Computing Assignment 1

Due Date: March 30, 2020, 5:00pm

Note. Please note that all computing assignments are group assignments. Further note that for the grading we will apply a "10%" rule, i.e. the maximum number of points for this assignment is 165, but 150 will be counted as 100%. Points that exceed 150 will be stored in a separate counter and used later for compensation of lost points in other assignments or (if not used up this way) the final exam.

The task of this computing assignment is to efficiently implement (in **Python**) a

> **two-stack abstract machine (2SAM) for processing query requests on a list-based object store**.

In the following we describe the object store, the two stacks used by the machine and the queries that are to be realised. The implementation must be generic in the sense that it can be used for any collection of lists in the object store and any well-formed query that can be executed on these lists.

Design decisions such as which data structure to exploit for the lists and stacks used are left deliberately open. It is also possible to use different syntactic conventions for the queries that are used as input for the abstract machine.

Points will be given for the correctness of the implementation (75), its efficiency (50), and the documentation of tests on meaningful data (40). Points will be deducted for incompleteness (parts of the specification have been left out), errors in conception and implementation, and insufficient evidence of correctness.

total points: 165

1 Object Store and Abstract Storage Objects

The **object store** is to contain a finite, non-empty collection of named lists $L_1, \ldots, L_k$. All these lists shall be structured lists, i.e. if the type of elements of list L_i is T_i, then all elements in T_i shall be records of a fixed type $(A_0 : U_0, A_1 : U_1, \ldots, A_{k_i} : U_{k_i})$. Here A_0 is to be defined as a key, i.e. no two records in the list L_i can have the same value of the A_0 attribute.

For abstract storage objects we assume a set I of *identifiers*, a set N of *external names*, and a set V of *values* with $I \cap V = \emptyset$.

A *simple* ***storage object*** is a triple (i, n, v) with $i \in I$, $n \in N$ and $v \in V$. A *complex* ***storage object*** is a triple (i, n, v) with $i \in I$, $n \in N$ and a non-empty set v of storage objects. In both cases we call i the *identifier* of the storage object (i, n, v).

Example 1. Let us take four lists L_1 = Theatre, L_2 = Performance, L_3 = Play and L_4 = Nationality. Let the attributes of the record types T_i be {key, cinema, address} for Theatre, {key, cinema, title, date} for Performance, {key, title, director} for Play, and {key, director, country} for Nationality. Then we can define the following storage objects:

(i_1, THEATRE, {(i_2, cinema, Abaton), (i_3, address, Grindle Alley)})
(i_4, THEATRE, {(i_5, cinema, Flora), (i_6, address, Old Village)})
(i_7, THEATRE, {(i_8, cinema, Holi)})

(i_{10}, PERFORMANCE, {(i_{11}, cinema, Flora), (i_{12}, title, The Piano), (i_{13}, date, May 7)})
(i_{14}, PERFORMANCE, {(i_{15}, cinema, Holi), (i_{16}, title, Manhattan)})

(i_{20}, PLAY, {(i_{21}, title, The Piano), (i_{22}, director, Campio)})
(i_{23}, PLAY, {(i_{24}, title, Manhattan), (i_{25}, director, Allen)})

(i_{30}, NATIONALITY, {(i_{31}, director, Campio), (i_{32}, country, USA)})
(i_{33}, NATIONALITY, {(i_{34}, director, Allen), (i_{35}, country, USA)})

Here, both the list and attribute names are used as external names. To capture multiple elements of a list, the list name can be used several times in abstract storage objects.

2 Environment and Result Stacks

The abstract machine uses two stacks:

ENV denotes the *environment stack* that is used to capture bindings of variables.
RES denotes the *result stack* that is to capture intermediate and final results.

Elements in **ENV** are finite sets of identifiers $S \subseteq I$ (represented by lists without duplicate elements), and for each $i \in S$ there must exist a storage object with identifier i. Initially, **ENV** contains just one element with identifiers of all storage objects corresponding to the records in the object store.

EXAMPLE 2. Continuing the previous example, **ENV** contains initially just one list $[i_1, i_4, i_7, i_{10}, i_{14}, i_{20}, i_{23}, i_{30}, i_{33}]$.

Then the update operations on **ENV** are just *pop* and *push*(S). In addition, we can search the stack from top to bottom for those identifiers of storage objects bound to a given name $n \in N$. This operation shall be call *search*(n).

