Universite de Mons-Hainaut
Faculté des Sciences

Interpréteur d'algèbre relationnelle

Olivier Christiaen
UMH-LIG1

Directeur du projet : M. Jef Wijsen

Année académique 2003-2004
Première licence en informatique de gestion

Interpréteur d'algèbre relationnelle

Introduction

Ce projet consiste en la réalisation d'un interpréteur d'algèbre relationnelle. Il vise à mettre à disposition des étudiants en informatique un outil didactique simple qui permettrait la réalisation d'exercices sur machine plutôt que traditionnellement sur papier.

Un langage spécifique et concis a été proposé pour permettre l'écriture des requêtes et la gestion des données, il a été baptisé SQF pour « simple query format »

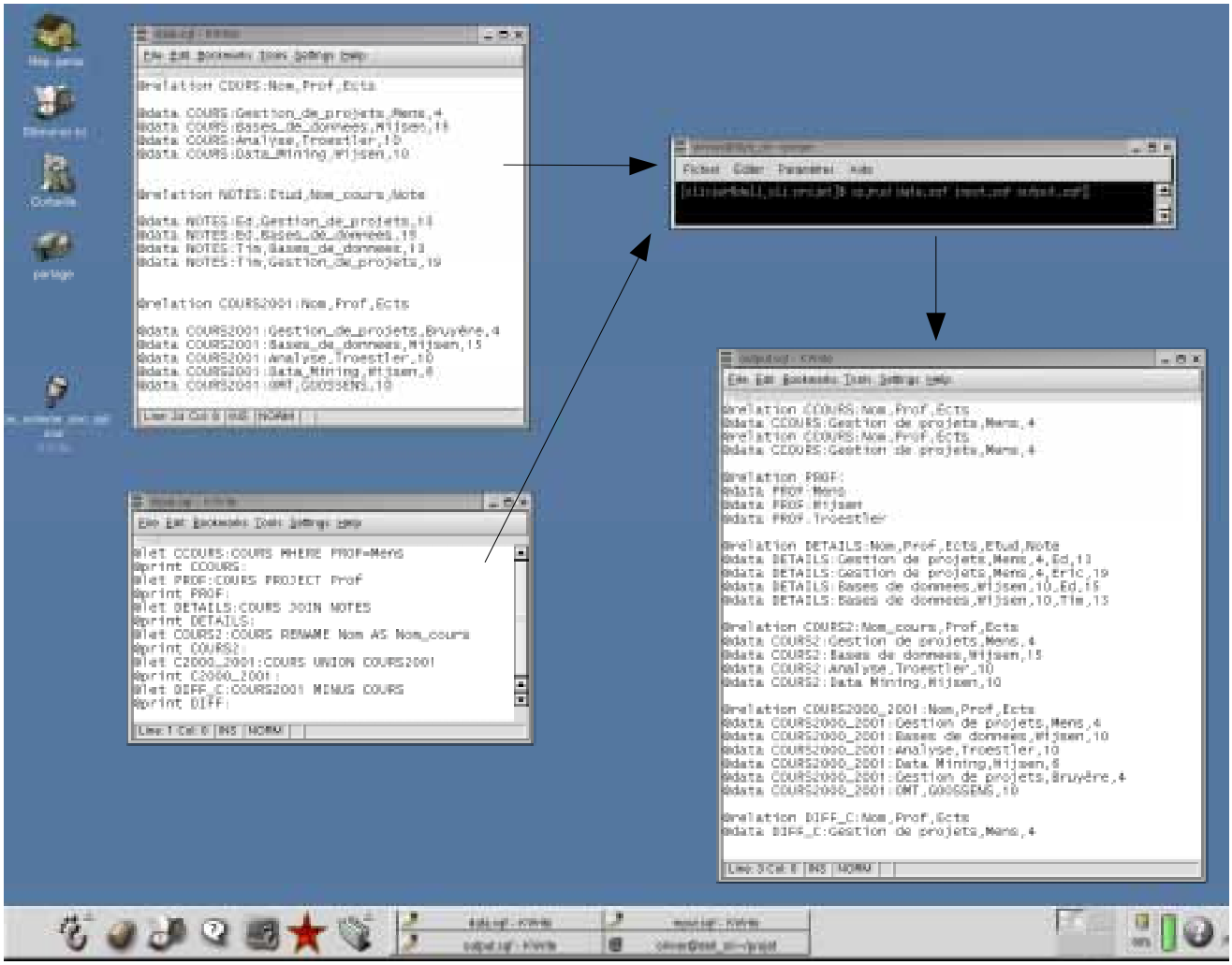
L'interface utilisateur est volontairement simplifiée de même que la syntaxe afin de permettre une prise en main rapide et de se focaliser sur l'algèbre à proprement parler.

