Institut National Universitaire Champollion

L3 Info
Algorithmes, types de données et preuves

Semestre 2
Année 2024 - 2025

Projet : Base de données orientée graphe

1 Contexte

1.1 Motivation

Le but de ce projet est de concevoir une base de donnée orientée graphes (BDG) minimaliste. Les BDG sont de extensions de bases de données relationnelles qui permettent de représenter plus naturellement des données en forme de graphe qui apparaissent dans des applications de transport (réseaux routier), sciences (interactions cellulaires en biologie), communication (internet, réseaux sociaux) ....

Ce projet se focalise sur l'aspect "langage de programmation" qui fait intervenir des questions d'analyse syntaxique, de typage, de "compilation" d'un langage source vers un langage plus élémentaire, et une sémantique d'exécution qui permet de calculer les résultat d'une requête. L'intention n'est pas d'optimiser l'exécution de requêtes ou de manipuler de très gros volumes de données.

Tandis que SQL est le langage de requêtes des BD relationnelles, une norme internationale ISO récente, GQL (Graph Query Language) [1], définit le langage de requêtes pour des BDG. La norme ISO n'est pas publiquement accessible, mais il y a des publications [FGG+18, GGL+19] sur Cypher, très similaire, qui est le langage du système commercial Neo4j. Le langage MiniGQL développé dans ce projet est inspiré de ces langages mais très fortement simplifié.

Les étapes essentielles du projet sont :

- Mise à jour et extension de l'analyse lexicale et syntaxique du langage MiniGQL, à l'aide des outils *ocamllex* et *menhir* (sect. 3.3).
- Traduction du langage source vers un langage d'instructions plus simple (sect. 3.4).
- Vérification des types du langage source (sect. 3.5).
- Exécution de ces instructions pour obtenir une réponse à des requêtes (sect. 3.6).
- Combinaison des étapes précédentes pour obtenir un exécutable (sect. 3.7).
- Développement et exécution de tests (sect. 3.8).

1.2 Travail demandé

Dates

Le projet doit être effectué dans des groupes d'au plus trois étudiants, mais vous pouvez aussi travailler seul. Merci de constituer les groupes et de m'en communiquer la composition jusqu'au **vendredi 7 mars 2025 à 12h**.
Le projet est à rendre jusqu'au **lundi 7 avril 2025 à 23h** au plus tard, pour une soutenance qui a lieu le **vendredi 11 avril 2025** aux horaires du TP.

Format des fichiers à rendre : Vous avez deux possibilités :

1. Vous pouvez déposer une archive (`tarfile` ou `zip`, pas de format `rar` ou autres formats propriétaires) sur Moodle, *avant la date limite*, avec le contenu indiqué en bas.

1. https://www.gqlstandards.org/

2. Alternativement, pour vous inciter à travailler avec des systèmes de gestion de version, vous pouvez aussi nous communiquer les coordonnées d'un dépôt (seuls formats admis : `git` ou Mercurial) où nous pouvons récupérer votre code. L'adresse du dépôt doit être envoyée à martin.strecker@irit.fr avant la date limite, et le dépôt doit rester stable pendant au moins 24h après la date limite, pour nous permettre de récupérer le contenu. Évidemment, le dépôt doit être accessible en mode lecture. Si vous avez des doutes, envoyez l'adresse du dépôt au moins 24h avant la date limite pour permettre des tests. Nous nous engageons à évaluer le code uniquement après la date limite.

Le travail doit être un travail authentique qui ne doit pas être la copie (partielle ou intégrale) du code d'un collègue. Bien sûr, on ne vous interdit pas un travail collaboratif, et si vous avez travaillé avec des collègues et que vous supposez une trop grande similarité du code, indiquez vos collaborateurs dans le `README` à joindre au projet. L'utilisation d'assistants basés sur l'IA (ChatGPT, GitHub Copilot) est permis mais doit également être signalé dans le `README`.

Fichiers à rendre : Déposez une archive (`tarfile` ou `zip`, pas de format `rar` ou autres formats propriétaires) sur Moodle, *avant la date limite* ; ou alternativement un dépôt Git [2] ou Mercurial [3]. L'ensemble des fichiers doit comporter :

- un répertoire d'un projet complet qui doit compiler avec `dune build`, voir sect. A.4
- un sous-répertoire `bin` qui contient notamment un `main.ml`
- un sous-répertoire `lib` qui contient les autres fichiers `*.ml`
- un sous-répertoire `test` avec des fichiers tests, voir sect. B.
- un fichier `README` avec des instructions comment compiler et exécuter vos tests. Ce fichier doit en plus contenir un résumé du travail effectué, éventuellement des difficultés rencontrées, ou vos observations ou remarques (ce que vous n'avez pas pu faire, ce qui est particulièrement bien réussi, etc.).

1.3 Fichiers fournis

L'archive de *fichiers Caml* fournie contient les fichiers suivants :

- Le noyau du projet (la plupart dans `lib`, et `main.ml` dans `bin`) :
  - `display.ml` : affichage de graphes
  - `graphstruct.ml` : le type des graphes
  - `instr.ml` : instructions du langage intermédiaire
  - `interf.ml` : interface avec le parser
  - `lang.ml` les types pour manipuler des programmes source
  - `main.ml` programme principal
  - `sem.ml` sémantique / évaluation de programmes.
  - `typing.ml` typage de programmes.
- Analyse lexicale et syntaxique, `lexer.mll` et `parser.mly` : Des fichiers pour le développement d'un lexer/parser avec ocamllex et menhir. Vous devez les compléter et les fournir avec le reste de votre projet.
- Des fichiers `dune`, voir sect. A.4. Il n'est pas nécessaire de les modifier, mais vous avez le droit de le faire (dans ce cas, merci d'indiquer vos modifications dans le `README`).

