encoding etc… booooh!

So, we will imagine that you have a dataset of plain text, for instance wikipedia vital articles. It helps if you know the content of the dataset. Querying random news article is not very easy to grasp because you have to read (!) the text to figure when and how they are relevant.

Before querying, you need to store the text, but to know **how** to store the text you need to know which query feature you want. To get started I will only consider positive keywords and negative keywords like `apple -bible`. So, we need to figure which find out which text contains the word `apple` but not the word `bible`. Looking up everything that does not contain `bible` is difficult, you would need to **scan** the whole database to what are those document. Instead

we will look for documents that contains the word `apple`. So the following document contains the word we are looking for:

```
(define doc1 "apple is looking for a search engine.")
```

That is the moment where the most advanced technology of our current century makes it appearance: the inverted index. **What is an inverted index? It reverses the relationship between the document and the word**. Instead of saying "this document contains the word apple" it says "apple is contained in this document". So we might have a procedure that returns the document identifier that contains `apple`, like:

```
(assert (contains? (inverted-index-lookup "apple") 1))
```

`inverted-index-lookup` returns a list, and that list contains the identifier `1` of the first document. That list might be big. So **you want to consider the least common word from the query**. I call that step *candidates selection*. Also, you might want to convert the positive terms into lemmas or stems to go toward semantic search, which will mean you need to store lemmas or stems at index time.

Anyway, the next step, given the list of documents that contain the least common word or term or lemma or stem, is to **filter and score** according to the full query. In the above case, that is remove the documents that contains `bible`. You can do that step serially, and it will necessarly take time. The trick is to use `for-each-par-map`. That is a cousin of map-reduce that execute the map procedure in parallel.

For instance something like:

```
(let ((hits '()))
  (for-each-par-map
    (lambda (uid score) (when score (set! hits (take 10 (sort (cons (cons uid score) hits))
    score (inverted-index-lookup "apple"))
  hits)
```

The score function is interesting. I think going the aho-corasick with a FSM is the best route because it is easy to implement proximity scoring, "phrase matching", or really anything I can think of.

That last paragraph is really the most important part of this post.

Conclusion

There is a gigantic leap that is going to happen in search because of hardware availability, and free software with readable source ie. the only thing that makes human progress possible: knowledge sharing.

Malfunction! Need input!

- https://en.wikipedia.org/wiki/Tf%E2%80%93idf

- https://en.wikipedia.org/wiki/Precision_and_recall
- https://hal.archives-ouvertes.fr/hal-01730479/document
- https://wiki.nikitavoloboev.xyz/web/search-engines
- https://github.com/amirouche/babelia

2020-08-19 - fauxtexte

- fauxtexte is a *fake text generator* based on markov chains.
- 153 lines of code.

Getting started

Here is the usage help text:

```
# fauxtexte
Usage:

  learn INPUT OUTPUT
  generate INPUT [SOME FUN ...]
```

- `fauxtexte learn INPUT OUTPUT` will learn the vocabulary. `INPUT` is supposed to be a text file with a single phrase per line. `OUTPUT` will be the name of the generated model. You can call multiple time `learn` with different `INPUT` but the same `OUTPUT`.
- `fauxtexte generate INPUT [SOME FUN ...]` will generate a single sentence using the `OUTPUT` of the above command. That is `INPUT` in this command is the name you passed as `OUTPUT` in the above!

Here is an example run:

```
# ./fauxtexte learn data/scheme-2020-04.txt model.fauxtexte
done!
# ./fauxtexte generate model.fauxtexte % scheme is
% scheme is pascal and pascal is fortran
# ./fauxtexte generate model.fauxtexte % scheme is
% scheme is fascinating
# ./fauxtexte generate model.fauxtexte % scheme is
% scheme is appropriate in your redefinitions
```

The `%` in the `fauxtexte generate` command will instruct the program to try to start with the beginning of known sentences, otherwise it can pick any word it learned from anywhere in a sentence.

ChangeLog

- 2020/08/19: v0.0.0 - initial release
  - alpine
  - debian buster ## 2020-10-12 - kvcli

- Tiny command line utility to store strings in a key-value store.
- 40 lines of code

2020-01-22 - Scheme fatigue

Scheme fatigue has two sides. In the following I try the describe my views about Scheme programming language and also poke at a few other topics.

It might not be very well structured.

Fragmentation

As far as I can tell, Scheme as a programming language is easy to the mind, there is a limited set of features (albeit sometime powerful ones) to grasp before being productive. Scheme is easy to maintain in the sense that when you come back to a piece of code (that is tested) months after you last checked it, it is possible to make sense of it, but also easy to rework or improve it. At least, Chez Scheme has profiling, a step debugger, with correct source file informations and is efficient. Did I mention it was fast?

Scheme has a healthy ecosystem of computer scientist, coders, standards and implementations.

Many say, there is too many implementations, and that it is bad luck for the programming language as a whole. I would add that there is many **maintained** implementations. I disagree with the fact that it is bad luck. Scheme programming language describes a Turing-complete programming language that is easy, fun, documented, and grateful to implement in an interpreter or a compiler. That offers a clear and well documented path from apprentice to master.

Too much choice, kills choice you might say; one side of the coin of the Scheme world: which Scheme implementation should I choose?

The official answer is that you should choose: none. It makes sense versus vendor lock-in. The Scheme Reports steering committee agreed on the fact that Scheme code ought to be be portable across implementations:

> The purpose of this work is to facilitate sharing of Scheme code. One goal is to be able to reuse code written in one conforming implementation in another conforming implementation with as little change as possible. Another goal is for users of this work to be able to understand each other's code based on a shared and unambiguous interpretation of its meaning.
>
> Charter for [R7RS] working group 2

Based on experiences, both R6RS and R7RS provide a way to create portable code. It is already possible to write portable code across Scheme implementations. The situation can only improve.

The standards help describe a path for portability, is a shared space to exchange ideas, good practices, innovations, and nurture emulation.

Maybe they were some disagreements about wording, philosophy and how big or small it ought to be. That is somewhat political. At the end of the day, each Scheme implementation retains its defining characteristics. That is helpful because it allows the split (or share) the work to explore new ideas and re-explore old ideas. It is also in the interest of commercial efforts because it avoids vendor lock-in.

Fragmentation is good.

Do It Yourself

The other side of the Scheme coin is the "Do It Yourself" philosophy.

It stems among other things from the fact that it used to be mostly a teaching material.

There is the idea that minimalism equals small. On this topic I like to quote:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

One might say that R5RS amd R7RS-small are good enough and that schemes' efforts should be focused on improving and refining the ideas of small but powerful minimalism. The above quote goes to the point, minimalism is defined as the result of the work of taking away spurious features. If, to get started you have an empty set, there is nothing to remove, then there can be no notion of minimalism. I entertain the idea that, in this regard, R6RS and the larger R7RS will allow to create a set of common idioms upon which Scheme will continue its true minimalism seeking adventure. Prolly, R7RS will not be the end of minimalism, an ideal. Maybe there will be things to take away from it, but as of yet, since it is not finished and because there is still room for experiments one can not definitly rule that R7RS-large is not Scheme spirit. For instance, I think, Scheme object type called `port` are misleading, do not promote good programming practices, clutter the specification and the implementations. I made recently the discovery that one can rely on procedures to mimic the behavior of ports in backward compatible way while re-using existing code and idioms. That is, even in R5RS, there is, according to me, things to take away. Without R7RS generators and accumulators, I would not be able to think that ports are a spurious abstraction coming from the past, haunting new students and seasoned engineers alike. The idea behind generators is well understood and self-contained to the extent that it does not break the small language kind-of idea and keeps around the DIY philosophy that is strong within the Scheme community.

There is the widespread meme named the Lisp Curse along the lines of:

> Problems that yields technical issues in other programming languages; with Scheme programming language, they yield social issues.

In particular, I quote the following:

> Every project has friction between members, disagreements, conflicts over style and philosophy. **These social problems are counteracted by the fact that no large project can be accomplished otherwise.** "We must all hang together, or we will all hang separately." But the expressiveness of Lisp makes this countervailing force much weaker; one can always start one's own project. Thus, individual hackers decide that the trouble isn't worth it. So they either quit the project, or don't join the project to begin with. This is the Lisp Curse.

That means, that there is no problem to write the code solving any problem, the issue is to **agree on the Good Thing**. In fact, we do not **need** to agree on everything. That is the point of Scheme.

The Lisp Curse essay, try to explain that it is way too easy to do a fork. I argue that we should let it be so. **I argue we should make it easier to fork**. Meanwhile we should *individually* strive for better softwares. Better software should include rationales, tests, documentations. And credits to be able to understand the lineage of the idea, and question whether the premise that lead to a given solution do still make sense. And, only later, maybe at some point, some guarantees like backward compatibility, deprecation policy, upgrade path and long time-support story. To put emulation, innovation and ideas melting at the forefront of humans goals.

I put long-time support at the end because the Good Thing is to be able to handle things on your own. Do it yourself. That is the most important idea that underlies the free and open source movement. That is what made the hop from pre-millenium based on cisor, paper and fossil fuel to current century possible.

"Lisp Curse" inflict upon the idea that lisp will never bring back the good old days that are before the AI-winter. Time traveling is not possible, or maybe there is, but that does not matter. The thing that matters is lisps have overcome their past daemons related to performance and efficiency. Actually, people (like me) have reported that their Common Lisp or Scheme code to be faster than the C or C++ equivalent. Also, lisp, in particular Scheme has improved the maintainability of Scheme e.g. the advent of hygenic macros, make the code much more readable, and more future proof. It is a complex, maybe complicated idiom, but a powerful one.

A consequence of the AI-winter is that lisps have, to some extent, lost mainstream momentum. Yeah, there was winter, but now it is spring.

People cooperate toward a common goal, in distributed, non-coordinated way, even in evil environments with non-trusted peers most of the time, if not, all

the time e.g. human race survival. That is, the bazaar is the Real Thing. Tho, I don't want to down play the role of the cathedral: the bazaar is made of many cathedrals, some times one-man lone-wolf hacker endeavors, some times the craft feat of many dedicated engineers.

I want to stress the fact that Scheme needs to further embrace, extend and encourage more the distributed non-coordinated cooperation model.

It is a fact that on the topic of cooperation, schemers are not at rest. There is the Scheme Request For Implementations and RnRS standards. More familiar to pythonistas and javascripters, more in the spirit of the bazaar, there is so many scheme package managers.

At this point, I think we agree that the lisp curse is not that bad, being able to fork is healthy! In many ways, Scheme and Common Lisp projects by themselfs contradict the idea that it is not possible to build and sustain long and multi party efforts toward the goal of building Earth scale software systems. I will not cite more specific details or particular commerical or wanna-be whole world changing distribution because what Scheme programming language implementations have achieved, are, in my opinion, valuable enough in a scope that is larger than the programming community itself.

Do-It-Yourself is good.

Conclusion

To be honest, I started the note to try to explain, first to myself, why I will work on a Scheme implementation that takes more inspiration from Python and JavaScript (but still a scheme lisp :P). There is already ALOT of package managers, schemes et al. The steering committee would like to see more portable Scheme code. I am a tiny little spark. I think there is definitly space for another Scheme, especially such as it rely on an existing compiler. DIY philosophy, fighting maintream ideas like "that is NIH and wheel re-invention" and to some extent "good enough" and boredom has shaped the human, minimalism and truth seeking aventurer and bytes wrangler that I am. I feel much more confortable with coding even if I forgot most of computer science theory. That does not mean it is useless. I learned some other ((more) practical) ideas e.g. immutability, Continuation-Passing-Style and trampolines that I already put to good use.

I need to decide what to do of my free time.

My hearth is happy with the idea of a peer-to-peer collaboration system that would make it possible to fork at the function level. Taking inspiration from radicle.xyz and unison and why not ethereum. It would also make it possible to translate definitions into your favorite natural language. No more english-only programmable programming systems. It looks like a gigantic hop, but awesome one.

2021-01-10 - Babelia search engine design planning (take three)

&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.pdf”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PDF” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.AI” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.pr
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PNG” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPG” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPEG” &q=85&fm=jpg&crop=entropy&cs=srgb&w=102
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JP2” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.j
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPF” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.BMP” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PS” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ep
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.EPS” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.MPS” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps.Z” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps.gz” &q=85&fm=jpg&crop=entropy&cs=srgb&w=1024

&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024”

I recycled two previous notes into one and added some.

I read Steve Jobs said something along the lines of “Before big projects come big thoughts”. I have been brainstorming the next steps of babelia for the last few weeks. Some of you might think that this is merely procrastinating. Announcing a project before it is done is against the netiquette, and dubbed useless. This is not only about bragging about my project and propping up my ego. One couple of months worth of thinking is a lot of work. There was almost no production of working code. Product design is a job on its own and hence deserves some recognition. That recognition will take the form of a blog post, a few hits on my web server, and I dare hope some feedback!

Prelude

I read more often than ever that people are looking for ways to build their own search engines.

Even if more on more “advanced” features are integrated into search engines. They are mostly based on human grunt work. Semantic search engine, whatever “semantic” does mean for you, is in fact merely a couple, not more than a dozen, set of tricks. I like to say, much of Google’s search engine is good old human labor. If you still doubt it, here is again: Google results are not only biased, also they are editorialized. Whether algorithms, and their bugs, party is irrelevant.

My point is: it is human made. Not some necessarily advanced alien tool.