Le logiciel doit être facilement installé et configuré et devrait idéalement être indépendant de la plate-forme utilisée ou de la disponibilité d'autres logiciels.

L'énoncé du projet est joint en annexe 1.

Fonctionnement attendu de l'application

L'utilisateur rédigera ses commandes SQF dans un fichier texte.
Il pourra créer plusieurs fichiers, un fichier constituant les tables et les données qui pourra lui servir à plusieurs reprises et un fichier contenant les requêtes SQF, ou inclure la totalité des commandes dans un seul fichier.
Ce ou ces fichiers seront ensuite passés en paramètres à la commande spjrud. L'utilisateur précisera également le nom du fichier qui contiendra les résultats. Ces opérations se dérouleront en mode console (sous Linux ou MS Windows), c'est à dire sans interface graphique, et en mode batch.
Le fichier de résultat répondra également à la syntaxe SQF de sorte qu'il puisse lui même servir de fichier d'entrée pour une autre exécution de l'application.
Les erreurs éventuelles seront affichées et comprendront le numéro de la ligne concernée.
La syntaxe des fichiers SQF sera détaillée dans un chapitre spécifique.



Exemple basé sur les données utilisées dans le chapitre relatif à l'algèbre relationnelle

L'algèbre relationnelle

Avant d'expliquer plus en détails le développement et le fonctionnement de l'application, je résume dans ce chapitre les concepts d'algèbre relationnelle qui seront utilisés dans le cadre de ce projet.

L'algèbre relationnelle[1] est une méthode d'extraction permettant la manipulation des tables (ou relations) et des colonnes.
Son principe repose sur la création de nouvelles tables (tables résultantes) à partir de tables existantes, ces nouvelles tables devenant des objets utilisables immédiatement.

L'algèbre comprend 6 opérations de base : sélection, projection, jointure, renommer, union, différence.

La sélection (ou restriction)

La sélection produit, à partir d'une relation, une relation résultante de même schéma mais ne comportant que les tuples qui répondent à la condition précisée en argument.

En sqf : @let CCOURS:COURS WHERE Prof=Mens

Relation initiale :

COURS	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	10

Résultat :

CCOURS	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4

La projection

La projection produit, à partir d'une relation, une relation résultante de schéma différent en ne conservant de la relation initiale que les attributs mentionnés en opérandes et les tuples correspondants en éliminant les doublons éventuels.

En sqf : @let PROF:COURS PROJECT Prof

Relation initiale :

COURS	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	10

Résultat

PROF	Prof
	Mens
	Wijsen
	Troestler

[1] Pour plus de détails, voir [02] et [05]

La jointure

La jointure produit, à partir de deux relations R1 et R2, une relation résultante de schéma différent dont les tuples sont obtenus en composant un tuple de R1 et de R2 lorsque ceux-ci ont la même valeur d'attribut pour des attributs de même nom.
S'il n'y a pas d'attribut en commun, l'effet de la jointure est un produit cartésien.

En sqf : @let DETAILS:COURS JOIN NOTES

Relations initiales :

COURS	***Nom_cours***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	6

NOTES	***Etud***	***Nom_cours***	***Note***
	Ed	Gestion de projets	13
	Ed	Bases de données	15
	Tim	Bases de données	13
	Eric	Gestion de projets	19

Résultat

DETAILS	***Nom_cours***	***Prof***	***Ects***	***Etud***	***Note***
	Gestion de projets	Mens	4	Ed	13
	Gestion de projets	Mens	4	Eric	19
	Bases de données	Wijsen	10	Ed	15
	Bases de données	Wijsen	10	Tim	13

Renommer

Renommer produit, à partir d'une relation, une relation résultante dans laquelle l'attribut désigné a été renommé par la nouvelle valeur transmise.