2 Premier aperçu du langage

Comme vous savez, dans sa forme la plus épurée, un graphe est composé de *noeuds* et d'*arcs*. Les graphes de MiniGQL peuvent en plus avoir des *attributs* associés aux noeuds (mais pas aux arcs, contrairement à GQL). MiniGQL permet

2. https://git-scm.com/
3. https://www.mercurial-scm.org/

— de construire des graphes, en créant des noeuds et des arcs,
— de les modifier, en supprimant des noeuds ou arcs ou en changeant des attributs
— d'interroger les graphes en cherchant des noeuds qui ont certaines propriétés (valeurs d'attributs ou relations avec d'autres noeuds).

Toutes ces manipulations seront appelées des *requêtes*.

Les graphes (et aussi le langage MiniGQL) sont typés. Dans notre exemple, nous avons des types de noeuds de Personnes et Entreprises (écrit `P` et `E`). Une personne a un nom (de type `string`) et un âge (de type `int`), une entreprise peut être une PME (attribut `pme` de type `bool`). Une personne peut être ami d'une autre personne (relation `ami`) et employée par une entreprise (relation `emp`). Une entreprise peut être fournisseur d'une autre entreprise (relation `f`). Le typage assure qu'une personne n'a pas l'attribut `pme` ou qu'une entreprise ne peut pas être employée par une personne.

Le typage doit être déclaré avant la construction de tout graphe. Les types des noeuds sont déclarés avec le nom du type et la liste des attributs (entre accolades). Les types des relations sont déclarées avec une notation visuelle en forme d'arc avec le type source, le nom de la relation et le type cible. La déclaration des types de l'exemple est :

```
(:P {nom string, age int})
(:E {nom string, pme bool})

(:P) -[:ami]-> (:P)
(:P) -[:emp]-> (:E)
(:E) -[:f] -> (:E)
```

Figure 1 – Types d'une base de données

Il est à noter que les types de noeuds dans les déclarations de relation sont des références vers des types précédemment déclarés. Vous aurez remarqué que ces déclarations définissent en fait un *graphe de types* dont les noeuds sont les types de noeuds et les arcs les types de relations.

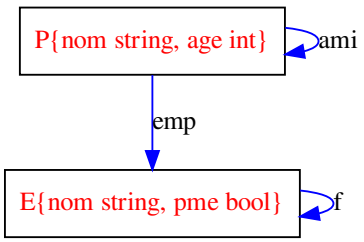

Figure 2 – Graphe de types

Nous pouvons maintenant construire le graphe et le manipuler. Voici les **constructeurs de requêtes** :
— `create pat` : création de noeuds et arcs conformes au pattern `pat`. La notion de pattern (motif) sera expliquée plus tard (sect. 3.2). Intuitivement, un motif décrit un sous-graphe qui a une certaine forme.
— `delete pat` : suppression de noeuds et arcs selon `pat`.
— `set v.a = e` : mise à jour de l'attribut `a` pour la variable `v` par la valeur de l'expression `e`.
— `match pats` : liaison de noeuds à des variables dans les `pats` (une liste de `pat`). La notion de liaison sera expliquée plus bas.

— `where e` : restriction à des liaisons qui satisfont l'expression (booléenne) `e`.
— `return vs` : renvoie les variables `vs`.

Une *requête* est une séquence de ces constructeurs, qui peuvent en principe être enchaînés dans n'importe quel ordre. Il faut pourtant noter que quelques séquences sont insensées et / ou produisent des erreurs, par exemple un `where` avant un `match` (pas de variables liées) ou un `match` avant un `create` (graphe inexistant). Un *programme* est composée d'une déclaration de types et d'une requête.

Voici des instructions pour créer un graphe :

```
create
   (marie: P) -[:emp]-> (ab: E),
   (pierre: P) -[:emp]-> (pp: E)
set
   marie.nom = "Marie␣Dubois",    marie.age = 25,
   pierre.nom = "Pierre␣Dupont", pierre.age = 24,
   ab.nom = "Airbus", ab.pme = false,
   pp.nom = "Petit␣Pain", pp.pme = true
create
   (pp) -[:f]-> (ab),
   (marie) -[:emp]-> (ab),
   (pierre) -[:emp]-> (pp),
   (marie) -[:ami]-> (pierre)
```

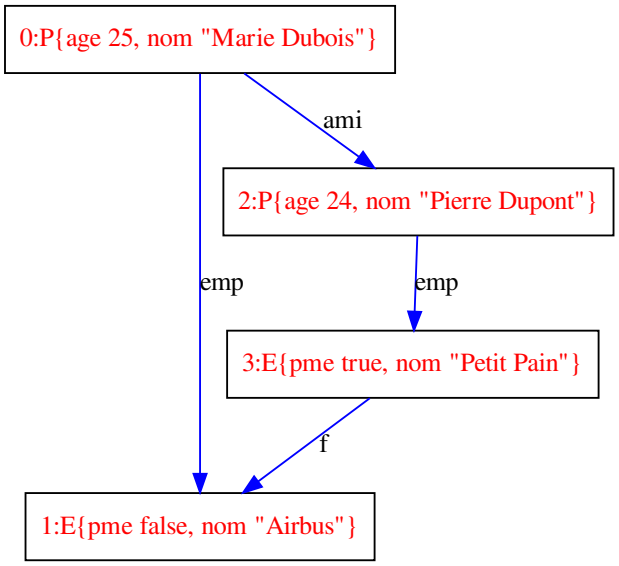

FIGURE 3 – Instance initiale du graphe

Avec `(marie: P)`, on déclare une *variable* nommée `marie` avec le type `P`. Dans ce cas, on crée aussi un seul noeud du type `P`. On lui associe un identifiant unique (un nombre, dans ce cas : 0). Pareil pour la déclaration `(ab: E)` avec la création du noeud 1. La première ligne a donc pour effet de créer les deux noeuds (sans attributs) et un arc `emp` entre eux.