The only thing preventing you to have your own search engine is there is no readily available software. In fact, there is commonsearch. The reason there is no public FLOSS search engine is because there is no cheap hardware.

This might sound like a crazy idea five or ten years ago, but with the advent of AMD Epyc and AMD threadripper ie. cost gravity at play getting together a personal search engine is, if not a necessity, at least a possibility.

The most common complain I read about Google is that it yields irrelevant text "that do no even contain my search terms". That is not a bug, that is a feature! It seems to me the subtext is that you can not easily customize Google or whatever search algorithm since it is privateer. Even retrieving Google search results for further processing if not forbidden, is difficult.

Big Picture

The primary user interface of a search engine is… dum dum dum… the search input box. What is interesting is what goes inside it: the so-called boolean-keyword query. For example:

```
search engine (postgresql OR psql OR pgsql OR postgres) -tsearch2
```

The intention behind that query is to retrieve the attempts to build a search engine with PostgreSQL without tsearch2 extension. As you see, the PostgreSQL concept can have multiple realization "psql" or "pgsql"… This could be handled by the search engine itself with synonym expansion.

While the discovery of synonyms is not planned, there will be a way for the user to customize babelia dictionary of lemma, one way synonyms and two way synonyms with a dedicated knowledge-base. It will be global to the instance. It will be built on top of what is known as copernic. If you are too lazy to click (and star), to put it simply: copernic is a cooperative knowledge base.

So instead, of typing the above query the user will only have to type the following:

```
search engine postgresql -tsearch2
```

On the subject of query expansion, I want to stress the importance of:

- Stem
- Lemma
- One-way synonyms
- Two-way synonyms

Possibly other lexical features that can be taken into account. For instance, the following query:

```
big search engine
```

Can be translated into the following query:

```
(big OR giant OR global) search engine
```

That can entirely be handled by the same machinery.

Another important feature of babelia is that it will both support stem and lemma. Usually stems allow you to find documents that contain words that look like the one typed in the query. Imagine the following query:

```
search engine product
```

You might want to also match the following:

```
search engine production
```

Nowadays this happens automatically. That behavior can be turned off using double quotes, like:

```
search engine "product"
```

The above query matches documents that contain the exact word “product”.

But the above approach does not always work. For instance, given the following query:

```
avoir le mojo
```

You might want to also match the following:

```
J'ai le mojo
```

Or

```
Il a le mojo
```

Or even:

```
Nous avons le mojo
```

You get it. Stems are a first step toward achieving high precision and recall because they reduce false negatives, but it does not completely eliminate the problem. To help with that, babelia will rely on lemma. Unlike stems, lemma can not be computed automagically, you need a database, that is another place where the knowledge-base will be useful.

I read often that you have to use only one of : a) stem or b) lemma. That is not true. In fact, you can use both.

To help the user a little with typos, babelia will feature a spell checker that takes its input from the index, so that it can improve itself without the need to manually update the dictionary. In the next sprint, I will only support ascii and languages that can be easily transliterated to ascii like french, Spanish, Italian, unlike Chinese, Korean and Japanese which are out-of-scope of the next alpha.

Another aspect that I prototyped in faux-texte, is query suggestion. Since, I do not have a vast amount of user queries, I can not compute nearest neighbors using word2vec or BERT with the help of something like faiss. I could do that against the index. Instead of considering user queries, which is not very privacy preserving, it will rely on the documents that were indexed. At index time, babelia will build a Markov-chain that will allow babelia to complete and mix-and-match queries.

Queries will be limited to one second at most. The limit will be based on word frequency and computed on available hardware. To workaround that limitation the user will be suggested related queries constructed with the help of Markov-chains described above.

One last aspect regarding queries. This is a significant feature, because the tagline of the project stems from that feature: the federation. Because of privacy, the queries will not be distributed all the time to other babelia instances. Instead, every babelia instance will advertise a summary of the content of their index. In case the user requests it or if there is no results in the current instance, babelia will analyze the terms and match them with other babelia instances and the user will be able to decide whether to remotely submit that query.

Problem:

If 10 users share a single server with 20 cores, hence 40 available threads that cost 100 euros per month (10 euros per person). Each user does 100 queries per day that last 1 second, the overall server utilization will still be at 2%. That CPU power needs to shared otherwise it is wasted.

Call to participation

If you are interested in a FLOSS search engine that is privacy respecting and most likely more relevant, chime into the conversation at <http://peacesear.ch>.

You can subscribe to the mailing list, send an email to the following address to subscribe:

`~peacesearch/peacesearch-discuss+subscribe@lists.sr.ht`

Conclusion

There is a gigantic leap that is going to happen in search because of hardware availability, and free software with readable source ie. the only thing that makes human progress possible: knowledge sharing.

> "Plans are only good intentions unless they immediately degenerate into hard work." Peter Drucker

2021-06-05 - Chaos: The Last Question

I am investigating this pattern combinator thing, because pattern are everywhere in programming e.g Chomsky's grammars, and also Machine Learning. The thing I am trying to build is different from Machine Learning pattern recognition. Pattern combinators are neither supervised nor unsupervised, they do not learn, they are teached with a specification constructored with combinators how to recognize patterns.

Pattern matching facility ease the work of constructing an interpreter or compiler. They are the basis of Scheme syntax-rule, syntax-case, nanopass framework, Kernel, and GNU Guile compiler tower [GNU-GUILE]. Part of Guile compiler tower, there is `foldt*` by Andy Wingo. `foldt*` builds upon Kiselyov's `foldt`. Both `foldt` and `foldt*` allow to recognize patterns in s-expr, they also allow to construct a new object (just like regular `fold`). Pattern combinators are concerned about traversing the source; the patterns in that source are recognized with user prodiced procedures (fhere, fdown, fup); pattern combinators always contruct an object following the same template, that is a flat environment, such as a mapping or association list.

In the previous note, I have glossed over `sum` that re-surface the implementation of a procedure. `dubito` some kind of inference engine. `cogito` was barely mentioned, but I meant it as an `eval`.

In this note, I will try to explain how `(sum cogito)` works.

Here is the *Shutt's Equation of Software*:

```
(define eval
  (lambda (exp env)
    (cond ((kernel-pair? exp)  (combine (eval (kernel-car exp) env)
                                        (kernel-cdr exp)
                                        env))
          ((symbol? exp)  (lookup exp env context))
          (else exp))))
```

This is a slightly modified version of `SINK`'s `eval`, I dropped the `context` that is the reification of a continuation, that in `(call/cc proc)` is passed to `PROC`. Unlike in Scheme, a context is encapsulated ie. it is its own type (see Racket and Gambit).

Tentative emulation of Kernel with Scheme

About Scheme, I will try to describe the semantic of Kernel using Scheme without actually building an interpreter in Scheme (that is what SINK does). Instead, I will describe a subset of Kernel in particular, the `$vau` operative, that allows to create new operative with Scheme pseudo-code:

```
(define-syntax $vau
  (lambda (stx)
```

```
    (syntax-case stx ()
      (($vau args env body ...)
       #'(lambda (args env) body ...)))))
```

That is wild guess ie. prelimeray translation. To call a procedure constructed with `$vau` you need another syntax transformer and a few helpers:

```
(define-syntax apply-vau
  (lambda (stx)
    (syntax-case stx ()
      ((apply-vau operative args ...)
       #'(let ((dynamic-environment (environment-current)))
           (operative 'args dynamic-environment))))))
```

(Now that I think about it, it can be implemented in terms of `syntax-rule`)

Mind the quoted `args`, `args` is constructed in the template as a quoted expression, and its evaluation might be done in `body ...` with the help of Scheme's `(eval exp environment)`.

`environment-current` is like Guile's `the-environment`.

Another emulation of Kernel with Scheme

Another approach is to use only procedures and no macros. For instance:

```
(define (my-operative args env)
  body ...)
```

Since there is two kinds of callables in `Kernel`: operative and applicative. One will need to prefix every Scheme call with a `kernel-call`:

```
(define (kernel-call combiner env args)
  (if (applicative? combiner)
      ((unwrap combiner) (map (lambda (arg) (eval arg env))))
      ;; otherwise, it is an operative
      (combiner args env)))
```

Where `applicative?` is the predicate associated with:

```
(define-record-type <applicative>
  (wrap operative)
  applicative?
  (operative unwrap))
```

Then what you write in Kernel as:

```
(my-combiner a b c)
```

Would be transpiled as:

```
(kernel-call my-combiner (environment-current) (list 'a 'b 'c))
```

`(environment-current)` current can be constructed at compile time, since it is the static / lexical scope (I hope).

The only missing piece I can forsee is that Kernel support trees as operands, but that requires more work, including a pattern matcher!

`sum`, or no `sum`

Previously, I wrote the goal of `sum` is ultimately to call `(sum eval)` and return an object language representation of `eval` in the meta-language. In other words, it would return the full source of the compiler, one language-level below the object language.

I will investigate `(sum +)`: what is the meta-language definition of the applicative `+` in Scheme:

```
(define (sum +)
  (applicative (operative (lambda (args env) (+ (env-ref env 'a) (env-ref env 'b))))))
```

It does not make sense to compile a mere `lambda`, you need a full lambda such as:

```
(lambda (args env) (+ (env-ref env 'a) (env-ref env 'b)))
```

It would be transpiled to the following web assembly:

```
(func $a.683 (param $cl.397 externref) (result i32) (result externref)
      (local $a externref)
      (local $b externref)
      (local $out externref)
      (local.set $k (table.get $stack (i32.const 0)))
      (local.set $a (table.get $stack (i32.const 1)))
      (local.set $b (table.get $stack (i32.const 2)))
      (local.set $out (call $+ (local.get $a) (local.get $a)))
      (table.set $stack (i32.const 0) (local.get $out))
      (i32.const 1) ;; continue = yes
      (local.get $k)) ;; continuation closure
```

The previous text would be the representation of `(sum (sum +))` in the object language. More or less.

References

- [GNU-GUILE] https://www.gnu.org/software/guile/docs/docs-2.2/guile-ref/Compiler-Tower.html

2021-06-04 - Codex: Dubito, ergo Cogito, ergo Sum

Last couple of weeks I learned about:

- 3-LISP: a theorical evolution of LISP where everything can be inspected;

- Kernel: an evolution of Scheme that forgo syntax transformers as described in Scheme;
- Control Flow Analysis / Just-In-Time and Ahead-Of-Time compilation / optimizations: what I call "dubito". It is known as partial / symbolic / algebraic evaluation, and includes type inference.

My goal is to devise and build an optimizing compiler which does not precludes AOT or JIT. In other words: I want both AOT and JIT. As a subgoal, I want to build an optimizing or optimized pattern matching program in the spirit of SRFI-200, and SRFI-204, that I call pattern combinators. One particular optimization, I think about is what Software Design for Flexibility [SDF] calls unification.

For instance given the following `match` expression:

```
(match exp
  ((a) out1)
  ((a d) out2))
```

Can be rewritten as follow:

```
(match exp
  ((a . rest) (match rest
                (() out1)
                ((d) out2))))
```

In that case, doing some kind of prefix compression.

`match` can be seen as a pattern combinator, that is a parser combinator that can traverse not only a sequence, but also a sequence made of nested sequence in a recursive way. For the purpose of the pattern combinator, a sequence is a finite acyclic ordered set. A sequence might be a list, vector or an ordered mapping. A parser combinator such as the one described in [GLL] is a good basis for a pattern combinator, except if does not support the optimizing / optimized part.

To be able to implement optimizing pattern combiner, it would require to reflect upon the match predicates, in particular have access to a predicate `subsume?` that allows implement a partial order over predicates. And that should be done at in the object language.

The object language is the language of the user. It is defined in opposition to meta language that is used to implement the object language. The meta language might be itself implemented in terms of another meta language until… until all the way down! That is where 3-LISP call into atoms, the physical atoms (not the atoms of the simulation).

We will just glimpse over the problem(s), and add a reflection applicative called `sum`, so that we can introspect how applicative and operative are made, so that in turn we may be able to optimize code at runtime from the object language.

Instead of trying to implement `subsume?` and the optimizing pattern compiler, I will implement only arithmetic folding (replace an arithmetic computation with its value) with an applicative based on the inferences / knowledge / theories built by an applicative called `dubito`.

So, given the following pseudo-code:

```
(define (make-frob-adder value)
  (lambda (other)
    (+ other value 36)))

(define my-frob-adder-2 (make-frob-adder 2))

(my-frob-adder-2 4) ;; => 42
```

A simple compiler will just compile `my-frob-adder-2` into a `lambda` that is passed an extra environment argument called `static` so that the resulting procedure does not have free-variables, it will look like:

```
(define (my-frob-adder-2-compiled static other)
  (define value (environment-ref static 'value))
  (+ other value 36)
```

The trick is that if we forgo the last line of the previous program, the closure of `my-frob-adder-2` is known at compile time. If we also consider the last line, the whole program can be compiled to `42`: That is what is called (AFAIK) constant folding.

Let's consider the following program:

```
(define (make-frob-adder value)
  (lambda (other)
    (+ other value 36)))

(define my-frob-adder-2 (make-frob-adder 2))

(my-frob-adder-2 (string->number (cadr (command-line))))
```

It is similar to the original program, except it `my-frob-adder-2` takes its argument from command-line, hence it can not be folded into an integer object, because… `(command-line)` is not known at compile time. So, indeed after a simple compilation, we end up with the following code:

```
(define (my-frob-adder-2 static other)
  (define value (environment-ref static 'value))
  (+ other value 36))

(my-frob-adder-2 (alist->environment `((value . ,2))) (string->number (cadr (command-line))))
```

That is naive example: an optimizing compiler can create the following optimized definition for `my-frob-adder-2`:

```
(define (my-frob-adder-2 other)
  (+ other 40)) ;; 36 + 4 = 40
```

So what is this all about?!1!