En sqf : @let CCOURS:COURS RENAME Nom AS Nom_cours

Relation initiale

COURS	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	10

Résultat

CCOURS	***Nom_cours***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	10

Union

L'union produit, à partir de deux relations de même schéma, une relation résultante de même schéma ayant pour tuples ceux appartenant au deux ou à une des relations en éliminant les doublons éventuels.

En sqf : @let C2000_2001:COURS_2000 UNION COURS_2001

Relations initiales

COURS_2000	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	6

COURS_2001	***Nom***	***Prof***	***Ects***
	Gestion de projets	Bruyère	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	6
	OMT	Goossens	10

Résultat :

C2000_2001	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	6
	Gestion de projets	Bruyère	4
	OMT	Goossens	10

Différence

La différence produit, à partir de deux relations de même schéma, une relation résultante de même schéma ayant pour tuples ceux appartenant à la première relation mais pas à la seconde.

En SQF : let DIFF_C :Cours_2001 MINUS Cours_2000

Relations initiales

COURS_2000	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	6

COURS_2001	***Nom***	***Prof***	***Ects***
	Gestion de projets	Bruyère	4
	Bases de données	Wijsen	10
	Analyse	Troestler	10
	Data Mining	Wijsen	6
	OMT	Goossens	10

Résultat :

DIFF_C	***Nom***	***Prof***	***Ects***
	Gestion de projets	Mens	4

Opérations dérivées

Des opérations dérivées (division et intersection) de même que des fonctions d'agrégats (AVG, COUNT, MAX, MIN, SUM) peuvent également être rencontrées mais ne seront pas développées dans le cadre de ce projet.

Considération pratique

L'algèbre relationnelle ne comprend pas d'opérations qui permettent de créer une relation, d'y introduire des données ou d'en afficher le contenu.
Ces opérations indispensables pour une projet automatisé seront définies dans SQF.

Algèbre relationnelle et Backus Naur Form (BNF[2]).

Syntaxe de l'algèbre relationnelle

La notation BNF permet de spécifier les règles de syntaxe d'un langage [04].
Les opérations de l'algèbre relationnelle, peuvent être définie avec le formalisme BNF de la façon suivante :

```
<relational expression> ::= <relvar name> | <relational operation> | (<relational expression>)
<relational operation>  ::= <select> | <project> | <join> | <rename> | <union> | <difference>
<select>                ::= <relational expression> WHERE <boolean expression>
<project>               ::= <relational expression> PROJECT <attribute name commalist>
<join>                  ::= <relational expression> JOIN <relational expression>
<rename>                ::= <relational expression> RENAME <renaming commalist>
<union>                 ::= <relational expression> UNION <relational expression>
<difference>            ::= <relational expression> MINUS <relational expression>
```

Syntaxe de SQF

Nous allons définir le langage SQF qui comprendra quatre instructions pour :

1. créer une relation : `@relation nom_relation:attribut1,attribut2,…attributZ`
2. insérer des données dans une relation : `@data nom_relation:data1,data2,…dataZ`
3. exécuter une opération de l'algèbre relationnelle : `@let nom_relation:<operation alg rel>`
4. afficher le contenu d'une relation `@print nom_relation:`

Ces instructions sont intégrées dans l'algèbre relationnelle définie ci-dessus. La syntaxe exhaustive de SQF peut alors être définie comme suit :

```
<sequence_sqf>          ::= <commande_sqf>+
<commande_sqf>          ::= <sqf_relation> | <sqf_data> | <sqf_let> | <sqf_print>
<sqf_relation>          ::= @relation <mot> : <tuple>
<sqf_data>              ::= @data <mot> : <tuple>
<sqf_print>             ::= @print <mot> :
<sqf_let>               ::= @let <mot> : <relational operation>
<relational expression> ::= <nom relation> | <relational operation> | (<relational expression>)
<relational operation>  ::= <select> | <project> | <join> | <rename> | <union> | <difference>
<select>                ::= <relational expression> WHERE <mot> <comparaison> <mot>
<project>               ::= <relational expression> PROJECT <tuple>
<join>                  ::= <relational expression> JOIN <relational expression>
<rename>                ::= <relational expression> RENAME <mot> AS <mot>
<union>                 ::= <relational expression> UNION <relational expression>
<difference>            ::= <relational expression> MINUS <relational expression>
<tuple>                 ::= <mot> | <liste_mots>
<mot>                   ::= <lettre>+(<lettre>|<chiffre>)* | <chiffre>+
<liste_mots>            ::= (<mot>,)+ <mot>
<comparaison>           ::= = | < | >
```