La fig. 3 montre le graphe construit par la séquence des `create` et `set`. Comme nous verrons dans la suite, une variable peut correspondre à plusieurs noeuds, mais pour l'instant, le rapport est univoque :

pp	pierre	ab	marie
3	2	1	0

Une telle correspondance entre les variables d'un programme et les noeuds du graphe est appelée *liaison*. De nouvelles liaisons sont crées par les opérations `create` ou `match` ; elles sont modifiées par un `where`, `return` et aussi `match` ; elles sont supprimées par un `delete`.

Par exemple, la commande

```
match (p: P) -[:emp]-> (e: E)
```

introduit deux nouvelles variables `p` et `e` et les lie aux deux instances de la relation `emp`. La table des liaisons est maintenant :

e	p	pp	pierre	ab	marie
3	2	3	2	1	0
1	0	3	2	1	0

Un `where` ne garde que les lignes de la table dont les noeuds valident l'expression booléenne :

```
match (p: P) -[:emp]-> (e: E) where p.age >= 25
```

supprime la première ligne (parce que Pierre Dupont n'a pas 25 ans).

e	p	pp	pierre	ab	marie
1	0	3	2	1	0

Enfin, si on ne s'intéresse qu'aux variables `p` et `e`, on peut les projeter avec un `return` :

```
match (p: P) -[:emp]-> (e: E)
return p, e
```

p	e
2	3
0	1

Notre langage permet d'écrire des motifs plus complexes de manière naturelle. Par exemple, la requête "cherche les personnes qui travaillent pour une entreprise et aussi le fournisseur de cette entreprise" s'écrit :

```
match (p: P) -[:emp]-> (e: E),
      (p) -[:emp]-> (ef: E) -[:f] -> (e)
```

Ici, on utilise deux motifs. Le premier crée des liaisons pour `p` et `e` comme avant. Le deuxième est une contrainte supplémentaire qui utilise un chemin avec deux arcs. Elle exige l'existence d'un noeud `ef` intermédiaire. On voit que ce motif n'existe pas dans le graphe, on obtient donc une table vide comme résultat de la requête.

3 Description du travail demandé

3.1 Graphes

La structure fondamentale est celle de graphe (voir fichier `graphstruct.ml`), assez classique : Un graphe est un ensemble de noeuds et d'arcs. Les noeuds `'n` sont identifiés par un identifiant unique et peuvent comporter des informations `'i` supplémentaires. Les arcs entre deux noeuds sont marqués par une relation `'r`. Avec cela, on obtient le type de graphe :

```
type ('n, 'i) db_node = DBN of 'n * 'i
type ('n, 'r) db_rel = DBR of 'n * 'r * 'n
type ('n, 'i, 'r) db_graph =
     DBG of (('n, 'i) db_node list) * (('n, 'r) db_rel list)
```

A faire

Il peut être utile de rajouter des définitions de fonctions auxiliaires dans ce fichier (chercher un noeud avec un identifiant etc.). Ceci peut être fait graduellement.

3.2 Le langage source

Le langage source (voir `lang.ml`) est celui qui permet d'écrire des déclarations de type et des requêtes dans la forme conviviale vue en sect. 2. Il y a aussi un langage d'instructions (sect. 3.4), plus élémentaire, sur lequel nous définirons le typage et la sémantique.

Nous recommandons de commencer avec un langage source relativement simple et de rajouter du sucre syntaxique et des aspects plus complexes au fur et à mesure. Ainsi, vous pouvez d'abord travailler avec des noeuds sans attributs et les rajouter plus tard.

D'abord un peu de terminologie, conforme avec [FGG+18]. Il y a plusieurs genres de noms qui apparaissent dans les programmes : des noms de variables (comme `marie, ab, pp`) ; des noms de champs d'attributs (comme `nom, age, pme`) et les types de noeuds (comme `P, E`) et de relations (`ami, emp, cl`). Ces derniers sont appelés *label* dans la littérature. In fine, tous ces noms sont des chaînes de caractères, mais il est utile d'introduire des types distincts :

```
type vname = string   (* names for variables ... *)
type fieldname = string   (* names for attribute fields *)
type label = string  (* labels serving as names for node or relation types *)
```

Les types apparaissant dans les attributs seront appelés `attrib_tp`. Nous nous limitons aux types d'attribut `bool, int, string` et introduisons :

```
type attrib_tp = BoolT | IntT | StringT
type attrib_decl = fieldname * attrib_tp
```

Avec ceci, nous pouvons définir les types d'une base de données (déclarations de types de noeuds et de relations, comme dans fig. 1), qui est la première partie d'un programme :

```
type db_tp = (label, attrib_decl list, label) Graphstruct.db_graph
type prog = Prog of db_tp * query
```

Le reste de cette section décrit le type `query`, qui est une liste de clauses, où une clause est un des six constructeurs de requêtes ; leur définition suit presque immédiatement de la syntaxe mentionnée en sect. 2.

```
type clause
  = Create of pattern list
  | Match of pattern list
  | Delete of delete_pattern
  | Return of vname list
  | Where of expr
  | Set of (vname * fieldname * expr) list
type query = Query of clause list
```

La structure des expressions `expr` est simple : c'est où bien une constante (de l'un des trois types de base : booléen, entier, chaîne de caractères), un accès à un champ (comme `marie.age`) ou une opération binaire. Nous n'entrons pas dans les détails ici.