It is also seems a little naive given that Chez Scheme can compile to native code an s-expr at runtime, given a perfomance timer (cost center), and counters, that gather new knowledge only known at runtime, it is completly possible to implement a manual JIT [MANUAL-JIT].

So what do `sum` do that is new? Not much: it puts all that together.

Kernel [R-1RK] will factorize macro and procedure into operatives, it will also reify all environments: static / lexical, and dynamic. The application `sum` will reify in the object language the implementation of any object… including `eval` in other words, `eval` can be reifyed as a meta-evaluator or better as the "source code" in the previous meta language. `sum` does not only mean that the source is embedded in the program, but also prescribe that it is possible to represent in a possibly infinitly resursive way all the meta-languages. The limit being our knowledge, time, and energy to encode that as source. Does it mean that this language is self-bootstrappable, in other words that it does not require a seed: no. Even it it self-hosted and self-describing, it requires a seed [BOOTSTRAPPABLE].

In the above paragraph I glossed over `(sum eval)`: i) does it return a meta-evaluator, a program represented in the object language that can evaluate the object language (itself), ii) does it return an evaluator in the previous meta-language. Both are possible, the latter is more interesting, because it expose the compiler tower [NANOPASS] to the object language.

`eval` is not the only interesting primitive procedure. In the context of Scheme, it is interesting to reflect upon a couple of procedures. One of the most interesting is `call/cc`, another interesting procedure is the non-standard `expand` and yet another everything related to the garbage collector. To stay goal-driven, a target application of that reflection, that I think may be interesting, can be to disable or customize the garbage collector for a subset of the program. Another goal might to change for a subset of the program the evaluation strategy without relying on meta-evaluation [EVALS].

In the case where `(sum eval)` returns the source of `eval` in the previous meta-language, `expand`, `call/cc`, and the garbage collector should be represented… somehow!

References

- [SDF] https://mitpress.mit.edu/books/software-design-flexibility
- [GLL] https://epsil.github.io/gll/
- [MANUAL-JIT] https://m00natic.github.io/lisp/manual-jit.html

- [R-1RK] https://web.cs.wpi.edu/~jshutt/kernel.html
- [BOOTSTRAPPABLE] http://bootstrappable.org/
- [NANOPASS] https://nanopass.org/
- [EVALS] https://en.wikipedia.org/wiki/Evaluation_strategy

2021-02-21 - An open letter to Mez Breeze

Dear Mez Breeze,

It is been long time. I turned 36 last week. This year also the 10th year I am working professionally as software engineer. Ten years ago, is also the time when I discovered your work. I remember in 2011, I read "Program or be programmed" by Douglas Rushkoff, and "The Filter Bubble" by Eli Pariser. Obama was president of the USA. I was trying to translate "Culture and Empire" by Pieter Hintjens in french. My domain name was something starting with hypermove, far fetched reference to Oxmo Puccino song "On ne danse pas":

That was validated by Ox himself. It was the beginning of Google+. I wanted to build a social network, reminiscent of my most popular previous project on top of Neo4J(ava), with the help of Tinkerpop, Gremlin, and my biggest time sink, and most dangerous programming language: Python.

I wanted an hypercube as the logo for wanna-be my hypersocial website. At the time, a socialite was defined by Google search engine as the one that likes to share. That is how I stumbled upon Augmentology blog. I remember that, not because I have giant memory. I remember all of that because it all makes sense in my life. You, explained a lot of the things I had doubt about. You were the concrete realization of many of my dreams, and questioning answers commonly accepted as truth. Even if you are noticeable and noticed, you take positions that go unnoticed or silenced around me. It is or least part of my education, partly influenced by various aspects of my life that getting noticed is not a good thing. After all, my dream is a continuous mesh of person with equals rights and opportunities where any of them can enjoy freedom, fulfill their passion, and realize their dreams. You have a Wikipedia page. That is where I discovered about Mezangelle. Mezangelle made my mind, and my skills stronger. A Big Sister. Eventually there was hope. I crafted some lines myself, even got encouragement by you and others.

I kept a sugared version of my favorite line:

> $ git commit –message="mezangelle: revert silence for a peaceful sanity"

I unleashed Mezangelle full steam hidden in a very common feature such as an alias: amz3. The official explanation is the following:

> amz3 is a compressed representation of a morph-syntactic combinatorics transderivationnal search in a graphical semantic space with

> the help of pattern matching that aggregates several senses. The induced experience while reading it is a reference to the Isaac Asimov novel «Little Lost Robot» in practice an obscure form of poetry named Mezangelle.

Karine and you were the ones that gave me the strength, courage and desire to voice myself.

Project-ion

What remains of that stream of thought is one question: “Should I learn French?”. A couple of month later, you requested a translation of one piece I wrote and recorded. I did my best, and you told me that it was great. It is not about you or for you. It is time to break the spell. It is for nobody, and everybody in particular. I was meant to taunt. It is not perfect, I like it, even with the imperfections. There is a particular imperfection, that I do not want to take away, I was merely trying to voice some the infinite meticulous poetry of Mezangelle. Any time I think about it, I remember about you, Mez Breeze and the perfect deep drum of senses rhythm that are spelled out in your Mezangelle. Your question about learning french was impossible to answer. “Choose you own destiny”. And the ever lasting question: For me? In fact, the last verses are a tribute to Mezangelle. And even the last line can only start to make sense if / when you both know french and Mezangelle.

It difficult today to tell what exists thanks to you, and what is my own deed.

You may or may not have invented Mezangelle, maybe all Mezangelle is a invention by you. My mother tongue and its associated history is a lie. Still, you are the one that wield it when I needed it.

Thank You!

2021-09-12 - How to choose a database?

How accurate the search hits should be (cf. Precision and Recall? Do you need “best ranked results” or “some result” or ”all the matching documents? or How fast should the query be: sub-second or a second / minute / hour / day? What is the expected workload regarding read and write? I guess consistency is must have, how fast do you need writes to be available for read? Do you need deep traversal kind of queries? Do you need textual search, and / or multiple criteria look up with exact match, sorting, and possibly facets?

2021-04-10 - (import (okdb))

Figure 6: The abstraction of architecture of the Abu Dhabi Louvre Museum ceiling it’s a piece of art on it’s own.

Status

Rework draft.

CHANGELOG

- 2021/04/10: v0
- 2021/04/17: misc

Issues

- Transaction begin, rollback, before and after commit hooks are missing.
- Maybe add specification about thread safety.
- Add a parameter `okdb-transaction-hygiene` that may be used to set the desired serializability guarantee, possibly on a per transaction basis with the help of parametrize. Using pseudonyms, that maybe change over time: perfect, strong, weak, none. And also using SQL standard names: serializable, snapshot-isolation, read-commited, read-uncommited.

Abstract

General purpose backend storage datastructure for building in-memory or on-disk databases that can handle concurrency thanks to ACID transactions.

Rationale

`okdb` can be the primitive datastructure for building many datastructures. Low level extensions include counter, bag, set, mapping-multi, binary object store. Higher level extensions include Entity-Attribute-Value possibly supported by datalog, Generic Tuple Store (nstore) inspired from Resource Description Framework that can trivialy match the query capabilities of SPARQL, nstore can painlessly implement RDF-star, or even the Versioned Generic Tuple Store (vnstore), that ease the implementation of bitemporal databases, and datomic high level interface. Also, it is possible to implement a property graph database, ranked set, leaderboard, priority queue. It is possible to implement efficiently geometric queries such as xz-ordered curves.

`okdb` is useful in the context of on-disk persistence. `okdb` is also useful in a context such as client-server applications, where the client need to cache heterogeneous data. It may be used in the browser, or in microservice configuration as a shared in-memory datastructure.

There is several existing databases that expose an interface similar to `okdb`, and even more that use an ordered key-value store (okvs) as their backing storage.

While `okdb` interface is lower-level than the mainstream SQL, it is arguably more useful because the implementation stems from a well-known datastructure part of every software engineering curriculum, namely binary trees, also because it

allows to implement SQL, last but not least it reflects the current practice that builds (distributed) databases systems based on a similar interface.

`okdb` extends, and departs from the common okvs interface inherited from BerkeleyDB, to ease the implementation thanks to bounded keys and values, while making the implementation of efficient extensions easier also thanks to the ability to estimate the count of keys, and the size of key-value pairs, in a given range.

This SRFI should offer grounds for apprentices to learn about data storage. It will also offer a better story (the best?) for managing data that may be durable, and read, or written concurrently.

Reference

`(make-okdb filepath [block-read block-write]) string? procedure? procedure? → okdb?`

Rationale: In SRFI-167, `make-okvs` could take various options. The interface was difficult, and did not work well. Instead, of trying to define a couple of options, a left aside others. With `okdb` it left to downstream to deal with options. It is the responsability of the implementer, and possibly eventually to the user to deal with options in an appropriate way. One way, for the implementer to enable options is to create a super procedure that a) returns multiple values, including the constructor, b) rely on generic methods, or something else.

Return a handle of the database. `FILEPATH` may be a string describing the on-disk file or directory where the database will be saved. In the case where `okdb` work only from memory, it should be ignored.

`BLOCK-READ` and `BLOCK-WRITE` if provided will be used to consume or produce bytes that will be read or written to disk, possibly using cryptography or compression on input bytes. Both `BLOCK-READ` and `BLOCK-WRITE` will take a generator and an accumulator as argument.

`(okdb? obj) * → boolean?`

Returns `#t` if `OBJ` is an `<okdb>` instance. Otherwise, returns `#f`.

`(okdb-close! db) okdb?`

Close `DB`. All transactions that completed successfully should be available the next time the database is open with `make-okdb` except in the case of fully in-memory database.

`(okdb-transaction? obj) * → boolean?`

Returns `#t` if `OBJ` is an `<okdb-transaction>` instance. Otherwise, returns `#f`.

`(okdb-cursor? obj) * → boolean?`

Returns `#t` if `OBJ` is an `<okdb-cursor>` instance. Otherwise, returns `#f`.

`(okdb-handle? obj) * → boolean?`

Returns `#t` if `OBJ` satisfy either `okdb?`, `okdb-transaction?`, or `okdb-cursor?`. Otherwise, returns `#f`.

`(okdb-key-max-size handle) handle? → number?`

Rationale: Most okvs implementations do not specify the maximum size of keys, making both the implementation and its use erratic. The same maximum size might not work in all situations, hence it might be subject to customization. The important is to guarantee some predicatable performance when keys follow that constraint. It also makes the implementation of `okdb` easier, among other thing because the implementation does not need to have to handle large binaries. That is not a negligeable constraint for the user as keys max size are not necessarly predicatable, but in any case should be small since in most implementations those are kept in memory.

Return the maximum size of key for the database associated with `HANDLE`. It is an error to call `okdb-key-max-size` if `okdb-key-max-size!` was never called before.

`(okdb-key-max-size! okdb size) okdb? number?`

Questions: Can `SIZE` be `+inf.0`? Does it work across restarts? Replace `okdb?` with `handle?`? Investigate why FDB does limit those.

Set the maximum `SIZE` of a key for the database `OKDB`.

`(okdb-value-max-size handle) handle? → number?`

Rationale: Same as the above: it is easier to implement. For the user perspective, it is much easier to handle the situation of large values since they can be split without loosing features.

Return the maximum size of a value of the database associated with `HANDLE`.

`(okdb-value-max-size! okdb size) okdb? number?`

Questions: same as `okdb-value-max-size!`

Set the maximum `SIZE` of a value for the database `OKDB`.

`(okdb-conflict? obj)`

Returns `#t` if `OBJ` is a conflict error. Otherwise returns `#f`. Such object may be raised by `okdb-in-transaction`.

`(okdb-in-transaction okdb proc [failure [success]]) okdb? procedure? procedure? procedure? → (values (every? *))`

Rationales:

- `okdb-in-transaction` does not include a retry logic when `okdb-conflict?` is raised because retrying might require to wait which depends on the implementation but also and more importantly on user code. The user is in the best position to know when, and how to retry the transaction. The last resort strategy is not even to retry the transaction immediatly, but to put the operation in queue possibly persisted in the database, and force the serialization through a single thread. In any case, retry should be explicit in user code.
- Serializable scheme trades guarantees regarding the consistency of the data, and hence ease of development because the state of the database is determinist versus performance. The prescription of serializable transactions is a strong requirement, that was thusfar almost completly left aside in the industry in favor of snapshot isolation. The philosophy here is: *make it work, then make it fast.* It is not possible to build reliable systems upon claims that are weak, or false, in the general case.
- Nested transactions were ruled out because it is still not clear whether they put a strain on the implementation that does not yield much help in user code. Nested transactions are similar to savepoints or autonomous transactions.

`okvs-in-transaction` describes the extent of the atomic property, the A in ACID, of changes against the underlying database. A transaction will apply all database operations in `PROC` or none: all or nothing. When `okdb-in-transaction` returns successfully, the changes will be visible for future transactions, and implement durability, D in ACID, and when the database implements on-disk storage, across restarts. In case of error, changes will not be visible to other transactions in all cases. Regarding isolation, and consistency, respectively the I and C in ACID, serializable transactions is prescribed: the concurrent execution of `(okvs-in-transaction okdb proc ...)` should render the database as if the concurrent transactions were executed serially ie. without overlapping time, in some order, possibly rejecting some of them with an error that satisfy `okdb-conflict?`. In particular, it is stronger than snapshot isolation.