*Conventions : | signifie OU, + signifie AU MOINS UN, * signifie ZERO ou PLUSIEURS*

2http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html (Th. Estier, CUI - University of Geneva)

Exemple de fichier au format SQF

```
@relation COURS:Nom,Prof,Ects
@data COURS:Gestion_de_projets,Mens,4
@data COURS:Bases_de_donnees,Wijsen,15
@data COURS:Analyse,Troestler,10
@data COURS:Data_Mining,Wijsen,10

@relation NOTES:Etud,Nom_cours,Note
@data NOTES:Ed,Gestion_de_projets,13
@data NOTES:Ed,Bases_de_données,15
@data NOTES:Tim,Bases_de_données,13
@data NOTES:Tim,Gestion_de_projets,19

@relation COURS2001:Nom,Prof,Ects
@data COURS2001:Gestion_de_projets,Bruyère,4
@data COURS2001:Bases_de_donnees,Wijsen,15
@data COURS2001:Analyse,Troestler,10
@data COURS2001:Data_Mining,Wijsen,6
@data COURS2001:OMT,GOOSSENS,10

@let CCOURS:COURS WHERE PROF=Mens
@print CCOURS:

@let PROF:COURS PROJECT Prof
@print PROF:

@let DETAILS:COURS JOIN NOTES
@print DETAILS:

@let COURS2:COURS RENAME Nom AS Nom_cours
@print COURS2:

@let C2000_2001:COURS UNION COURS2001
@print C2000_2001:

@let DIFF_C:COURS2001 MINUS COURS
@print DIFF:
```

Méthodologie

L'énoncé du projet est précis et autorise une grande latitude quant à la mise en œuvre ce qui permet d'exploiter et d'évaluer plusieurs pistes.
J'identifie les différentes phases de la conception comme étant :

1. Réalisation d'un interpréteur en vue d'assurer l'analyse lexicale et syntaxique des fichiers en entrée
2. Définition des structures de données adéquates à la modélisation des relations
3. Ecriture du code assurant l'exécution des opérations de l'algèbre
4. Intégration des modules détaillés ci-dessus dans une seule application : l'interpréteur.

Concernant les phases d'analyse lexicale et syntaxique, je me suis documenté sur les outils génériques que sont Lex et Yacc. Issus du monde Unix il y plusieurs années, ceux-ci permettent de créer un compilateur en utilisant les expressions régulières et un formalisme proche de BNF.
Le choix de ces outils (j'ai retenu leurs équivalent sous licence GPL à savoir GNU Flex[3] et GNU Bison[4]) permet de bénéficier d'une part d'algorithmes éprouvés et performants mais surtout d'en réexploiter le concept pour d'autres développements.
Néanmoins, leur prise en main et l'étude de certains concepts des techniques de compilation a nécessité un investissement important.
J'explique brièvement le fonctionnement de Flex et Bison dans un autre chapitre.

Initialement, j'avais envisagé pour la gestion des données (et la modélisation des structures) un recours à un SGBD. La méthode consistait à traduire les requêtes SQF en SQL à l'aide du compilateur créé avec Bison.
Il s'agissait ensuite d'exécuter ces requêtes dans le SGBD et d'en traduire le résultat en SQF.

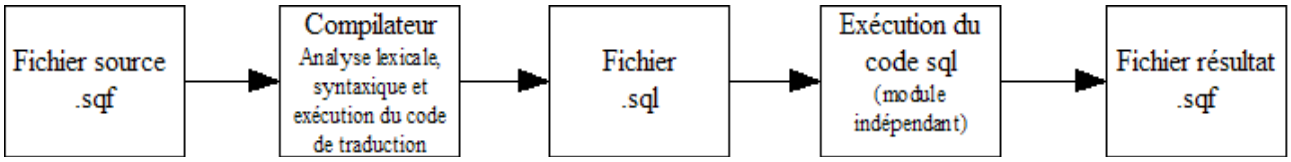


Fig : solution envisagée initialement

Le principal avantage était de pouvoir utiliser un système SGBD et de profiter ainsi des similitudes entre l'algèbre relationnel et SQL. Le code SQL était isolé et pouvait être destiné à un autre usage ou être optimisé.
Par contre, la traduction de l'algèbre relationnelle en SQL n'est pas toujours triviale et l'utilisateur doit disposer d'un SGBD sur sa machine pour exécuter l'application.
Il était également important d'être indépendant d'une base de données particulière et il fallait recourir à une solution portable (du type ODBC par exemple).
Il restait ensuite à accéder à ces bases de données depuis le programme principal.