Un `pattern` est une séquence arbitrairement longue de chemins dans un graphe de la forme `(n0) -[:r0]-> (n1) ... (nk) -[:rk]-> (n(k+1))` ou un `ni` peut ou bien déclarer une nouvelle variable, de la forme `(marie: P)`, ou référencer une variable existante, de la forme `(marie)`.

```
type node_pattern
  = DeclPattern of vname * label     (* full node pattern with type declaration *)
  | VarRefPattern of vname           (* simple reference to var representing a node *)
type pattern
  = SimpPattern of node_pattern
  | CompPattern of node_pattern * label * pattern
```

Un `delete_pattern` est similaire sauf qu'on ne peut pas y déclarer de nouvelles variables mais uniquement supprimer des (noeuds liés aux) variables existantes.

```
type delete_pattern
  = DeleteNodes of vname list
  | DeleteRels of (vname * label * vname) list
```

Avec ceci, nous concluons le survol de `lang.ml`.

A faire

Essayez de bien comprendre les définitions des types. Il n'y a rien à programmer ici.

3.3 Analyse lexicale et syntaxique

Nous vous fournissons les fichiers `lexer.mll` (complet, rien à modifier) et `parser.mly` (fragmentaire), que vous devez compléter pour parser des programmes. Puisque la syntaxe concrète reflète assez directement la syntaxe abstraite, le parser est facile à écrire (120 lignes de code).

A faire

Complétez la définition de la grammaire.

3.4 Instructions

Pour avoir un langage plus facile à manipuler que le langage de sect. 3.2, nous introduisons un type d'instructions :

```
type action = CreateAct | MatchAct
type instruction
  = IActOnNode of action * vname * label
  | IActOnRel of action * vname * label * vname
  | IDeleteNode of vname
  | IDeleteRel of vname * label * vname
  | IReturn of vname list
  | IWhere of expr
  | ISet of vname * fieldname * expr
```

La différence essentielle est que les motifs (`pattern`) plus complexes du langage source ont été remplacés par des actions qui n'opèrent que sur un noeud ou une relation à la fois, ce qui facilite le typage et la sémantique.

Regardons la requête

```
create
   (marie: P) -[:emp]-> (ab: E), (pierre: P) -[:emp]-> (pp: E) -[:f]-> (ab)
return marie, pierre
```

du langage source qui est une manière compacte d'effectuer plusieurs opérations, à savoir la création de 4 noeuds et de 3 arcs entre eux. La séquence d'instructions correspondante est :

```
IActOnNode (CreateAct, "marie", "P"); IActOnNode (CreateAct, "ab", "E");
IActOnRel (CreateAct, "marie", "emp", "ab");
IActOnNode (CreateAct, "pierre", "P"); IActOnNode (CreateAct, "pp", "E");
IActOnRel (CreateAct, "pierre", "emp", "pp");
IActOnRel (CreateAct, "pp", "f", "ab");
IReturn ["marie"; "pierre"]
```

Surtout, dans le motif `(marie: P) -[:emp]-> (ab: E)`, il y a la déclaration de deux nouveaux noeuds qui doivent être créés avant l'insertion de la relation `emp` entre eux. Par contre, dans le motif `(pp: E) -[:f]-> (ab)`, il suffit de référencer le noeud `ab` parce qu'il a été créé précédemment.

Nous disons qu'un programme est "normalisé" s'il contient une liste d'instructions au lieu d'une requête au sens de sect. 3.2.

```
type norm_query = NormQuery of instruction list
type norm_prog  = NormProg of db_tp * norm_query
```

A faire

Écrivez un traducteur de `prog` vers `norm_prog`. Évidemment, l'essentiel de ce traducteur est une fonction `normalize_clause: clause -> instruction list` ; la `db_tp` reste inchangée.

3.5 Vérification des types

La vérification de types (à programmer dans le fichier `typing.ml` ; se référer aussi à sect. D.1 pour voir comment commencer) a pour but de vérifier le bon typage des programmes de MiniGQL. La vérification de types se fait normalement sur le langage source pour avoir de meilleurs messages d'erreur. Pour des raisons de simplicité, nous la ferons dans ce projet sur les instructions de sect. 3.4.

Un `norm_prog` est composé de deux parties, et ainsi :

1. nous vérifions d'abord la cohérence des déclarations de types pour construire un `db_tp` ;
2. sur cette base, on fait la vérification des instructions de `norm_query`.

Déclaration de types : Les seules erreurs lors de la déclaration de types sont des déclarations multiples du même type de noeud ou de relation, et la référence à un label non déclaré dans une relation, par exemple, dans fig. 1, une relation `(:P) -[:r]-> (:L)` pour un label non déclaré `L`. Si ces déclarations sont correctes, on peut construire le graphe de types, comme dans sect. 2.

A faire

Écrivez la fonction `check_graph_types` qui prend un graphe de types (`db_tp`), et qui renvoie un résultat de type `result`.

Vérification des instructions Sur la base du graphe de types, nous vérifions les instructions du programme, une après l'autre. Lors du traitement de ces instructions, intuitivement, de nouvelles variables apparaissent ou disparaissent. Comme dans d'autres langages de programmation, nous enregistrons l'information sur les variables actuellement actives dans un environnement :

```
type environment = { types:  db_tp; bindings: (vname * label) list }
```

Le graphe de types (composant `types`) reste inchangé lors de la vérification de types, tandis que les `bindings` évoluent. Le résultat de la vérification aura le type `tc_result` défini par :

```
type tc_result = (environment, string list) result
```

utilisant le type prédéfini `result` qui sera discuté en cours. Dans le cas d'une vérification réussie, le résultat sera donc un nouvel environnement. Dans le cas d'une erreur de vérification, nous produisons une chaîne de caractères qui détaille les raisons de l'erreur et la rajoutons à la `string list`.