Elements in the result stack **RES** can be arbitrary including structured lists or lists of tuples of identifiers that can be subject to dereferencing.

3 Queries and Algebraic Operations on 2SAM

Queries are defined inductively as follows:

- Each list name L_i in the object store is a query;
- If Q is a query and φ is a selection formula applicable to the result of Q, then Q *where* φ is a query;
- If Q is a query and A is an attribute of the result of Q, then $Q.A$ is a query;

– For queries Q_1 and Q_2 the cartesian product $Q_1 \times Q_2$ is a query;
– Nothing else is a query.

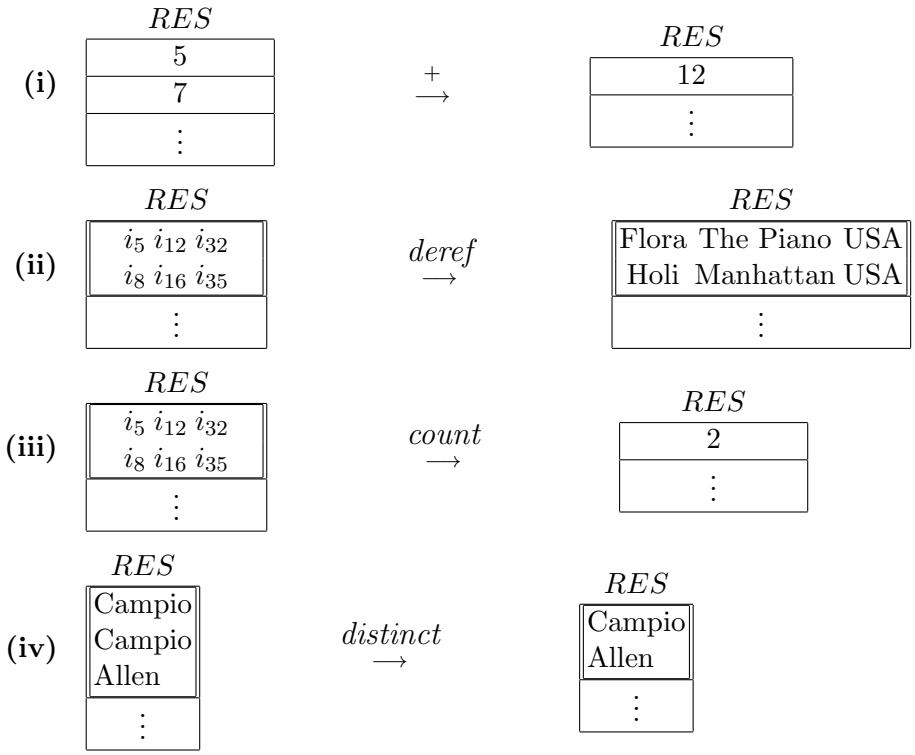
Note that an *equi-join* as used in Lab/Discussion 1 can be expressed in the form $(Q_1 \times Q_2)$ where φ—for efficiency reason it is better to treat equi-joins as one query operation rather han first building a cartesian product and then processing a selection.

In addition, we poermit the following *algebraic operators:*

– aggregation operators such as *count, sum, min, max* and *average*;
– an operation *distinct* for removing duplicates;
– comparison operators such as $<$, $>$, $\leq$, $\geq$, $=$ and *neq*;
– arithmetic operators such as $+, -, *, /$;
– Boolean operators such as *and*, *or* and *not*;
– set/list operators such as *Exists*, *In* and *Contains*;
– a derefering operator *deref* used to eplace an identifier by a name and value.

Then we can build general *algebraic requests*, which take either the form ΘQ or $Q_1 \Theta Q_2$ with an algebraic operator Θ. A key difference between algebraic requests and queries is that the former ones are evaluated without using the environment stack **ENV**, whereas the latter ones exploit **ENV**. Algebraic requests and queries together define all *query requests*.

Example 3. Here are a few examples how to use the algebraic operators:



Clearly the processing of any query request requires first to evaluate the subqueries (or algebraic requests) and then apply the operator in the expression.

4 Query Request Evaluation Examples

For an algebraic request $Q_1 \Theta Q_2$ we first evaluate Q_1, then Q_2, which adds two new elements on **RES**. Then result := $pop_{\mathbf{RES}}$ followed by result := $eval_\Theta(pop_{\mathbf{RES}}, \text{result})$ and $push_{\mathbf{RES}}(\text{result})$ completes the evaluation. For algebraic requests of the form ΘQ we proceed analogously.