Begin a transaction against the database, and execute `PROC`. `PROC` is called with first and only argument an object that satisfy `okdb-transaction?`. In case of error, rollback the transaction and execute `FAILURE` with the error object as argument. The default value of `FAILURE` re-raise the error with `raise`. Otherwise, executes `SUCCESS` with the returned values of `PROC`. The default value of `SUCCESS` is the procedure `values`.

`okdb` does not support nested transactions.

TODO: what about hooks

In case `okvs-in-transactions` raise an error that satisfy `okdb-conflict?`, the user may re-run the same transaction taking care that `PROC` is idempotent.

`(okdb-call-with-cursor handle proc) okdb-handle? procedure? → (values (every? *))`

Open a cursor against `HANDLE` and call `PROC` with it. When `PROC` returns, the cursor is closed.

If `HANDLE` satisfy `okdb?`, `okdb-call-with-cursor` must wrap the call to `PROC` with `okvs-in-transaction`.

If `HANDLE` satisfy `okdb-cursor?`, `okdb-call-with-cursor` must pass a cursor positioned at the same position as `HANDLE` and the same keys snapshot.

The produced cursor is not positioned, except when `HANDLE` satisfy `okdb-cursor?`, it is an error to call `okdb-cursor-next?`, `okdb-cursor-previous?`, `okdb-cursor-key`, or `okdb-cursor-value`. When `HANDLE` does not satisfy `okdb-cursor?`, the user should call `okdb-cursor-seek?` immediatly after `okdb-call-with-cursor`.

The cursor that is created will see a snapshot of keys of the database inside the transaction taken when `okdb-call-with-cursor` is called. During the extent of `PROC`, `okdb-set!` and `okdb-remove!` will not change the position of the cursor, and the cursor will see removed keys and not see added keys. Keys which value was changed during cursor navigation, that exist when `okdb-call-with-cursor` is called, can be seen. That is, the cursor is stable.

`(okdb-estimate-key-count handle [key other [offset [limit]]]) handle? bytevector? bytevector? integer? integer? → integer?`

Rationale: It is helpful to know how big is a range to be able to tell which index to use as seed. Imagine a query against two attributes, each attribute with their own index and no compound index: being able to tell which subspace contains less keys, can speed up significantly query time.

Return an estimate count of keys between `KEY` and `OTHER`. If `KEY` and `OTHER` are omitted return the approximate count of keys in the whole database.

If `OFFSET` is provided, `okdb-estimate-key-count` will skip the first `OFFSET` keys from the count.

If `LIMIT` is provided, `okdb-estimate-key-count` will consider `LIMIT` keys from the count.

`(okdb-estimate-bytes-count handle [key [other [offset [limit]]]]) okdb-handle? bytevector? bytevector? integer? integer? → integer?`

Rationale: That is useful in cases where the size of a transaction is limited.

Return the estimated size in bytes of key-value pairs in the subspace described by `KEY` and `OTHER`. If `OTHER` is omitted, return the approximate size of the key-value pair associated with `KEY`. Otherwise, return the estimated size of the whole database associated with `HANDLE`.

If `OFFSET` is provided, `okdb-estimate-bytes-count` will skip the first `OFFSET` keys from the count.

If `LIMIT` is provided, `okdb-estimate-bytes-count` will consider `LIMIT` keys from the count.

`(okdb-set! handle key value) okdb-handle? bytevector? bytevector?`

Associate to the bytevector `KEY`, with the bytevector `VALUE`. If `HANDLE` satisfy `OKDB?` wrap the operation with `okdb-in-transaction`.

If `HANDLE` satisfy `okdb-cursor?`, `okdb-set!` does not change the position of `HANDLE`.

`(okdb-remove! handle key) okdb-handle? bytevector?`

Removes the bytevector `KEY`, and its associated value. If `HANDLE` satisfy `OKDB?` wrap the operation with `okdb-in-transaction`.

If `HANDLE` satisfy `okdb-cursor?`, `okdb-set!` does not change the position of `HANDLE`.

`(okdb-cursor-seek cursor strategy key) okdb-cursor? symbol? bytevector? → symbol?`

Position the `CURSOR` using `STRATEGY`. `STRATEGY` can be one of the following symbol:

- `less-than-or-equal`
- `equal`
- `greater-than-or-equal`

The strategy `less-than-or-equal` will first seek for the biggest key that is less than `KEY`, if there is one, it returns the symbol `less`. Otherwise, if there is a key that is equal to `KEY` it will return the symbol `equal`. If there is no valid position for the given `KEY`, it fallbacks to the symbol `not-found`.

The strategy `equal` will seek a key that is equal to `KEY`. If there is one it will return the symbol `equal`. Otherwise, it returns the symbol `not-found`.

The strategy `greater-than-equal` will first seek the smallest key that is greater than `KEY`, if there is one, it returns the symbol `greater`. Otherwise, if there is

a key that is equal to `KEY` it will return the symbol `equal`. If there is no valid position for the given `KEY`, it fallbacks the symbol `not-found`.

`(okdb-cursor-next? cursor) okdb-cursor? → boolean?`

Move the `CURSOR` to the next key if any. Return `#t` if there is such a key. Otherwise returns `#f`. `#f` means the cursor reached the end of the key space.

`(okdb-cursor-previous? cursor) okdb-cursor? → boolean?`

Move the `CURSOR` to the previous key if any. Return `#t` if there is such a key. Otherwise returns `#f`. `#f` means the cursor reached the begining of the key space.

`(okdb-cursor-key cursor) okdb-cursor? → bytevector¿`

Return the key bytevector where `CURSOR` is positioned. It is an error to call `okdb-cursor-key`, when `CURSOR` reached the begining or end of the key space or when `CURSOR` is not positioned.

`(okdb-cursor-value cursor) okdb-cursor? → bytevector¿`

Return the value bytevector where `CURSOR` is positioned. It is an error to call `okdb-cursor-key`, when `CURSOR` reached the begining or end of the key space or when `CURSOR` is not positioned.

`(okdb-query handle key [other [offset [limit]]]) handle? bytevector? bytevector? integer? integer? → (either? bytevector? procedure?)`

`OKDB-QUERY` will query the associated database. If only `KEY` is provided it will return the associated value bytevector or `#f`. If `OTHER` is provided there is two cases:

- `KEY < OTHER` then `okdb-query` returns a generator with all the key-value pairs present in the database between `KEY` and `OTHER` excluded ie. without the key-value pair associated with `OTHER` if any.
- `OTHER < KEY` then `okdb-query` returns the equivalent of reversing the generator returned by `(okdb-query handle KEY OTHER)`.

If `OFFSET` is provided the generator will skip as much key-value pairs from the start of the generator.

If `LIMIT` is provided the generator will generate at most `LIMIT` key-value pairs.

`(okdb-bytevector-next-prefix bytevector) bytevector? → bytevector?`

Return the bytevector that follows `BYTEVECTOR` according to lexicographic order that is not prefix of `BYTEVECTOR` such as the following code iterates over all keys that have `key` as prefix:

```
(okdb-query handle key (okdb-bytevector-next-prefix key))
```

2021-08-25 - Jack: One Thread Per Core

I have being making progress with Babelia on the web user interface, IRC interface, and also the backend.

Regarding the backend, even if Chez Scheme is fast, it is clear even on the small dataset I have put together, that is around eleven gigabytes without compression. I need something like map-reduce [1], in LISP world known under the name of `for-each-parallel-map`.

In full-text search, and in a search product like google, they are tips, and tricks to avoid to hit the worst case. The worst being a query where the least frequent word is also one of the most frequent in the index. Possible workarounds include 0) using `AND` as the default operator 1) eliminating most common word (also known as stop-words); 2) caching results; 3) approximating results with user profiling…

All those workarounds give rise to other problems, or they need a lot of work such as profiling users, which is in my opinion not a problem when that is limited to profiling users' data that are published in the open (unlike tracking search users via their queries, or mail, etc…).

Anyway, threads are difficult, so I wanted to give it try. The above is trying to explain from where my motivation stems from.

> It is still unclear whether `make-jack` works reliably all the time. You tell me.

`(make-jack count) → procedure?`

`make-jack` initialize a pool of `COUNT` parallel threads, and return a possibly endless generator that produces *jacks*. A jack is made of two procedure:

1. The first procedure is an accumulator that will consume one or more thunks. That is how the user request the parallel execution of something.
2. The second procedure will generate the results of the thunks submitted with the associated accumulator in an unspecified order.

Example:

```
(call-with-values jack
  (lambda (consumer producer)
```

```
    ;; submit some work to the pool, the consumer will block
    ;; if there is too much work already scheduled.
    (consumer thunks)
    ;; pull results
    (let loop ((count (length input)))
      (unless (fxzero? count)
        ;; producer will block the current thread until there
    ;; is something to produce
        (display (producer))
    (newline)
        (loop (fx- count 1))))))
```

Here are some numbers that backup the claim that it may work as expected, the tests were done with pool of size five. The work, called `THUNKS` in the above snippet, is the computation of one thousand times fibonacci of 40:

```
(time (call-with-values jack ...))
    no collections
    0.029955076s elapsed cpu time
    152.646622367s elapsed real time
    592688 bytes allocated
        Command being timed: "scheme --libdirs src/ --program main.scm"
        User time (seconds): 761.84
        System time (seconds): 0.04
        Percent of CPU this job got: 498%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 2:32.71
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 49624
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 14194
        Voluntary context switches: 1011
        Involuntary context switches: 3646
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

The code:

```
  (define-record-type* <queue>
```

```
  ;; This is a fifo queue, that can signal when items are available
  ;; or space is available. Space is available when there is less
  ;; than MARK items inside the REST. It is used in jacks in both
  ;; accumulators and generators.
  (make-queue% name head rest mark mutex item-available space-available)
  queue?
  (name queue-name)
  (head queue-head queue-head!)
  (rest queue-rest queue-rest!)
  ;; mark is an integer that allows to keep the number of produced,
  ;; and accumulated values low; Tho, there is room for starvation.
  (mark queue-mark)
  (mutex queue-mutex)
  (item-available queue-item-available)
  (space-available queue-space-available))

(define (make-queue name mark)
  (make-queue% name '() '() mark (make-mutex) (make-condition) (make-condition)))

(define (queue-pop! queue)
  (mutex-acquire (queue-mutex queue))
  (if (null? (queue-head queue))
      (if (null? (queue-rest queue))
          (begin
            ;; Wait for work...
            (condition-wait (queue-item-available queue) (queue-mutex queue))
            (mutex-release (queue-mutex queue))
            ;; recurse
            (queue-pop! queue))
          (let* ((item+new-head (reverse (queue-rest queue)))
                 (item (car item+new-head))
                 (new-head (cdr item+new-head)))
            ;; There is nothing in the head, but the rest has stuff
            ;; reverse the rest to keep it FIFO, and replace the
            ;; HEAD with it. Return the first item immediatly to
            ;; avoid to pressure the mutex, and best performance.
            (queue-rest! queue '())
            (condition-signal (queue-space-available queue))
            (queue-head! queue new-head)
            (mutex-release (queue-mutex queue))
            item))
      ;; There is work, all is well.
      (let ((item (car (queue-head queue)))
            (new-head (cdr (queue-head queue))))
        (queue-head! queue new-head)
        (mutex-release (queue-mutex queue))
```

```
        item)))

(define (queue-push! queue . items)
  (mutex-acquire (queue-mutex queue))
  ;; we only check that the rest is less than the mark. BUT the user
  ;; may append more than mark ITEMS.
  (if (fx<? (length (queue-rest queue)) (queue-mark queue))
      (begin
        (queue-rest! queue (append items (queue-rest queue)))
        ;; TODO: It may not be necessary to wake up all waiting
        ;; threads, but only (length (queue-rest queue))?
        (condition-broadcast (queue-item-available queue))
        (mutex-release (queue-mutex queue)))
      (begin
        ;; Block until some work is done.
        (condition-wait (queue-space-available queue) (queue-mutex queue))
        (mutex-release (queue-mutex queue))
        ;; TODO: here it is possible to append the items without
        ;; recursing.
        (apply queue-push! queue items))))

(define (make-jack count)

  ;; That is the queue for all work for the created thread pool.
  ;; The mark is an arbitrary number, it could be an argument.
  (define input (make-queue 'input (fx* count 2)))

  (define (worker-loop index input)
    ;; TODO: replace thunk+output with output+thunk, and avoid the
    ;; cdr before the car.
    (let ((thunk+output (queue-pop! input)))
      (queue-push! (cdr thunk+output) ((car thunk+output)))
      (worker-loop index input)))

  (define (worker-init count)
    (let loop ((count count))
      (unless (fxzero? count)
        ;; count is passed as the thread index for debugging
        ;; purpose
        (fork-thread (lambda () (worker-loop count input)))
        (loop (fx- count 1)))))

  (define (input-consumer input output)
    (lambda (thunks)
      ;; TODO: avoid the call to apply. The reverse is necessary, to
      ;; keep around the priority information FIFO.
```

```
    (apply queue-push! input (reverse (map (lambda (thunk) (cons thunk output)) thunks)))

(define (output-producer output)
  (lambda ()
    (queue-pop! output)))