J'ai choisi de développer (en C) l'exécution des commandes SQF ce qui assure une plus grande indépendance et réduit les modalités de configuration pour l'utilisateur.
L'aspect didactique de l'application prime et il est évident qu'aucune comparaison de performances ne peut être établie entre un SGBD et le système développé.

3http://www.gnu.org/software/flex/
4http://www.gnu.org/software/bison/

La méthode utilisée est représentée schématiquement comme suit :

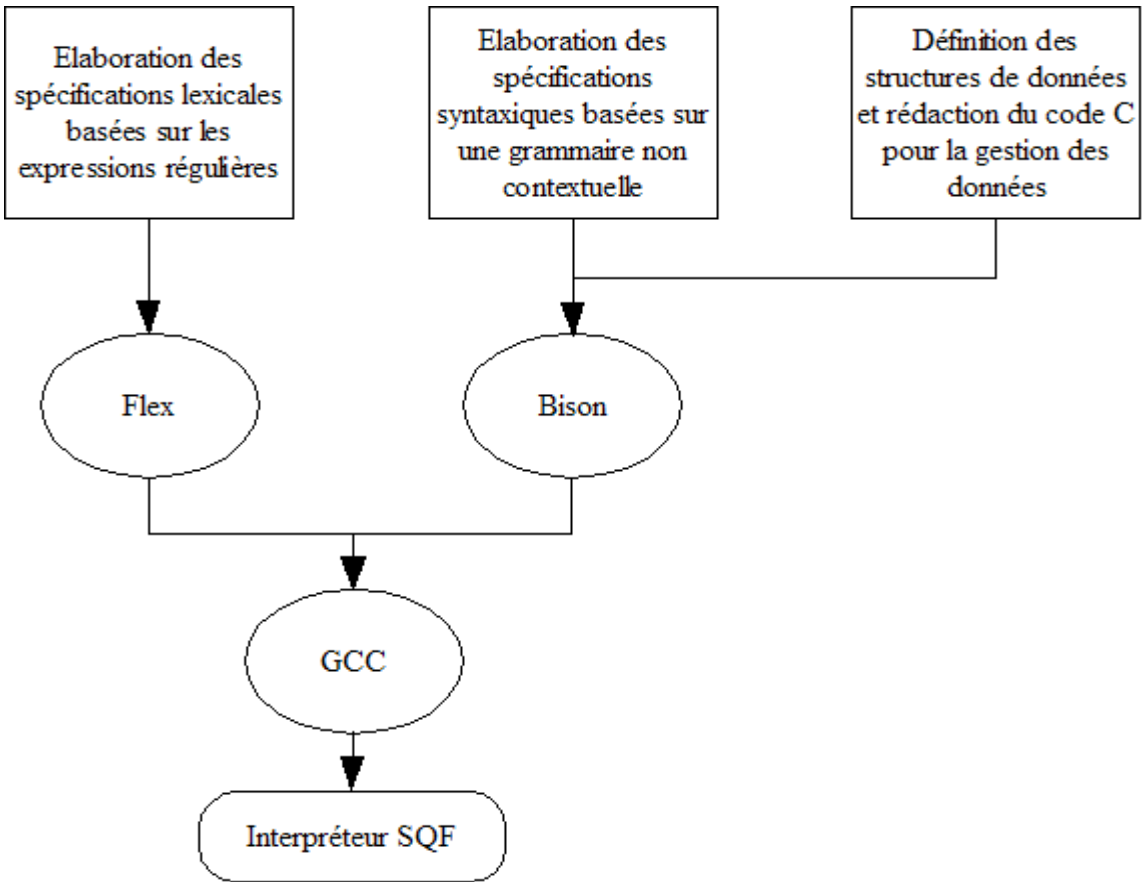


Fig : les phases du développement

Le code des trois fichiers de base est disponible en annexe.
Les outils Flex, Bison et GCC sont également disponibles pour MS Windows ce qui permet une compilation des codes sources pour les environnements Linux et/ou Windows.