A faire

Écrivez la fonction `tc_instr` qui prend une instruction, un environnement et produit un `tc_result`.

Voici quelques indications supplémentaires sur l'effet de la vérification de types pour chacune des instructions :

- `IActOnNode` (que l'action soit `CreateAct` ou `MatchAct`) : Après avoir vérifié que la variable n'est pas encore liée et que le label existe, on rajoute la variable et le label à l'environnement.
- `IActOnRel` : on fait des vérifications similaires, mais renvoie l'environnement inchangé.
- `IDeleteNode` : on vérifie que la variable est déclarée, et la supprime de l'environnement.
- `IReturn` : Comme suggéré dans l'exemple introductoire, un `return vs` ne garde que la liste des variables `vs`, les autres sont supprimées de l'environnement. Les variables `vs` devraient être distinctes.
- `IWhere` : On vérifie que l'expression est bien typée et de type `BoolT` (l'environnement ne change pas). Le typage d'une expression est classique : écrivez une fonction `tp_expr: environnement -> expr -> attrib_tp` ; le seul aspect nouveau est de vérifier qu'un accès à un attribut de la forme `p.age` est bien défini.
- `ISet` : Il faut vérifier que lors d'une affectation de la forme `marie.age = 25`, l'expression affectée est bien typée et correspond au type de l'attribut. L'environnement ne change pas.

A faire

Combinez la vérification du graphe de types et d'une liste d'instructions pour vérifier un `norm_prog`.

Vous pouvez expérimenter avec deux manières de gérer des erreurs : Lorsque vous constatez une erreur, vous pouvez arrêter la vérification immédiatement (dans ce cas, vous n'avez qu'un seul message d'erreur) ; ou vous pouvez continuer la vérification afin d'identifier toutes les erreurs du programme. Pour le faire, utilisez différentes méthodes de propagation d'erreurs du type `result`.

A faire (optionnel)

Proposez et implantez une méthode pour assurer un accès toujours défini à des attributs.

Voici quelques remarques pour illustrer ce point : Avant une affectation de la forme `marie.age = 25`, l'accès `marie.age` à l'attribut produit une erreur parce que la valeur n'est pas définie. Pour éviter cette erreur, on peut s'assurer statiquement (avant exécution de la requête) que toute lecture d'un attribut est précédée d'une écriture de cet attribut. Une manière de le faire est d'enrichir l'environnement d'un composant qui enregistre pour chaque variable quels attributs sont définis. Ce composant est mis à jour lors d'une affectation à un attribut. Implantez les détails.

3.6 Évaluateur

Ici, nous implantons la sémantique des requêtes (voir fichier `sem.ml`) en écrivant un évaluateur pour les instructions.

Les requêtes se comportent comme un programme impératif, au sens que le programme modifie un *état*. Comme vu en sect. 2, cet état a deux composants : un graphe et une table associant des noeuds du graphe aux variables actives à un moment donné. Une instruction peut modifier le graphe, la table ou les deux.

Nous pouvons donc définir un état comme suit :

```
type ('v, 'n) table = Table of 'v list * 'n list list

type db_graph_struct = (Graphstruct.nodeid, attrib_struct, label) db_graph

type state = State of db_graph_struct * (vname, nodeid) table * nodeid
```

Le troisième composant de `State` de type `nodeid` sera évoqué plus tard.

A faire

Écrivez la fonction `exec_instr : state -> instruction -> state`.

Pour le faire, il est indispensable d'écrire des fonctions auxiliaires qui implantent la sémantique pour chacune des opérations.

Dans la suite, nous expliquerons la sémantique de chaque instruction parce que la sémantique de GQL que nous adoptons ici est parfois un peu particulière. Tout d'abord, nous rappelons qu'une ligne d'une table correspond à une combinaison de valeurs qui satisfont une contrainte qui est le résultat des requêtes précédentes. Par exemple,

p	e
2	3
0	1

correspond aux résultats de la requête

```
match (p: P) -[:emp]-> (e: E)
return p, e
```

et indique qu'il existe deux arcs entre des noeuds étiquetés `P` et `E`, à savoir les couples (2, 3) et (0, 1).

— *Création d'un noeud* (instruction de la forme `IActOnNode (CreateAct, v, lb)`) : La sémantique est subtile : Pour chacune des lignes du tableau, on crée un nouveau noeud (autrement dit, on ne crée pas un seul noeud qui sera rajouté à chacune des lignes). Dans le contexte de la requête en haut, un `create (x: P)` aurait l'effet de créer deux nouveaux noeuds, de les rajouter au graphe (ce sont des noeuds isolés) et au tableau, qui devient :

x	p	e
4	2	3
5	0	1

Le constructeur `State` du type `state` a trois arguments ; le dernier enregistre le plus petit `nodeid` qui n'est pas encore utilisé dans le graphe (dans l'exemple : 6) et peut donc être utilisé lors de la création du prochain noeud comme `nodeid`. Enregistrer ce `nodeid` évite de parcourir le graphe pour le trouver.