;; Initialize thread pool at call site, that is somewhat unusual
;; for a generator to have side-effects outside producing values.
(worker-init count)

(lambda ()
  ;; again the mark is a clueless guess.
  (define output (make-queue 'output (fx* count 2)))
  (values (input-consumer input output) (output-producer output))))
```

Reference

1. MapReduce: Simplified Data Processing on Large Clusters

2021-01-31 - mutation: review & rework of mutmut

A coworker told me about mutation testing, I was immediately interested because testing is interesting. Testing became even more interesting when I read how FoundationDB was made. I recommend you watch the video about testing with deterministic simulation.

I started looking into mutmut and a fork that adds parallel runs support… eventually I though: how hard can it be?

Yeah, you might think that is my thing, and that is the thing of many (most?) developers starting on a new project: "the code is evil, let's rewrite!". And that is somewhat the idea behind this series of logs, so I will not say I am not into "rewrites" myself (more on that later).

I went swiftly through all projects that pop'ed in google first page result, and still was interested to rewrite. Let's dive deeper into those projects.

Instead of an introduction to mutation testing, let our imagination play with the following nice track from Disiz Peter Punk called Mutation:

Disiz Peter Punk Intro Mutation

> Standing on the shoulders of giants

mutmut

After my initial review, mutmut seemed like the most straightforward except the fact that you can no run tests in parallel but there is a fork that does. The

&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.pdf”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PDF”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ai”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.AI”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.png”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PNG”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jpg”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPG”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jpeg”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPEG”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jp2”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JP2”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jpf”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPF”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.bmp”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.BMP”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PS”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.EPS”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.mps”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.MPS”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.pz”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps.Z”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps.Z”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps.gz”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps.gz”

&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024”

Figure 7: black man with the same mutation wolfverine from x-men

code is another story. I will go through the fork because it is the code that I am most interested in.

`cache.py`

In no particular order:

- I am still clueless about what is the point short lines of text as global variables,
- I am an early fan of Object-Relation-Mapper, and I changed my mind. I want to stress that I am not the only one to find the ORM abstraction dubious.
- init_db is way too much indented and score 7 level of indentations.
- If we dive into init_db we figure that it is a decorator, hence it is not as evil as I was thinking to nest function definition. Nested function definition do not play nice with the module pickle. It is also painful from a performance perspective because the CPython VM will not optimize it (closure allocation, useless free variables and even the function definition that will be re-computed and re-instanciated everytime the VM goes through init_db. The situation is worse when a class is allocated at runtime).
- Re init_db, the first branch of the if should return early. That is a case where "Do not Repeat Yourself" is harmful.
- The except OperationalError: pass is dubious.
- There is other problems with code details, but really the big design mistake is to make database initialization something unpredictable. initdb will decorate database function and initialize the database once and only once the first time one of the cache (!) function is called. From experience, it is much better to have a dedicated function e.g. the function called wrapper called at every entry points of the program or programs.
- Also db object is a global variable instanciateed from pony.orm.Database at the top-level. I will not spend time discussion slow global variable access with CPython, instead ask myself, why the database initialization is not done at that point.
- The good thing in this code is that it use transaction (even if the name is not explicit)
- A small nitpick the following code is difficult to test:

```
def hash_of(filename):
    with open(filename, 'rb') as f:
        m = hashlib.sha256()
        m.update(f.read())
        return m.hexdigest()
```

Because it rely on a side-effect open the good abstraction if one is necessary, is a sugar (!) that takes bytes as arguments and computes the digest:

```
def sha256sum(bytes):
    m = hashlib.sha256()
    m.update(bytes)
    out = m.hexdigest()
    return out
```

The function sha256sum is very easy to tests, no need to fiddle with on disk files during testing.

- In hash_of_tests: Modern python code should use pathlib.Path and glob pattern matching.
- The whole thing could be re-written as a list comprehension and a proper use of a helper function.
- `found_something` is not necessary, it should be an independent predicate. I am not a great fan of exceptions, but return NO_TESTS_FOUND would be written as raise SomeException in pythonic code (in the case where there is no predicate that guard the hash computation).
- print_result_cache has too many arguments and is too much nested (maybe use a dataclass?)
- sorted(iterable) == list(sorted(iterable)) the list is spurious, it just does copy the list created by sorted.
- itertools.groupby takes a sorted iterable as argument.
- create_html_report does not follow the "separation of concerns" principles. It should really be at least two function 1) one to gather the data 2) another to render the html
- Overall, the functions that are really related to the database take high level abstractions that makes it a) difficult to test b) force the data access layer to do operations that are not really data related (like preparing the actual values to create, read, update or delete) c) basic data types are easier to debug (except generators).
- I frown upon the use of getattr

I like the idea of cache.py but the name is not well chosen, I prefer db.py or something like dal.py for data access layer.

`__init__.py`

I start to think having code in **init**.py is not as evil as I though, I might just be biased because of my experience with Django in the early days.

- RelativeMutationID would be a good case of a dataclass.

- Again a lot of indentation.
- mutant_statuses would make a good Enum
- Ha! `**_`: it looks like a snake with big glasses!
- All the mutation function and sort-of framework could use their own file.
- The use of Context instances is broad and large in the code base but there is no docstring. It seems to be a configuration, with a lot of @property . Note: I consider @property harmful. dataclass? dict?

The following pattern:

```
try:
    something()
except SomethingException:
    print("something that wants to be useful")
    raise
```

The above is useless. Instead of print it is better to comment the code and avoid the try / except.

- In my opinion multi-line strings are painful. I prefer to use msg += line especially when I end up with a multi-line statement like raise ProgrammingError(msg)
- mutate_node is almost two page long, with a very important code at the end. It would be easier to read such as:

```
def mutate_code(node, context):
    context.stack.append(node)
    try:
        maybe_mutate_code(node, context)
    finally:
    contexte.stack.pop()
```

An even better pattern is to use a context manager.

- Config vs. Context ?!

`__main__.py`

That is the cli definition with the library called click that is not my favorite library, I believe it is better to keep thing simple hence I rely on docopt that does less magic (!) and gives you more control (also, docoptdoes display all options). Interface are complex topic, and I have no definitive answer regarding cli.

`loader.py`

install will create a class object on the fly without directly relying on type with top-level functions passed as arguments. That is a performance optimization, but when the time of execution is several days, most milliseconds matters.

Relative imports are difficult to read.

cosmic-ray

Next I looked at cosmic-ray mostly because there was an exchange between cosmic-ray's maintainer and mutmut's maintainer and I wanted to see by myself what was the problem. I do my review a few month or years after that exchange happened so the situation is different.

Spoiler: I find the code of cosmic-ray better, I disagree that the mutations are not easy to extract and use them independently (except that it requires to dive into openstack libraries, but that ought to be good thing right !

The only thing I disagree with is the fact it rely on Celery (how hard can it be. Celery in that case is not necessary, because it is easier to rely on multiprocessing, also even more so nowadays it is easier to setup and configure a single machine with 20, 40 or even 128 thread cores than the equivalent infrastructure with multiple machines. Also less costly.

On the subject of server costs, it is a perfect time to share the following blog post:

Cerebralab Blog Note: Some details of the stories in this article are slightly altered to protect the privacy of the companies I worked for It's somewhat anecdotal, but in my work, I often encounter projects that seem to use highly inefficient infrastructure providers, from a cost perspective. https://cerebralab.com/Is_a_billion-dollar_worth_of_server_lying_on_the_ground

There is some interesting library in the requirements like yattag which is not my favorite in-python html templating library but still an interesting take, also stevedore should be the subject of follow up review!

The code is rather short with 2196 python lines of code. The code look visually nice, and is well commented. It use log as the variable name that holds the python logger, hence I am not the only one to do that.

There is a few mistakes here and there, but the overall code is good!

I recommend to read cosmic-ray code if you are getting started with Python!

Others

I did not have time to review the following projects:

- EvanKepner/mutatest Are you confident in your tests? Try out mutatest and see if your tests will detect small modifications (mutations) in the code. Surviving mutations represent subtle changes that are undetectable by your tests. These mutants are potential modifications in source code that continuous integration checks would miss.

- mutpy/mutpy MutPy is a mutation testing tool for Python 3.x source code - mutpy/mutpy!

Rework

- Unlike the maintainer of mutmut I think that parallel testing is a requirement for this kind of tool.
- Unlike cosmic-ray's maintainer I think Celery is overkill (mind the fact that Celery is an add-on in cosmic-ray)
- Deterministic behaviors are a good thing, and mutmut seems to miss that.
- mutmut seems to rely on sampling, but there is no way to control it.
- Again the process to test 15k lines of source code takes around 24 hours on my side even with the fork that use a thread-pool. More optimizations? Yes, maybe, but more importantly, it would be nice to be able to have a look at the results while the process is running with a cli or better with a feature creep web interface.

Overall I am happy with the result, except the following:

- I could use multiple module files especially for the class describing the mutations.
- Some functions could use better names.
- I need to replace the use of the imp package.

Last but not least, I need to replace parso with Python 3.9 ast because it produce less noisy mutations.

forge at ~amirouche/mutation. ## 2021-04-01 - NLnet supports Babelia

I am happy to announce that nlnet supports Babelia.

I am very pleased with the financial support, but even more so about the recognition of my past work, I have been working on this for 10 years, and the immediate efforts on Babelia.

That is the occasion to share rationales, the roadmap I devised for 10 months, and some technical notes.

Why a search engine?

TL;DR: I can do better than Google.

Search has been an essential part of knowledge acquisition from the dawn of time. It is even more prevalent today because it is more accessible than ever before. Most people have access to a large amount of knowledge thanks to privateer (sometime wanna-be privacy-friendly) search engines.

Here is what problems my search engine will solve compared to existing search engines:

- Free and open search engine that anybody will be free to study, fork, and run;
- Hence, it will not be under the control of a single (possibly selfish) group. People will be able to tweak it, even without programming knowledge thanks to full control over the crawler, with control over the knowledge base and with the help of the moderation tools;
- It will not be privacy-friendly wann-be: the source code will be available for anybody to check any privacy claim I make;
- It may yield more interesting results;

One thing I will not claim is that it will eliminate the filter bubble. In a way that is a similar situation to the fediverse: a self-hosted or community search engine will entice some kind of filter bubble because of confirmation bias such as "The results of that instance are good, hence all results are good", or such as "I agree with the results, so all results are truthy". Also, because of unknown unknowns. In any case, it will be, like today, the responsability (and duty) of any fediverse citizen (fedizen) to cross-check results, and likely escape the gilded cage they constructed for themself, and their community.

What I can claim is the following: there will be more search engines, with more diversity, and hopefully expert instances that are curated fairly.

I build a search engine because that is apparantly the most difficult software that can be built.

To summarize "why a search engine" boil down to a) the current situation can be improved, b) I can do better, c) I want my own search engine.

Why not an existing FLOSS search engine?

TL;DR: There is none.

Even if some software might qualify as "search engine", the bare minimal requirement is that they should be written in an easy to the mind programming language such as Python or… Scheme.

Also, I have added non-biased constraint features such as easy to setup, run, and maintain, that disqualify all other thinkable software or set of software.

NB: I plan to out-perform solutions based on ElasticSearch such as commonsearch.

NB: Recently, on my side, Google search started to group results for things such as StackOverflow, reddit, Quora and whatnots. Since a couple of weeks, for some reason, Google started to try to present more diverse set of results. Maybe they hope that it will bring back the 1990 era of geocities, or the era

before they killed feed readers, eventually improving organically Google search, because Google algorithms can not figure that almost anything about software outside StackOverflow is subpar, and it particular it can not figure what is a good page. Google wants to tackle the problem of centralization…

NB: At some point, searching for `simhash` was yielding three mirrors of wikipedia's page entitled simhash, those websites were setup by black hats to capture ad clicks.

NB: I wrote down the above story to be able to drop: Google quacks!

Why will I succeed where others have failed?

TL;DR: Maybe I will fail. Maybe I will succeed, I worked a lot for that, and made choices.

First, there is always a chance or evil luck that I will fail.

Outside the very top notch high world-class level of self-esteem ego I have, there are several source of confidence:

- Babelia is not a privateer search engine;
- Babelia does not aim to be a global search engine;
- I did my best to eliminate all source of known unknowns.

Babelia seek to eliminate the need for Google, and down the road even deliver better tooling that any wanna-be unicorn… Babelia does not play by the same rules. Babelia will not consider that the user is stupid. Babelia is FLOSS, that alone will be enough to get rid of Google. It will also be part of the fediverse. And further in the future the basis of a fully decentralized Internet at the application level

NB: By the way, I believe Google's search engine is already built around the idea of a federation, except it happens in a controled environment. So, what Babelia will achieve is more difficult that what Google try to do (and arguably sometime succeed).

NB: I do not know why Cliqz failed or even Qwant did not succeed. I may find out.

Really, why a search engine?

TL;DR: Star-system is the limit, Facebook and GitHub are next.

The search engine is gathering place, like a library. I like to think Google is the modern incarnation of the Library of Alexandria. There is also Wikimedia projects. I mean, having loads of books without a way to find out the book you are interested in is not useful to have.

In my opinion, a search engine is a fundamental piece in a knowledge construction.

What about wikipedia, wikidata or other wiki stuff, or Facebook, or even GitHub? Those things are more social than a search engine: a registry, a filling cabinet can be automated, whereas decyphering mushrooms caps, and categorizing into species, as of yet, can not be automated so much.

Another fundamental piece in knowledge construction is communication.

To improve upon the establishment, and established practices, my goal is to mimick the world wide web distribution model (easy updates, and the long gone view-source), and getting inspiration from GNU/Linux distributions (network of trust), with the far-reaching perfection and/or minimalism dedication of projects such as suckless, netbsd, or… R7RS.

The idea is to build a desktop environment that stems around a decentralized (publicly distributed) code space where you can stream updates from a network of friends. No, Bill Gates is not your friend.

Yes, the basis of the desktop environment will be a networked programming language. It will not be networked in the sense of Erlang.

Unlike anthropocene desktop environment, it will not try to hide programming from you. It should be painless to share your tweaks, scripts or programs into the public network. The best and still wanna-be feature is easier internationalization (i18n) of code. Footnote: Clickports will be available.

Code is a meme. Where you can share codes, you can share memes. The inverse is not always true. I experimented with a peer-to-peer social network with qadom. While, it is incomplete, it allowed me to figure that it is not impossible, there might be hidden corners, tho.

To summarize: Other aspects of the Internet are more social than a search engine. To deliver the features that can compete with Facebook, GitHub while sticking to demonstrated good pratices such as network of trust, the way forward is to build a desktop environement that is built around a programming language that embody, in world made of diversity, the social nature of software making. Being able to share code as easily as sharing cat memes, and being able to download those memes, after a review, and possibly some tweaks, install them in your own desktop is: the next big thing.

Roadmap

Back to the more prosaic roadmap toward the release of Babelia 1.0:

- Milestone 1, Graphical User Interface: The first goal is to build the user interface. For regular users, the mighty input box will be featured with search results (called hits), along a search pad… The operator will be presented how to populate the knowledge base, how to control the crawler, along a dashboard to display health, and sort out moderation requests.
- Milestone 2, Boolean-Keyword Search Engine: The second step is to build

the basis of the backend that can achieve boolean-keyword search with exact-match, and negated keywords, without the support of the operator `OR`. Also, I will create a program to convert `.zim` files from kiwix.org into a SQLite LSM database, add the ability to populate the database with files using the Web ARChive format nicknamed `.warc`. At which point, it will make sense to package Babelia in NixOS.

- Milestone 3, Knowledge Base: in that step the goal is to create the backend that will allow to create the knowledge base that gathers information about known entities and their relatedness. At this point it will be possible to display recognized entities on result page. The main goal being the added possibility to travel to the right of the semantic continuum through hops in relations between known entities and keywords from the query.
- Milestone 4, Crawler: The point of this milestone is to build a crawler (also known as spider). Unlike a privateer spider, it does not aim to be very fast. One of the main goal of Babelia is to be one stop solution for your search need. Hence, the main feature of the crawler is to be well integrated with the rest of search engine. It will be possible to subscribe to RSS, and ATOM feeds. Also, it will be handy, to also support firefox bookmarks. It will be possible to ignore URLs with glob patterns. Also, the number of hops outside seed domains will also be configureable to help escape the gilded cage.
- Milestone 5, Moderation and Dashboard: Here the goal is to allow the operator of a Babelia instance curate domains with the ability to delete indexed documents, or even purge a domain that were flagged by an user.

Architecture

A few notes about the design of the whole thing:

- Milestone 1, the graphical user interface will be built with Gambit Scheme JavaScript backend, preact, and okvslite. I will to try to use Gambit with Termite to help with concurrency.
- Milestone 2, Boolean-Keyword Search Engine: The inverted index will associate words to documents after they are transliterated to ascii. At query time, keywords in the query will be also transliterated to ascii, the least common token will serve as the seed to select the smallest set of documents that contains all the results, those are called candidates. At which point, a non-distributed map-reduce will score document against the query, whie Aho-Corascik is streaming the words from cached candidate documents. Map-reduce will keep top N results, where N might be configurable.
- Milestone 3, Knowledge Base: that will be a port of copernic, `vnstore2` will be built on top of the new extension I invented called `nstore2`. The advantage of `vnstore2` over `nstore2` is that it allows to track changes, and the added benefit of being able to rollback. At query time depending

on the number of bangs `!`, the search engine will follow links between keywords and their related entities, to travel in the semantic continuum possibly yield more interesting results, it might also increase false positives. `vnstore2` will also be useful to add summaries or descriptions to known entities, that could be linked to their official page or wikipedia page.

- Milestone 4, Crawler: Unclear as of yet.
- Milestone 5, Dashboard: It is a Create-Read-Update-Delete graphical interface, the backend code looks boring…

You might ask: I am not a coder, what is in there for me? In that case re-read the above, and send me an email with a precise description of what the above is missing.

NB: I plan to deliver before rustaceans start to consider coding software that are not command-line helpers.

2021-01-09 - noontide: review & rework of loconotion

Loconotion is a python program that allows to generate a static website from a notion.so page.

Review

Loconotion is a great tool to build a website from a notion page. The only user experience problem is the absence of pypi.org package so that you can just `pip install` loconotion and then call loconotion to create a website. Another slight cognitive overhead is the optional configuration file. It is optional, hence not required, but FOMO…

Overall, the code is classic Python code: not evil, but not great either. Let's dive in.

Here is the source code organization:

```
loconotion/
   conditions.py (52 sloc)
   __main__.py (128 sloc)
   notionparser.py (467 sloc)
```

First thing I notice is that it looks small, sloccount reports 647 lines of code, so that could definitely be a single module file. Again sloccount reports 128 lines in `__main__`. I recommend against having code inside `__main__` and `__init__` outside imports or a trivial main function, because I rarely see code in those files, which in turn makes it difficult to remember to check that it is in fact trivial or empty files, and check in particular that it does not contain import time logic. Relying on code executing at import time is notoriously evil as it breaks various tools e.g. pydoc. Not only it breaks essential tooling, but because the code executes at import time, depending on how imports happen, the code

&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.pdf”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PDF”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ai”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.AI”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.png”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PNG”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jpg”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPG”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jpeg”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPEG”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jp2”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JP2”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.jpf”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.JPF”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.bmp”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.BMP”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.PS”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.EPS”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.mps”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.MPS”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.pz”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps.Z”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps.Z”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.ps.gz”
&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024.eps.gz”

&q=85&fm=jpg&crop=entropy&cs=srgb&w=1024”

Figure 8: locomotive

will execute in some order instead of another without having touched the file where the code is, making the behavior of an application or worse a library unpredictable.

Executing code at import time is evil

I do not recommend Flask for that reason. The order of imports change the order used to resolve URL into views which can break an application in unpredictable ways.

Luckily, in the case of loconotion `__main__.py` contains only command line interface logic, it is wrapped inside a function and only executed when `__name__ == "__main__"`, so that is OK

Let's move to `loconotion/conditions.py`. It is a very tiny file with 52 lines of code. I do not create files before refactoring. Predicting whether a file will be useful help for the reader is difficult to do before the code is written. Also it is much easier to navigate a single file than a directory and many small files. A file is not a zero-cost abstraction in terms of cognitive load for the reader or the writer (if there is such a thing such as "zero cognitive cost abstraction"). The writer needs to figure a good name for the file, and the reader must keep track of where objects definitions are located with the supplementary vocabulary that was invented ad-hoc to host in the case of `loconotion.conditions` two classes each of which contains one method called… `__call__`! By the way, whether you use a "code navigator" so called jump-to-definition or just a grep-like tool does not matter: one file with only 600 lines will ALWAYS beat one directory and three files.

YAGNI complex or clever hierarchy, that is premature optimization.

Further reading `loconotion.conditions`, the thing that strikes in this particular case, but that dominates the world wide python mainstream mind-share: `class`. You might disagree with me about the broader Python ecosystem (this is not the last review!). So, let's just focus on this case:

```
class notion_page_loaded(object):
    """An expectation for checking that a notion page has loaded."""

    def __call__(self, driver):
        notion_presence = len(
            driver.find_elements_by_class_name("notion-presence-container")
        )
        if (notion_presence):
            unknown_blocks = len(driver.find_elements_by_class_name("notion-unknown-block"))
            loading_spinners = len(driver.find_elements_by_class_name("loading-spinner"))
            scrollers = driver.find_elements_by_class_name("notion-scroller")
            scrollers_with_children = [];
            for scroller in scrollers:
                children = len(scroller.find_elements_by_tag_name("div"))
```

```
                if children > 0:
                    scrollers_with_children.append(scroller)
            log.debug(
                f"Waiting for page content to load"
                f" (pending blocks: {unknown_blocks},"
                f" loading spinners: {loading_spinners},"
                f" loaded scrollers: {len(scrollers_with_children)} / {len(scrollers)})"
            )
            all_scrollers_loaded = len(scrollers) == len(scrollers_with_children)
            if (all_scrollers_loaded and not unknown_blocks and not loading_spinners):
                return True
            else:
                return False
        else:
            return False
```

What this function does *ahem* I mean to write `class` is: "check whether a page is fully loaded" because notion will load lazily a page and its content, so loconotion need the page to be fully loaded by the headless browser (in the snippet, that is the variable `driver`), before reading the complete html and writing it to a local file. Something is strange. I mean, even if you do not know the semantic of `SomeClass.__call__` a class that inherits nothing and has a single method is "prolly evil".

My first take would be to replace this class with one or more function, it does not loose generality or expressive power.

One thing that I do systematically: first return trivial cases, so that I do not need to think about it while reading the rest of the code, for instance:

```
if value:
    ## ...
    ## something something
    ## ...
    return complexity
else:
    return triviality
```

The above is much more readable as follow:

```
if not value:
    return triviality
## ...
## something something...
## ...
return complexity
```

Do not be fooled by the `not` operator, it does not matter whether the predicate is "reversed", deal with simple cases first. Also, it saves a level of indentation.

The following:

```
if whatever:
    return True
else: return False
```

If you really need a boolean, it is equivalent to:

```
return not not whatever
```

All things considered, the following is fine:

```
return whatever
```

In the second class from the same file condition.py there is constructor method `__init__` which might trick you into thinking that class is useful. Think twice! The class `toggle_block_has_opened` (verbatim snake case), can be rewritten as follow:

```
def toggle_block_has_opened(driver, toggle_block):
    ## ...
    ## something...
    ## ...
```

About `loconotion/notionparser.py`:

- There is single class that is around 600 lines,
- The name of the class is Parser but it does much more,
- Too much indentation, that can be reduced with the previous trick where first the code deal with simple cases,
- Scraping the page which includes loading the complete page should be separated into different functions to make it more testable,
- Recursion is not pythonic.
- There is several small nitpicks to do about e.g. multi-line assignments to multiple variables, and in general multi-line statements: that hurts readability.

The code is well documented and it is easy to figure what happens. Also, it does the job!

Rework

It is not a drop-in replacement for loconotion, and does not support the same features.

- It 280 lines of code
- It use lxml and xpath instead of beautiful soup
- It use httpx instead of requests
- It use pyppeteer instead of selenium
- There is no configuration file

- There is no “nice urls” and some urls are definitly ugly e.g. a page inside a table.

It took me 8 hours

I do not believe the code is perfect, but that is a code base I can work with. There is one thing to change to make really testable: move the code fetching the page outside crawl function. It is not perfect, but that is what I used to use to render this website

The only feature I miss is a feed :)

noontide forge.

2021-02-13 - okvslite

I do not want to spread Fear, Uncertainty and Doubt (FUD) about SRFI-167 (okvs) and SRFI-168 (nstore). Those libraries are useful and can be used as demonstrated in guile-babelia, and guile-nomunofu. They can also be improved. In this article, I try to tackle problems specific to SRFI-167 aka. okvs.

Engines

The engine procedures were supposed to abstract the underlying implementation to be able to swap implementation hence storage database backend at will in user code.

A better abstraction is generic functions as seen in chibi. The document is little light, but the tests explain very well how it works:

```
(define-generic add)

(define-method (add (x number?) (y number?)) (+ x y))

(define-method (add (x string?) (y string?)) (string-append x y))

(test 4 (add2 2))

(test "22" (add "2" "2"))
```

Another way I would call it, probably is: “predicate-based multi-methods” or “predicate-based multiple dispatch”. See the following article:

Multiple dispatch Multiple dispatch or multimethods is a feature of some programming languages in which a function or method can be dynamically dispatched based on the run-time (dynamic) type or, in the more general case, some other attribute of more than one of its arguments.

In Python, it looks much like an abstract abc class with the added support of multiple dispatch that is more powerful that single dispatch. See python documentation:

- abc - Abstract Base Classes - Python 3.9.1 documentation This module provides the infrastructure for defining abstract base classes (ABCs) in Python, as outlined in PEP 3119 ; see the PEP for why this was added to Python. (See alsoPEP 3141 and the module regarding a type hierarchy for numbers based on ABCs.)
- functools - Higher-order functions and operations on callable objects - Python 3.9.1 documentation Source code: Lib/functools.py The module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module. The module defines the following functions: Simple lightweight unbounded function cache. Sometimes called "memoize".

Also, the dispatch is done with any predicate that is slightly more powerful than Python's isinstance.

So, in okvslite, all engine-fooobar procedure will be generic methods.

pack and unpack

The signature of pack has a problem: (pack . key) -> bytevector?. That is symmetric with (unpack bytevector) -> list?. We can simulate their use with the following code:

```
(assume? (equal? (unpack (apply pack key)) key))
```

That is ok in most case, except the fact that key rest argument will force the creation of a new list in some (most?) scheme implementation, hence stress the garbage collector in the hot path. This is might not be a problem, if they were no easy faster alternatives. In that case, there is a performance trick that is also an enabler. We can change the signature of pack to:

```
(pack obj) -> bytevector
```

Similarly unpack becomes:

```
(unpack bytevector) -> obj
```

Hence the above simulation becomes:

```
(assume? (equal? (unpack (pack key)) key))
```

See that apply disappeared. That is not only slightly faster and memory efficient, and it will add the ability to pass any basic scheme object to pack as top level value. Possibly a vector? instead of a list, and also an atomic value like some number, a boolean.

That will also enable another small optimization, again in the hot path, while reducing the complexity of the code in many cases. For instance, in the case where the value part is an atomic value, previously it was required to pass a list as value, otherwise said, the value was necessarily wrapped inside a list:

```
;; To store 42 as a value before you were required to do the following.
;; Mind the fact that the rest argument,
;; makes it implicit that the value is a list!