We now provide examples for the evaluation of the different types of queries. Here we need the fllowing definition:

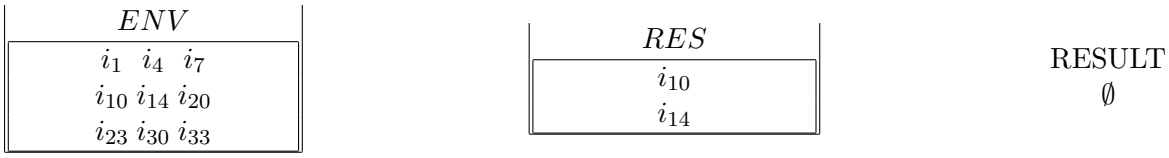
$$nested(i) = \begin{cases} \emptyset & \text{if } v \in V \text{ in the storage object } (i, n, v) \\ \{i_1, \ldots, i_k\} & \text{if } v = \{(i_1, n_1, v_1), \ldots, (i_k, n_k, v_k)\} \text{ in the storage object } (i, n, v) \end{cases}$$

Example 4. Continuing our previous examples consider the query Performance. This is an external name in N, so all identifiers bound to this name are placed in a new list on **RES**:

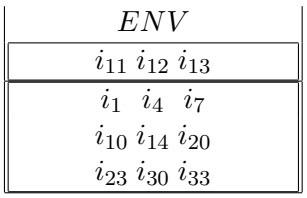
The result can be further processed, e.g. dereferenced.

For the evaluation of a selection query Q *where* φ we iterate over the elements in the list in $top_{\mathbf{RES}}$ resulting from the evaluation of Q first. For an identifier t in this list push $nested(t)$ onto **ENV**, evaluate φ and if true, insert t into result, which will finally be pushed onto **RES**.

Example 5. Consider the selection query Performance *where* cinema = Flora. After performing the elementary query Performance, we have the following state:

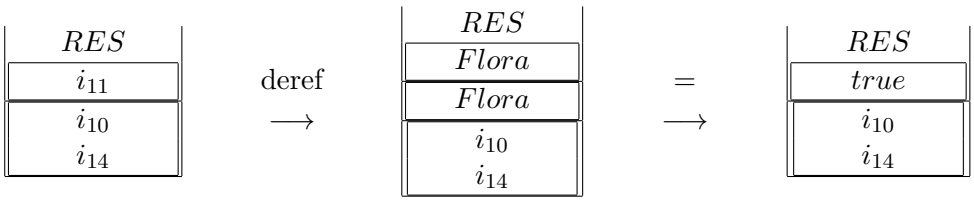


Running through the loop we first execute $push_{\mathbf{ENV}}(nested(i_{10}))$ with the following result:

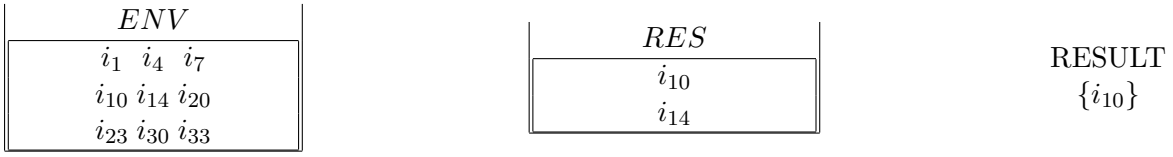


The following evaluation of eval(cinema = Flora) moves i_{11} to **RES**, as the external name is cinema, derefences it to yield Flora, pushes the selection term Flora also onto **RES**, and

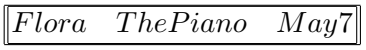
finally evaluates the equality:



The next runs through the loop are executed analogously



resulting in **ENV** as before, while **RES** contains the list $[i_{10}]$. Multiple dereferencing yields the final result



Products $Q_1 \times Q_2$ and projection queries $Q.A$ are handled analogously, but in the latter case instead of adding t to the result, we take $top_{\mathbf{RES}}$, the result of the projection operator.

EXAMPLE 6. Let Q be the query from the previous example, and consider the query Q.title. As we have seen, the result of Q is

unning once through the loop with the projection we obtain result = $\{i_{12}\}$, so we finally get