— *Création d'un arc* (instruction de la forme `IActOnRel (CreateAct, sv, lb, tv)`) : Dans le graphe, on crée un arc labélisé `lb` pour toutes les liaisons des variables `sv` et `tv` enregistrées dans le tableau. Le tableau lui-même reste inchangé. Dans l'exemple en haut, `create (x) -[:emp]-> (e)` crée les arcs avec label `emp` entre les noeuds 4 et 3 ainsi que 5 et 1.

— *Match avec un noeud* (instruction de la forme `IActOnNode (MatchAct, v, lb)`) : Le graphe reste inchangé, mais on modifie le tableau comme suit : Conceptuellement, on crée un nouveau tableau qui a pour seule colonne la variable `v` contenant tous les noeuds étiquetés `lb`. On forme ensuite le produit cartésien ("join") entre ce tableau singleton et le tableau déjà existant. Autrement dit, chaque solution pour `v` est combinée avec chacune des lignes du tableau existant.
Par exemple, pour le tableau

p	e
2	3
0	1

où les seuls noeuds de type `P` sont 2 et 0, un `match(m:P)` lie `m` a 2 et 0 et combine chacune de ces solutions avec chaque ligne du tableau, ce qui donne :

m	p	e
2	2	3
2	0	1
0	2	3
0	0	1

— *Match avec une relation* (instruction de la forme `IActOnRel (MatchAct, sv, lb, tv)` : Le graphe n'est pas modifié, mais le tableau : on ne garde que les lignes du tableau qui satisfont la contrainte, à savoir l'existence d'un arc étiqueté `lb` entre les noeuds `sv` et `tv`. Continuons l'exemple avec le tableau précédent, et une instruction `match (m) -[:emp]-> (e)`. Supposons à titre d'exemple qu'il existe un arc `emp` entre les noeuds (2, 3) ; (0, 1) mais pas entre (2, 1) ; (0, 3). La table résultant de ce match est alors

m	p	e
2	2	3
0	0	1

— *Suppression d'un noeud* (instruction de la forme `IDeletenNode v`) : On supprime le noeud du graphe. Pour ne pas avoir d'arc sans ancrage (*"dangling edge"*), il faut aussi supprimer tous les arcs qui ont ce noeud pour origine ou cible. La modification apportée à la table consiste en la suppression de la colonne de la variable. Il est possible que ceci laisse des lignes identiques - on n'essayera pas de les fusionner. Supposons que nous supprimons la variable `e` avec un `delete(e)`. Ceci réduit le graphe aux noeuds 0 et 2 (sans arcs) et laissera le tableau

m	p
2	2
0	0

— *Suppression d'une relation* (instruction de la forme `IDeleteRel (sv, lb, tv)`) : Le graphe est modifié en supprimant tous les arcs concernés, la table reste inchangée.
— *Restriction à une liste de variables* (instruction de la forme `IReturn vs`) : On projette les colonnes des variables `vs` de la table, en respectant l'ordre des variables dans `vs`. Dans la table précédente, un `return ["m"; "p"]` laissera la table inchangée tandis que `return ["p"; "m"]` inverse les colonnes. Encore une fois, ceci peut laisser des lignes identiques dans la table. Contrairement à un `delete`, le graphe reste inchangé ; évidemment, quelques noeuds présents dans le graphe peuvent disparaître de la table.
— *Sélection* (instruction de la forme `IWhere e`) : on ne garde que les lignes de la table pour lesquelles l'expression `e` évalue à `true`. Le graphe reste inchangé.
— *Modification d'attributs* (instruction de la forme `ISet (vn, a, e)`) : la table reste inchangé, et on modifie le graphe en changeant l'attribut `a` de chacun des noeuds liés à la variable `vn` avec la valeur de l'expression `e`.

3.7 Création d'un exécutable

A faire

Combinez les fonctions des sections précédentes en un exécutable que vous pouvez lancer de la console (voir fichier `bin/main.ml`).

Les principales étapes sont : l'analyse lexicale et syntaxique (sect. 3.3), la normalisation du programme (traduction des motifs complexes en liste d'instructions, sect. 3.4), vérification des types (sect. 3.5), évaluation du programme (sect. 3.6). Comme dernière étape, vous pouvez utiliser les fonctions d'affichage d'un graphe et d'une table fournies avec le projet (voir `display.ml`). Ces fonctions utilisent les applications `dot` [4] respectivement LaTeX [5] qui doivent être disponibles sur votre ordinateur, et qui génèrent les fichiers `graph.pdf` et `table.pdf`.

Vous pouvez actuellement lancer l'exécutable de deux manières différentes :

— mode interactif avec : `dune exec Proj_GraphDB i`
Permet de saisir des expressions pour tester le parser, par exemple `create (a:A)` (*terminer saisie avec Ctrl-d, arrêter avec Ctrl-c*)

— batch mode, de la forme : `dune exec Proj_GraphDB f test/tiny.q`
Exécute le programme contenu dans le fichier donné comme argument.

3.8 Documentation et tests

A faire

Documentez votre code et rajoutez quelques tests.

Ceci veut dire :

— Fournissez un niveau de documentation raisonnable de votre code dans les fichiers Caml. Il n'est pas nécessaire de commenter en détail toutes les fonctions auxiliaires, mais les fonctions essentielles doivent être commentées.

— Rajoutez aussi quelques tests dans le répertoire `test`, voir sect. B.

— Fournissez en plus un `README` qui décrit les spécificités de votre projet, par exemple des parties que vous n'avez pas réussi à implanter, des extensions que vous proposez, des algorithmes particulièrement notables ou intéressants que avez conçus, ...

A Gestion de projets "complexes" sous Caml

Nous utilisons plusieurs composants de l'écosystème Caml pour gérer des projets plus "complexes", ce qui veut dire ici :

— nécessitant des ressources externes comme d'autres paquetages : nous utilisons `opam` pour les installer (voir sect. A.1). Dans ce cas, nous vous invitons à installer quelques paquetages spécifiques pour le projet (sect. A.2).