(okvs-set! #vu8(C0 FF EE BA D0) (pack 42))

;; Then when you query that key:

(define fortytwo (car (okvs-ref #vu8(C0 FF EE BA D0)))
```

The new code is more readable:

```
;; There is no implicit list!
(okvs-set! #vu8(C0 FF EE BA D0)
(pack 42))
;; Then when you query that key:
(define fortytwo (okvs-ref #vu8(C0 FF EE BA D0))
```

Hence there is less car in the hot path!

okvs-in-transaction

In the SRFI document, I did not explain what is a transaction:

A database transaction wrap operations that will all be applied, otherwise in case of error, none will be applied.

The full signature is:

```
(okvs-in-transaction okvs proc [failure [success [make-state [config]]]])
```

failure and success are similar those used in hash-table-ref. A similar pattern with Python is try / except / else :

```
try:
    out = proc()
except Exception:
    return failure()
else:
    return success(out)
```

The advantage of the Scheme approach is that it makes a stack shuffling aka. exception or non-local exit, optional, hence it makes some optimizations possible (compared to Pythonic code that rely on exceptions in similar cases). Most of the time with Scheme exceptions are opt-in. Also the code is much shorter, and suggest to create a procedure with failure and success which leads to more readable code.

More on okvs-in-transaction: I think I will drop all-around config from all procedures because this is really here to enable some optimizations with wiredtiger, but wiredtiger… make-state is useful but rather obscure, I need to document

more cases that makes use of it (mind the fact that is the last optional argument when config will be gone).

`okvslite-query`

A big change that is coming is to merge okvs-ref and okvs-range to make it explicit that the API is the query Domain Specific Language. So, okvslite-query looks like:

```
(okvslite-query okvslite key [comparator1 comparator2 other [offset [limit]]])
```

A common realization is to query `[a..b[`:

```
(okvslite-query okvslite a '<= '< b)
```

To retrieve a reversed generator, that is from b to a where b is excluded:

```
(okvslite-query okvslite b '< '<= a)
```

Then okvs-remove will have a similar signature.

okvs-range-prefix will be renamed okvs-query-prefix as sugar one-liner helper to avoid to dive into strinc… okvs-query-prefix can be easily expressed with okvs-query when strinc is public and understood:

```
(define (okvs-query-prefix okvslite key-prefix)
  (okvslite-query okvslite key-prefix '<= '< (strinc key-prefix)))
```

Hooks

I think okvs-hook-on-transaction-begin must be called again in the case where okvs-in-transaction rollback to replay the transaction.

Prolly, similarly to make-state, hooks are too advanced, and I do not use them anymore…

2021-01-12 - raxes: review & rework of searx

searx is privacy-respecting metasearch engine. It is search engine that does not have its own index, but rely on other search engines to deliver its results. It rely on Flask, requests and lxml.

Review

According to the Makefile, the entry point of the web application, is ./searx/webapp.py. The main problem with searx is that it is not easy to use it as library.

Looking up @app.route yield only 13 routes/

Let's look through each from least interesting to the most clever, that is approximately a bottom-up read.

/translations.js

There is two small issues with this code:

1. It is easier to debug code when return is followed by a simple variable e.g. return foobar. Here things are made worse, because the returned value span multiple lines.
2. the return is multiple line statement. That requires to zigzag the code. Instead of reading top-down left-right you need to zigzag and reconfigure your brain to also do read down-top (and even right-left in more problematic cases).

/config

The use of underscores _ in variable names is odd. In this case, it is a way to avoid to think and overload the reader with useless and poorly named variables names. I do that a lot with Scheme, but with Scheme you can use more readable and better looking characters for naming variables. underscore is difficult to read, and sometime it disappears because of poor resolution or image scaling. The following code:

```
_engines = []
for name, engine in engines.items():
    something = process(name, engine)
    _engines.append(something)
```

Can be reworked into a list comprehension:

```
engines = [process(name, engine) for (name, engine) in engines.items()]
```

That much more obvious that one can use a list comprehension instead of the following code:

```
_plugins = []
for _ in plugins:
    _plugins.append({'name': _.name, 'enabled': _.default_on})
```

Mind the use of _ as variable name. When used as variable name placeholder, _ should not be accessed. It would be trivial to replace _ with item and bundle it inside a list comprehension.

/favicon.ico

That is a complex one-liner:

```
@app.route('/favicon.ico') def favicon():
    return send_from_directory(
        os.path.join(app.root_path, static_path, 'themes', get_current_theme_name(), 'img'),
        'favicon.png',
```

```
        mimetype='image/vnd.microsoft.icon'
    )
```

I will not repeat that multi-line return statements are difficult to read.

In that case, the code is trivial enough, but nesting calls is a evil habit.

Things like:

```
out = qwe(asd(zxc(iop)), jkl(bnm))
```

Are not only difficult to read (even with normal variable names) but also more complex to debug, because the intermediate result does not get their own variable.

/opensearch.xml

That is nice code. I like the following pattern that is used twice in that function:

```
out = default_value if not nominal: out = something
```

It is odd to see the HTTP method spelled lower case but that might be something specific to opensearch. Otherwise I try to avoid shortcut variables names in that function ret and response could both be renamed out since it is the output of the function.

/robots.txt

Not very interesting, except maybe it would be easier to create a global constant (and in some case, but prolly not here, use an external text file).

/stats/errors

Again, factoring the body of the for and using a list comprehension could be nice.

The following code:

```
foobar = list(something.keys()) foobar.sort()
```

That is equivalent to:

```
foobar = sorted(something.keys())
```

When you use the key keyword argument in list.sort, sorted or others, you might want to rely on the standard library operators.

/stats

Nothing interesting to say about this function.

/image_proxy

The following code:

```
headers = dict_subset(request.headers, {'If-Modified-Since', 'If-None-Match'})
```

Can be rewritten to be more readable with a **single-line** dictionary comprehension:

```
headers = {key: headers[key] for key in headers if key in ['foo', 'bar']}
```

I still do not understand why everybody use the variable name logger.

The following:

```
counter = 0 for item in foobar: counter += 1 do_something(counter, item)
```

Can be rewritten:

```
for counter, item in enumrate(foobar): do_something(counter, item)
```

enumerate is builtin function, hence always available.

/preferences

I will not repeat what I already wrote about comprehensions. There is good pattern in there. But the last line of the function kills everything.

/autocompleter

An idea here about how to avoid to use a class and sadly hide the interesting logic away from the where the action happens:

RawTextQuery return an object with a public interface that has 4 methods include some that have side-effects like the following call to changeQuery:

```
for result in raw_results:
    raw_text_query.changeQuery(result) ## add parsed result
    results.append(raw_text_query.getFullQuery())
```

That is necessarily a side-effect because it does not assign a variable otherwise it would be dead-code. Here raw_text_query is mutated, mutable objects are difficult to debug.

Following the spirit of the code, the function searx_bang could be a method of RawTextQuery especially since it is used only once in the whole code base.

Anyway the full function could rewritten as follow:

```
def autocompleter():
    """Return autocompleter results"""
    query = request.form.get('q', '')
    try:
        query, parts, languages, specific = query_parse(query)
```

```
    except BadQuery:
        return '', 400

    ## parse searx specific autocompleter results like !bang
    hits = searx_bang(query, parts, languages, specific)

    backend_name = request.preferences.get_value('autocomplete')
    try:
        completer = autocomplete_backends[backend_name]
    except KeyError:
        pass
    else:
        if not hits and (len(parts) > 1 or (len(languages) == 0 and not specific)):
            ## get language from cookie
            language = request.preferences.get_value('language')
            language = language.split('-')[0] if (language or language == 'all') else 'en'
            ## run autocompletion
            more = completer(query, language)
            hits += more

    ## parse results (write :language and !engine back to result string)
    hits = [do_something(hit) for hit in hits]

    response = hits_to_response(request, hits)
    return reponse
```

Mind the fact:

1. I dropped the disabled_engine variable, because I am not sure where it is useful.
2. It is not clear what happens in the last for loop especially with the mutated raw query, so I replaced the whole thing with a comprehension and factored the body in a function called do_something.

But that is not everything we can do. To make the project usable as a library, it will be nice to extract the logic and keep environment specifics in the flask view.

If you do it yourself, you might end up with something like that as the view function autocompleter:

```
@app.route('/autocompleter', methods=['GET', 'POST'])
def autocompleter():
    """Return autocompleter results"""
    query = request.form.get('q', '')

    if query.isspace():
        return 'thanks, but no thanks!', 400
```

```
    backend_name = request.preferences.get_value('autocomplete')
    language = request.preferences.get_value('language')
    language = language.split('-')[0] if (language or language == 'all') else 'en'

    try:
        hits = autocomplete(query, backend_name, language)
    except BadQuery:
        return 'invalid query because...', 400

    response = hits_to_response(request, hits)
    return reponse
```

/about

It is a static page so not much to say, except the indentation is not good.

/search

That is the gist of the project. Here is the big problem: validation, logic and rendering is mixed into a giant view function, factorization was done, but there is room for more especially regarding the output generation.

Here is the code that execute the meta search:

```
## search
search_query = None
raw_text_query = None
result_container = None
try:
    search_query, raw_text_query, _, _ = get_search_query_from_webapp(request.preferences, r
    ## search = Search(search_query) ##  without plugins
    search = SearchWithPlugins(search_query, request.user_plugins, request)
    result_container = search.search()
except SearxParameterException as e:
    logger.exception('search error: SearxParameterException')
    return index_error(output_format, e.message), 400
except Exception as e:
    logger.exception('search error')
    return index_error(output_format, gettext('search error')), 500
```

There is no point into defining as None the first three variables since it is useless without an actual result, the code that follow expect something that is not None.

Something that is always odd: multiples statements inside the try block.

SearchWithPlugins is used only once in the whole code base and the only public method is search. That begs to become a function, it will make clear how do a search!

How meta search works wit searx?

After jumping around definitions I end in the class Search at the method search_multiple_requests:

```
    def search_multiple_requests(self, requests):
        search_id = uuid4().__str__()

        for engine_name, query, request_params in requests:
            th = threading.Thread(
                target=processors[engine_name].search,
                args=(query, request_params, self.result_container, self.start_time, self.ac
                name=search_id,
            )
            th._timeout = False
            th._engine_name = engine_name
            th.start()

        for th in threading.enumerate():
            if th.name == search_id:
                remaining_time = max(0.0, self.actual_timeout - (time() - self.start_time))
                th.join(remaining_time)
                if th.is_alive():
                    th._timeout = True
                    self.result_container.add_unresponsive_engine(th._engine_name, 'timeout'
                    logger.warning('engine timeout: {0}'.format(th._engine_name))
```

It does trigger simultaneously, using threads, a search query against a search engine based on user preference and the last `for` block will retrieve the result under a timeout. That is if a search engine does not reply under less that some configured time, it is considered an error.

The part that interest me is:

`target=processors[engine_name].search,`

After some jumping around, I find the mighty directory `searx/engines` that contains all the logic to query and scrape results.

Rework

This is a very quick rework of searx, it is very far from as feature parity.

forge.

2021-09-02 - Let there be binary executables

I released ruse-exe. A Chez program that allows to create binary executables from Scheme programs.

Here is in full the usage documentation:

```
Usage:

  ruse-exe [--dev] [--petite] [--optimize-level=0-3] [EXTENSIONS-OR-DIRECTORIES ...] progra

Given a Scheme program inside the file `program.scm`, produce a
standalone executable inside the current directory called `a.out`.
ruse-exe will look for the necessary files inside
`EXTENSIONS-OR-DIRECTORIES`.  The arguments following two dashed
`ARGS` will passed to the underlying C compiler.

The flag --dev will enable generate allocation and instruction count.

The flag --petite will only build with petite scheme.