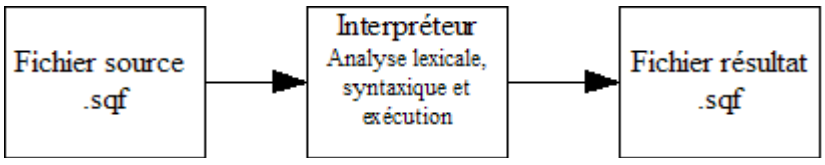


Fig : l'interpréteur accessible par la commande `spjrud`

Analyse lexicale et syntaxique

Comme décrit dans la rubrique méthodologie, la première phase du projet consistait à trouver un moyen pour détecter si le texte source en entrée était bien constitué d'unités lexicales propres à SQF (mots réservés, symboles) et ensuite de déterminer si la syntaxe des commandes correspondait à la grammaire de SQF (extraction des blocs et expressions, vérification des règles).

C'est en tentant de résoudre ce problème que j'ai découvert Flex et Bison. Ces logiciels sont mieux connus dans le monde Unix sous les noms de Lex et Yacc[5] [06].

Le but de l'analyse lexicale est de transformer une suite de symboles en terminaux (un terminal ou token peut être par exemple un nombre, un signe '+', un identificateur, etc...). Une fois cette transformation effectuée, la main est repassée à l'analyseur syntaxique. Le but de l'analyseur lexical est donc de 'consommer' des symboles et de les fournir à l'analyseur syntaxique.

L'analyse syntaxique est une analyse hiérarchique. L'objectif de cette analyse est de regrouper les terminaux fournis par l'analyse lexicale en phrases grammaticales.

Flex[6]

Flex est un utilitaire Unix[7] qui permet de générer automatiquement un analyseur lexical à partir d'une spécification.
Flex prend en entrée un ensemble d'expressions régulières[8] et produit en sortie le texte source d'un programme C qui, une fois compilé, est l'analyseur lexical correspondant au langage défini par les expressions régulières en question.

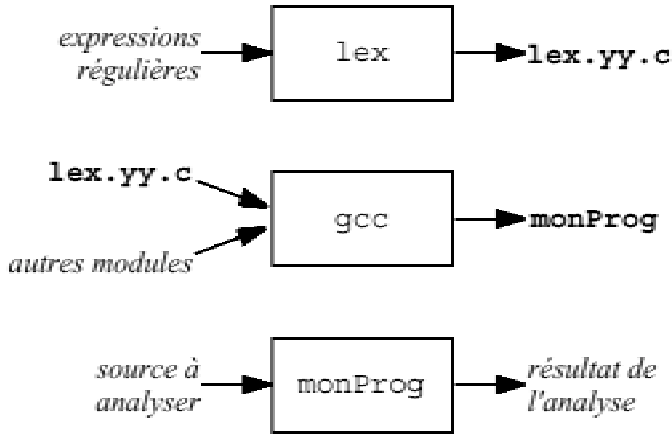


Fig . principe de fonctionnement de Lex (extrait de [01])

Le fichier de spécification pour Flex se compose de trois parties:

- la section de *définitions* ;
- la section de *règles* ;
- des procédures et fonctions fournies par le programmeur.

Une *définition* permet d'associer un identificateur à une expression régulière et de se référer par la suite dans la section de règles à cette expression régulière à travers son identificateur.

[5] The Lex & Yacc Page : http://dinosaur.compilertools.net/
[6] des informations plus complètes sur Flex sont disponibles dans le manuel de l'utilisateur : http://www.gnu.org/software/flex/manual/
[7] Flex et Bison pour Win32 : http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html
[8] Les expressions régulières de Flex : http://www.infeig.unige.ch/support/cpil/lect/lex/node2.htm

La section des *règles* contient des définitions d'expressions régulières avec les actions associées qu'il faudra exécuter dans le cas où une expression est reconnue.

La troisième section composée de procédures et de fonctions est recopiée telle quelle à la fin du programme résultat.

Le code source de l'application SQF constituant le fichier d'entrée pour Flex est joint en annexe.

Bison[9]

Bison est un outil qui permet de générer automatiquement un analyseur syntaxique.