— un éditeur performant. Nous suggérons d'utiliser VSCode (sect. A.3), mais ceci n'est pas imposé et reste un choix individuel.

— composés de plusieurs fichiers, nécessitant un contrôle intelligent sur l'ordre de compilation. Pour cela, nous utilisons `dune` (sect. A.4)

4. http://www.graphviz.org/
5. https://www.latex-project.org/

A.1 Utilisation d'`opam`

Opam est un gestionnaire de paquetages de l'écosystème Caml. Opam peut être installé comme décrit sur le site [6] ou être installé par des gestionnaires de paquetages de votre système d'exploitation.

Pour interagir avec Opam, lancer une première fois `opam init`

A.2 Paquetages `opam` pour le projet

A.2.1 Menhir

Nous utilisons Menhir [7] qui est au point de remplacer l'ancien générateur de parsers, `ocamlyacc`. Les détails sont décrits sur les transparents du TP Théorie des Langages. Son installation se fait avec `opam install menhir`

A.2.2 Deriving

Les instances d'un type de données ne peuvent typiquement pas être affichées directement, par exemple :

```
# Printf.printf "%s\n" (IntV 42) ;;
Error: This expression has type value but an expression was expected of type string
```

Pour afficher les instances d'un type non-fonctionnel avec `printf`, on peut rajouter une directive `deriving` à la définition du type. Un pré-processeur [8] génère alors une fonction `show_(nom du type)`. Par exemple :

```
type value
    = BoolV of bool
    | IntV of int
    | StringV of string
  [@@deriving show]
```

La fonction dérivée a le nom `show_value`, on peut maintenant écrire

```
# Printf.printf "%s\n" (show_value (IntV 42)) ;;
```

L'installation du pré-processeur qui génère les fonctions `show` se fait avec

```
opam install ppx_deriving
```

A.2.3 Librairie de graphes

Nous utilisons une librairie de graphes pour l'affichage de graphes avec Graphviz/Dot. L'installation se fait par :

```
opam install ocamlgraph
```

6. https://opam.ocaml.org/
7. https://gallium.inria.fr/~fpottier/menhir/
8. https://github.com/ocaml-ppx/ppx_deriving

A.3 Ocaml sous VSCode

VSCode[9] est une IDE très performante que nous recommandons d'utiliser, mais il existe d'autres éditeurs excellents comme Emacs[10] avec l'extension Tuareg[11].

Avant de lancer VSCode, installez les paquetages nécessaires pour faire le lien entre Ocaml et VSCode avec :

— `opam install ocaml-lsp-server`
— `opam install ocamlformat`

Maintenant, lancez VSCode, typiquement avec `code . &` dans la racine de votre projet, et installez l'extension `OCaml Platform` sous VSCode. *Attention,* il y a plusieurs extensions pour Ocaml, la `Platform` semble la plus actuelle et adaptée.

Pour voir les fonctionnalités disponibles sous VSCode, ouvrez la Command Palette (`Ctrl-Shift-P` sous Linux) et tapez Ocaml. Vous pouvez ensuite lancer `Open REPL` et évaluer des expressions sélectionnées à la souris. Si vos fonctions dépendent d'autres modules, cette démarche risque d'échouer ; voir sect. C.1.

A.4 Dune

Dune est un utilitaire d'aide à la compilation (*build system*), voir la description complète[12].

Vous pouvez en principe créer un nouveau projet nommé `Proj_GraphDB` avec

```
dune init proj Proj_GraphDB
```

ce qui va créer un répertoire et les fichiers de configuration essentiels. *Ne le faites pas ici* parce que nous vous fournissons déjà l'archive avec une configuration de base.

Après avoir extrait l'archive ou créé le projet avec `dune`, faites le suivant :

```
cd Proj_GraphDB
dune build
dune exec Proj_GraphDB
```

Ici, `dune build` compile le projet entier, et `dune exec` l'exécute (et en fait, le compile avant si ce n'est pas encore fait).

Le cycle de développement typique itère donc les étapes suivantes : création et édition de fichiers ; compilation avec `build` ; exécution avec `exec`. Vous pouvez rajouter d'autres fichiers `*.ml` de préférence dans le répertoire `lib` ; il seront pris en compte automatiquement par Dune.

Nous recommandons fortement d'utiliser un système de gestion de versions comme Git ou Mercurial. Voir la section *Using Git* du manuel[13]. En bref, rajoutez tous les fichiers au système de gestion de versions sauf ceux dans le répertoire `_build`.

B Exécution de tests

Pour effectuer des tests, vous pouvez utiliser une approche intégrée dans `dune`. Les tests ont la forme de *tests de régression* :

— On écrit les tests avant le code (ils servent alors à une spécification partielle du code) ou stocke les résultats d'exécutions du code (après vérification que le résultat est acceptable).
— Lors de tests ultérieurs, on peut constater si le code se comporte comme souhaité, ou comme il se comportait précédemment. Une divergence peut indiquer une erreur introduite par une modification du programme (régression).