The arguments `EXTENSIONS-OR-DIRECTORIES` will replace the default
extensions, source and library directories. If you want to compile a
project that has both `.ss` and `.scm` with libraries in the current
directory, you need to use something along the line of:

  ruse-exe .ss .scm . program.scm

Homepage: http://letloop.xyz
```

You can grab a ready to use on ubuntu 21.04 binary release at github.

2021-05-18 - Ruse Scheme shall be

Ruse Scheme, formely known as Arew Scheme, is at this stage, a collection of Scheme libraries for Chez Scheme. There is a grand scheme plan plot machination for it. Read on.

What is a civilization kit?

A civilization kit is a software or set of software that ease the organization of life. So far, there is really one civkit that is mostly privateer that includes and is not limited to:

- Wikimedia project such as Wikipedia, Wikidata, Wiktionary…
- Google, Facebook, Github, Instagram, Twitter, Reddit, StackOverflow, Quora…
- Android, iOS, Firefox, Chrome..
- MacOS, FreeBSD, NetBSD, Windows, Debian, Fedora, Ubuntu…
- Mastodon, and other projects that rely on activitypub…

And many more… that is only the visible part of Earth software system. They are software that aim to ease the the production of software or hardware. They

are also software that helps with governance, provide tools to ease law making process, sustain production chain of food, energy, medecine, culture, education…

They are a lot of software, and that collection form a civkit.

Is Ruse Scheme a new Scheme?

Yes, and no. It depends what is meant by a new Scheme.

Sometime a Scheme is a software that gathers many Scheme libraries, and rely on existing Scheme to execute their code. That is the case of Ruse.

Most of the time, a Scheme is a software that interpret and/or compile a lot of parentheses that is more or less compatible with RnRS. In this regard, Ruse is a Scheme, but it is not completly new. It rely on Chez Scheme to produce executables that can be run on a server or a desktop. Ruse will support Web Assembly and JavaScript to run Scheme in a Web browser.

Some Scheme implementation do a little of both, and also deliver features that go beyond past or current RnRS. Ruse does that, and shall reach beyond…

The main difference with existing Scheme implementations is not found at the programming language level. Ruse is and will stay a Scheme.

The main area Ruse try to innovate is the rest: whether it is the the production or sharing of code, Ruse aim to make it easier than sharing a meme. Another area Ruse try to innovate is to state upfront the scope of the project.

What are the short term goal of Ruse Scheme?

The short term goal of Ruse Scheme is to build a scalable search engine: Babelia. Babelia will both scale-up and scale-down in terms of required hardware. In other words, it may run in the cloud or on a Raspberry Pi.

That first milestone will demonstrate how to build a distributed Von Neumann architecture that aim to be easier to work with than current approaches.

This is the first milestone because it is easier than going fully dencentralized first. It will deliver the necessary tools to work with the current Internet.

The plan is to deliver Babelia in 2022.

What is the next Internet?

The next Internet is an Internet that is more open, more decentralized, more accessible, and resting upon the fundamental principle.

What is the distributed Von Neumann architecture?

The distributed Von Neumann architecture is like a regular computer that rely on multiple commodity computers.

It is different from a compute grid, because it is not meant only for batch processing.

In Babelia, that distributed computer has volatile memory, non-volatile memory, possibly vectors or graphics processing units, and generic computation units.

The goal is to make it easier to build software inside a trusted network of computers.

What are the mid term goals of Ruse Scheme?

Mid term goals of Ruse Scheme are three folds:

- Offer enough tooling to make it easier to create, sell and make a living by producing Scheme code. This includes making it painless to interop with existing software.
- Implement a package manager inspired from Nix, and backed up by content-addressable code that can be translated into multiple natural languages with the help of a decentralized peer-to-peer network.
- Explore a new operating system desktop paradigm resting upon the fundamental principle.

What is the goal of Ruse Scheme?

The goal of Ruse Scheme is to build a coherant bootstrapable whole-stack civkit for a sustainable civilization, resting upon the fundamental principle.

What is whole-stack?

Whole-stack build upon the full-stack concept to include programming databases, and kernels.

What is Ruse Scheme license?

Ruse Scheme is licensed under the Cooperative Non-violent Public License without exceptions.

What is the fundamental principle?

> If a system must serve the creative spirit, it must be entirely comprehensible by a single individual.

2021-03-15 - Versioned generic tuple store 2

A few years back, I set myself the task to create a versioned database. I did not come up with that idea myself! Several readings and professional experiences, lead me to think that was a good idea:

- AuditTrail - Django As raised in a recent discussion on django-developers, this code is one solution for creating an audit trail for a given model. This is working in multiple production sites, but is still incomplete. See Caveats below for more information. The code below requires an SVN checkout as of r8223 or later.
- Wikidata Wikidata is a free and open knowledge base that can be read and edited by both humans and machines. Wikidata acts as central storage for the structured data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wiktionary, Wikisource, and others. Wikidata also provides support to many other sites and services beyond just Wikimedia projects!
- Collaborative Open Data versioning: a pragmatic approach using Linked Data - CORE By Lorenzo Canova, Simone Basso, Raimondo Iemma and Federico Morando Most Open Government Data initiatives are centralised and unidirectional (i.e., they release data dumps in CSV or PDF format). Hence for non trivial applications reusers make copies of the government datasets to curate their local data copy.

And most recently:

- A More Human Approach To Databases End-user databases are all the buzz these days - Notion, Airtable, Coda, Roam, etc. These products have made it possible for people to model information in a way that feels more natural and intuitive to the way we experience it in our daily lives. https://ccorcos.github.io/filing-cabinets/
- Design of system for pending approval and history I am looking for some insight on how to design a solution that handles both pending changes as well as a history of an entire entity. I have found several examples of how to handle this for a single entity object, but I am unsure how to apply this to a object that can contain several “attached” entities.)

I do not claim my approach is bullet proof to every use-case possible. As its name imply it is a generic solution to implement versioned database. What the title does not say, is that it support change request mechanic similar to github pull-requests.

When I started this adventure, the index factor was 120 times the size of the raw data. Given this factor whether the data is text or bytes does not matter: one gigabyte times 120 is 120 gigabytes, a lot.

In 2019, I reduced the factor to 10, which is still a lot given at the time, wikidata was 3 terabytes without change history.

In 2020, thanks to FoundationDB HighContentionAllocator, I managed to store wikidata lexemes in less space that the textual format. You read it very well. It requires less space to store the textual data, than store it inside a database with versioning enabled, and querying possible in timely manner.

Today, I devised a plan to reduce further the space requirement with little or no visible feature difference. That further reduce the space requirement by almost a half.

To summarize: I started with 120 time the size of the data, and today the versioned and query-able data requires at most 0.6 times the size of the original data.

How?

Remember the nstore? It is a generalization of triple stores where the number of tuple items can be any integer. I used that to drop from 120 to 10. The nstore stored the whole tuple in the key of the okvs. (That may look like a problem because keys can not grow big, but in practice since they go through the HighContentionAllocator it is not a problem). The value was empty!

What I used to think is that when I am required to expose an n versioned tuple store, I needed n+2 generic tuple store:

> amirouche/copernic Versioned structured data, with change-request mechanic, at scale. - amirouche/copernic

That is not the case. I can drop the alive? flag from the tuple item and put it in the value part. That way it reduce the number of items in a tuple to 4 and according to make_indices that requires only 6 permutations to be able to query any pattern in one hop.

alive? is accessed twice in the current code base. Mind the fact that in both case alive?.

> amirouche/copernic Versioned structured data, with change-request mechanic, at scale. - amirouche/copernic

The code says something along the line of:

1. Given a tuple items,
2. Lookup all the history of that items and retrieve their changeid and alive? status
3. For each of such history item, keep the status alive? of the tuple with the biggest significance

The returned value called in the above snippet `found` tells whether items is alive at the latest version of the database.

Another case is the VNStore.FROM does a more general query that try to bind some patterns against the latest version of the data:

> amirouche/copernic Versioned structured data, with change-request mechanic, at scale. - amirouche/copernic

What the code says is something along the line:

1. For a given pattern
2. From the versioned tuples, fetch all bindings that match the pattern and include the alive? and changeid (the latter being useless in that particular method, but since it is unknown, it can not be provided by upstream).
3. If the binding is dead aka. `not bindings['alive?']`, then the binding is not valid in all cases according to the latest version (so, instead of a variable alive?, we could query with True… see below)
4. Otherwise, if the binding is alive, we check that it is alive at the latest version. Here lies a bug: if in history there is several times the same items, that are alive but introduced in different change, it will yield multiple bindings that are the same. Instead of:

```
self.ask(*bind(pattern, binding)
```

It should be:

```
self.latest(bind(pattern, binding), alive=True, changeid=binding["changeid"])
```

Since we are only interested in latest version bindings.

Anyway, my point was that alive? is always a variable (except if we change that in the third bullet). Also, as part of time traveling queries, it seems to me rare to query on alive? having a particular value except when asking for:

> How many times a particular tuple was added or removed

In the case of wikidata, where changes are curated, it seems extremely unlikely that the same tuple items will be added and removed maybe times. Here, there is a choice to be made between CPU time and disk requirement.

Indeed, I might trade half the space requirements with some CPU time in some rare cases. Hence the advent of nstore2.

2021-01-31 - Why I still Scheme (and you should too!)

That is a reply to the following post on medium:

> Why I still Lisp (and you should too!) As a long-time user (and active proponent) of Scheme/Common Lisp/Racket, I sometimes get asked why Istick with them. Fortunately, I have always headed up my own engineering orgs, so I have never had to justify it to "management", but there's the even more important constituency of my own engineering colleagues who have never ever had the pleasure of using these languages.

I like the punchline of the post:

> An s-expression based, dynamically typed, mostly functional, call-by-value -calculus based language

I fully agree with that quote. I disagree with the use of medium, and they are few facts that are not accurate.

-calculus based language.

First, like some one else noted in the comments, the JVM (will) support TCO and call/cc with the loom project.

Second, the biggest advantage of -calculus is that everything can be explained in terms of -calculus. I am not familiar with -calculus, what I understand is that one can explain every single form of Scheme with a transformation of the code into some `lambda`. A `lambda` is some kind of anonymous function, it looks more like ES5 function but with explicit rest arguments. That is true for every Scheme forms except `if` and `set!`. Why should you care? Because the immediate consequence is that it is very easy to explain how the programming language works. In fact, you only need to know about `lambda`, `if` and `set!`. The latter allows to change what a variable is holding (from mutating a data structure).

I am not sure -calculus helps to "play back the code" that is more like an argument of side-effect free functions.

The consequence that everything with scheme can be expressed with lambda is that it is much more clear how variable scoping works (unlike python scoping).

I fully agree with that:

> As you might have noticed, I'm not using the word functional language to describe Scheme. That's because while it is primarily functional, it does not skew all the way to non-mutability. As much as it discourages its use, Scheme recognizes that there are genuine contexts where there may be a use for mutations and it permits it without the artifice of auxiliary devices.

But that is not feature of -calculus but really a choice during the design of Scheme.

Call-by-value

I am not familiar with the "call-by-value". Reading through that section of the blog post the alternative is lazy evaluation with Haskell. With Scheme, lazy evaluation is opt-in as explained in the last paragraph of that section.

Mostly functional

> Functional programming is great. Playing back functional code in your head is simple; the code is easy to read and the lack of mutations reassuring. Except when it isn't enough.

To be precise, the lack of mutations is reassuring, except when it isn't enough, Scheme allows mutations.

Dynamically Typed

The world today is going on and on about typed languages.

There is something that is not precise: The world today is going on and on about statically typed languages. The key word is "statically". Scheme is typed. A lot of people boo dynamically languages with the argument Python or Scheme are untyped: wrong. Also, people "dynamic" other stuff e.g. introspection… Or "dynamic is slow" more on that below.

I may or may not disagree with the line "static type checker are bullshit, because they do not prevent anything more than obvious bugs". Again it is not accurate to spread the idea that the "compiler" or "static typing" possibly help avoid even minor bugs, because a) it is not necessarily a compiler (mypy anyone?) and also because static typing as in "required written types near some variables" is not what mypy or even rust compiler use to find some problems. What is significant is type information. And that can come from static typing with the help of Hindley-Milner algorithm or some other tool and heuristics.

I agree with the arguments on invariant, that leads to a better take on a priori checks of programs or even runtime checks: assert. That is the best of all worlds, you can get type information and also check invariant using predicates (predicates that may be registered in the tool doing the checks or possibly inferred with partial evaluation).

The author goes the route of "no tool to check my code", I prefer the tool to be opt-in like assert.

> What improves (and somewhat guarantees) software quality is rigorous **meticulous* testing.

The absence of proof is not the proof of the absence.

Another mistake: type or constraint or contract information does not necessarily translate into fast code!

S-expression based

> The biggest advantage of this form of syntax is a form of minimalism

I prefer "powerful minimalism" because it allows to create a macro system very easily.

Macros also exist in programming languages that are not based on lispy s-expression.

The examples of good use of macros are wrong.

Conclusion

Comes the paragraph on performance, it is prolly about CPU performance:

> And lastly, there's the issue of performance. First, let's put the common misconception to rest: Lisp is not an interpreted language. It is not slow, and all implementations come with lots and lots of levers to tweak performance for most programs. In some cases the programs might need assistance from faster languages like C and C++ because they are closer to the hardware, but with faster hardware, even that difference is becoming irrelevant. These languages are perfectly fine choices for production quality code, and probably more stable than most other choices out there due to decades of work that has gone into them.

Performance is not an issue (not anymore, and certainly not compared to JavaScript, Python or Ruby) especially the biggest LISP implementations (SBCL, CCL, Chez Scheme, or Gambit).

> implementations come with lots and lots of levers to tweak performance

That is not the case of Chez Scheme, and it is still very fast compared to Python or Java or even C.

> I do recognize, learning Scheme/Lisp/Racket is a wee bit harder than learning Python (but a lot easier than learning Java/JavaScript).

That is completely false. The number of things you need to ignore in Python when getting started with programming out number the total count of concepts that expert Scheme engineer need to know to be considered expert. The only problem with Scheme, in particular Chez Scheme is merely the absence of battle tested libraries for everyday use like HTTP, image processing or browser automation and very specific but mainstream libraries.

> the beauty of these languages

Beauty is neat. Good tool for the job is better…