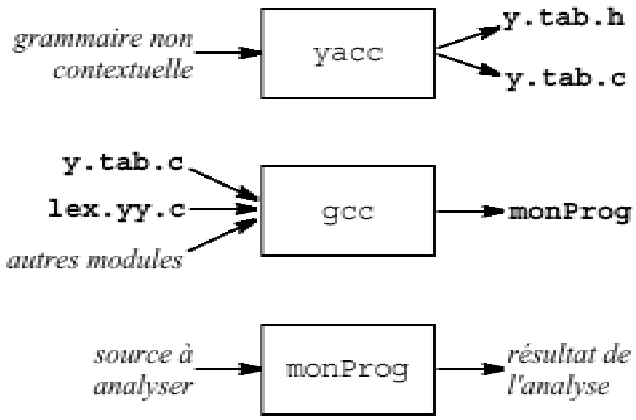


Fig . principe de fonctionnement de Yacc couplé ici à Lex (extrait de [01])

Le fichier de spécification pour Bison se compose de trois parties:

- la section de *déclarations* ;
- la section de *productions* ;
- des procédures et fonctions fournies par le programmeur.

La section *déclarations* contient : la déclaration des unités lexicales de la grammaire, la déclaration de variables temporaires, et des directives d'inclusions de fichiers d'en-tête.

La section des *productions* contient les règles de traduction. Chaque règle est formée d'une production de la grammaire et de son action associée (séquence d'instructions en C).
La définition de ces règles de traduction est proche de la notation BNF.

La troisième section composée de procédures et de fonctions qui aident à la traduction ou de récupérations d'erreurs.

Les informations à traiter par Bison doivent être fournies par un analyseur lexical. Dans ce projet, il s'agira de Flex.

Le code source de l'application SQF constituant le fichier d'entrée pour Bison est joint en annexe.

[9] des informations plus complètes sur Bison sont disponibles dans le manuel de l'utilisateur : http://www.gnu.org/software/bison/manual/

Mise en œuvre de Flex et Bison dans le cadre du projet

La spécification reprise dans la partie *définitions* du fichier Flex est la suivante :

```
Chiffre      [0-9]
Lettre       [A-Za-z_]
Espace       [ ]
Mot          ({Lettre}+({Lettre}|{Chiffre})*)|{Chiffre}+
Liste_mots   ({Mot}[,])+{Mot}
Comparaison  [=<>]
```

Elle décrit à l'aide d'expressions régulières les terminaux qui constituent la base de SQF.
Il faut encore y ajouter ceux qui constituent les commandes ou les opérateurs du langages. Ceux-ci sont ajoutés dans la partie *règles*, ce qui permet d'y associer une action. Dans notre cas, l'action consiste toujours à transmettre les tokens à l'analyseur syntaxique (Bison).

```
[ \t]               {} /* ignore les espaces et tabulations */
[\n]                {num_ligne++;} /* No de ligne pour msg erreur */
"WHERE"             {return OP_WHERE;}
"PROJECT"           {return OP_PROJECT;}
"JOIN"              {return OP_JOIN;}
"UNION"             {return OP_UNION;}
"MINUS"             {return OP_MINUS;}
"RENAME"            {return OP_RENAME;}
"AS"                {return OP_AS;}
"@relation"         {return SQF_RELATION;}
"@data"             {return SQF_DATA;}
"@let"              {return SQF_LET;}
"@print"            {return SQF_PRINT;}
{Mot}               { /* transmet a Bison le token et sa valeur */
                    yylval=(char *)malloc((1+yyleng)*sizeof(char));
                    strcpy(yylval, yytext);
                    return Mot;
                    }
{Liste_mots}        {
                    yylval=(char *)malloc((1+yyleng)*sizeof(char));
                    strcpy(yylval, yytext);
                    return Liste_mots;
                    }
{Comparaison}       {
                    yylval=(char *)malloc((1+yyleng)*sizeof(char));
                    strcpy(yylval, yytext);
                    return Comparaison;
                    }
.                   {return yytext[0];}
```

Ces règles sont parcourues dans l'ordre de leur affichage ce qui permet de distinguer un mot d'une commande par exemple.
Cette spécification est ensuite fournie à Flex qui, sur cette base, produit un code C à compiler qui sera l'analyseur lexical SQF.
Si le contenu du fichier d'entrée (data.sqf par exemple) qui est soumis à l'analyseur lexical ne correspond à aucune des règles lexicales définies : l'exécution est interrompue par un message d'erreur.