9. https://code.visualstudio.com/
10. https://www.gnu.org/software/emacs/
11. https://github.com/ocaml/tuareg
12. https://dune.readthedocs.io/en/latest/quick-start.html
13. http://ocamlverse.net/content/quickstart_ocaml_project_dune.html

Pour des détails, voir la documentation [14], voici un résumé :

1. L'approche s'appuie sur des *cram tests* développés dans le monde Python [15]. Pour l'utiliser, il faut d'abord installer (une fois) un paquetage Python : `pip install cram`
2. Veuillez stocker vos tests dans le répertoire `test`. Dans notre cas, un test est composé de deux fichiers (merci de respecter cette convention, le mécanisme peut facilement dérailler) :
   — un *query file* avec l'extension `*.q`, par exemple `entreprise.q` qui contient une requête du langage source ;
   — un fichier test avec l'extension `*.t`, par exemple `entreprise.t`, qui contient une ligne appelant votre programme avec le fichier de requête correspondant, par exemple :
   ```
   $ dune exec Proj_GraphDB f entreprise.q
   ```
   Attention, la ligne doit commencer avec deux espaces blancs et un dollar.
3. Appelez `dune runtest` une première fois. Tous les tests (fichiers `*.t`) dans `test` sont exécutés. Vous voyez un message d'erreur (sans surprise, vous n'avez pas encore spécifié le résultat attendu) avec le résultat affiché par votre programme.
4. Si vous validez ces résultats, appelez `dune promote` (sinon, corrigez votre programme d'abord). Ceci stocke les résultats dans les fichiers `*.t`. Vous avez le droit de modifier ces fichiers à la main.
5. Tout appel ultérieur à `dune runtest` compare les résultats actuels avec les résultats stockés dans les fichiers `*.t`. En cas de conformité, rien n'est affiché ; sinon, vous voyez les différences et pouvez décider s'il s'agit d'une erreur de régression.

C Caml interactif

C.1 Utop

Contrairement à l'écriture de quelques fonctions isolées, le développement d'un projet plus conséquent s'appuie sur des librairies externes (par exemple `ocamllex, menhir`) ou requiert des modules compilés ou traités par un pré-processeur (par exemple *deriving*, sect. A.2.2). Pour bien gérer toutes les dépendances, nous recommandons d'utiliser `utop`, une version évoluée du desktop `ocaml`.

Pour l'installer, faire

```
opam install utop
```

Utop est lancé avec `dune utop`. Après modification de fichiers, il ne semble pas y avoir un autre moyen de les recharger que de quitter Utop, de faire un `dune build` et redémarrer `dune utop`.

Utop est assez convivial à utiliser, avec auto-complétion (tabulation), rappel de la ligne précédente (`Alt-p`), navigation dans la ligne dans le style de *bash* (`Ctrl-a, Ctrl-e, Ctrl-b, Ctrl-f`). On quitte Utop avec `Ctrl-d`.

C.2 Espace de noms

Supposons que votre projet s'appelle `Proj_GraphDB`. La compilation avec Dune place tous les composants dans cet espace de noms. En plus, chaque fichier, par exemple `lang.ml`, crée un espace de noms (nom du fichier avec majuscule initiale : `Lang`). Toute fonction ou autre composant est donc précédé du nom du projet et du module :

```
utop # Proj_GraphDB.Lang.show_value ;;
- : Proj_GraphDB.Lang.value -> string = <fun>
```

14. https://dune.readthedocs.io/en/stable/tests.html
15. voir https://bitheap.org/cram/

Vous pouvez toujours identifier un composant par son nom complet, ce qui peut devenir illisible. Pour vous débarrasser des préfixes, vous pouvez ouvrir l'espace de noms. Ceci peut créer des conflits si des composants de différents modules portent le même nom.

```
utop # open Proj_GraphDB ;;
utop # Lang.show_value (Lang.IntV 42) ;;
- : string = "(Lang.IntV␣42)"
utop # show_value (IntV 42) ;;
Error: Unbound value show_value
utop # open Lang ;;
utop # show_value (IntV 42) ;;
- : string = "(Lang.IntV␣42)"
```

D Comment progresser

D.1 Vérification de types

Nous vous conseillons de commencer avec la vérification de types, comme décrit en sect. 3.5. Parce que nous n'avons pas encore de parser à ce moment, il faut saisir la syntaxe abstraite directement en Caml. Une fois le parser implanté, vous pourrez traiter des exemples plus complexes.

Voici l'instance du type `DBG` produit par la déclaration de la fig. 1 :

```
DBG ([(DBN ("P", [("nom", Lang.StringT); ("age", Lang.IntT)]));
      (DBN ("E", [("nom", Lang.StringT); ("pme", Lang.BoolT)]))],
     [(DBR ("P", "ami", "P"));
      (DBR ("P", "emp", "E"));
      (DBR ("E", "f", "E"))])
```

1. Prenez ce graphe de types et implantez la fonction `check_graph_types`.
2. Pour tester votre fonction, modifiez cette définition de types
3. Sur la base d'un `graph_type` stable, écrivez la fonction `tc_instr` qui prend une instruction et un environnement et renvoie un `tc_result`. Quelques exemples d'instances d'instruction sont donnés en sect. 3.4. L'environnement utilisé peut être `initial_environment gt`, où `gt` est le *graph type* plus haut (la fonction `initial_environment` est prédéfinie dans le code).
4. Écrivez maintenant une fonction qui vérifie une liste d'instructions, comme la liste d'instructions en sect. 3.4.
5. Écrivez une fonction `typecheck` qui combine essentiellement les fonctions `check_graph_types` et la vérification d'une liste d'instructions pour faire la vérification d'un `norm_prog`. Avec ceci, vous avez accompli les exigences de la sect. 3.5.

Références

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher : An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, page 1433–1445, 2018. https://www.research.ed.ac.uk/files/56321692/cypher_sigmod18_crc_1.pdf.

[GGL+19] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. Updating graph databases with cypher. *PVLDB*, 12(12) :2242–2253, 2019. http://www.vldb.org/pvldb/vol12/p2242-green.pdf.