La partie *déclarations* du fichier de spécifications de Bison contient la liste des tokens que Bison est susceptible de recevoir depuis l'analyseur lexical.

```
%token Mot Liste_mots Comparaison
%token SQF_RELATION SQF_DATA SQF_LET SQF_PRINT
%token OP_WHERE OP_PROJECT OP_JOIN OP_UNION OP_MINUS OP_RENAME OP_AS
```

Il contient également des directives qui permettent de préciser le type des tokens et la priorité associée à ceux-ci.

La partie *productions* contient la définition de la grammaire sous un formalisme proche de BNF décrit précédemment. A chaque production, peut correspondre une action qui dans cette application, consiste en l'appel de fonctions de manipulation de données. Ces fonctions sont définies dans la partie *procédures* de Bison et reçoivent en paramètres les tokens obtenus.

```
Sequence_sqf          : commande_sqf
                      | sequence_sqf commande_sqf
;
commande_sqf          : sqf_relation
                      | sqf_data
                      | sqf_let
                      | sqf_print
;
sqf_relation          : SQF_RELATION Mot ':' tuple  {creer_relation($2,$4);}
;
sqf_data              : SQF_DATA Mot ':' tuple      {ajouter_tuple($2,$4);}
;
sqf_let               : SQF_LET Mot ':' relational_operation
;
sqf_print             : SQF_PRINT Mot ':' {afficher_relation($2);}
;
relational_expression : Mot
                      | relational_operation
                      | '('relational_expression')'
;
relational_operation  : r_select     {$$=$1;}
                      | r_project    {$$=$1;}
                      | r_join       {$$=$1;}
                      | r_rename     {$$=$1;}
                      | r_union      {$$=$1;}
                      | r_difference {$$=$1;}
;
r_select              : relational_expression OP_WHERE Mot Comparaison Mot
                            {selection();}
;
r_project             : relational_expression OP_PROJECT Mot {projection();}
;
r_join                : relational_expression OP_JOIN relational_expression
                            {jointure();}
;
r_rename              : relational_expression OP_RENAME Mot OP_AS Mot
                            {renommer();}
;
r_union               : relational_expression OP_UNION relational_expression
                            {union_rel();}
;
r_difference          : relational_expression OP_MINUS relational_expression
                            {difference();}
;
tuple                 : Mot
                      | Liste_mots
;
```

Bison reçoit chaque token depuis l'analyseur lexical et tente de l'associer à une règle en construisant un arbre syntaxique. En cas d'échec, une erreur de syntaxe empêchera le déroulement du programme.

Ce fichier de spécifications est traité par la commande Bison pour produire un analyseur syntaxique. L'analyseur lexical et l'analyseur syntaxique sont ensuite combinés par compilation pour obtenir l'interpréteur SQF.

Structures des données

La relation

Une *relation* est modélisée par une structure qui comprend :

- le nom de la relation
- le nombre de colonnes de la relation
- le nombre de tuples de la relation
- le nombre de tuples actifs de la relation (les tuples en double sont marqués comme étant inactifs et ne sont plus pris en considération dans les opérations)
- le nombre maximum de tuples de la relation (avant une réallocation mémoire éventuelle)
- un tableau de pointeur sur des tuples qui reprend l'adresse des tuples de la relation

Le tuple

Un *tuple* est modélisé par une structure qui comprend :

- son état : actif ou inactif
- la longueur du tuple (est une des clés de contrôle avant la vérification de doublon, si la longueur de deux tuples est différente, il est inutile de vérifier s'il s'agit de doublons)
- un tableau qui reprend la valeur de chaque champ sous la forme d'une chaîne de caractères

Le dictionnaire des relations

Un *dictionnaire des relations* a été implémenté, il contient :

- le nombre de relations existantes
- un tableau de pointeur sur les entrées du dictionnaire qui sont elles-mêmes constituées :
  - du nom de la relation
  - de l'adresse de la relation

Ce dictionnaire est consulté avant la création ou l'utilisation de toute relation, il est tenu à jour en permanence et est finalement utilisé en fin d'application pour permettre la libération de la mémoire utilisée.

Limitations :

Cette application ne gère pas de types différents pour les valeurs manipulées, toutes les valeurs sont considérées comme étant des chaînes de caractères (sans espace).
Un ensemble de directives (#define) fixe différentes limites à l'application, il s'agit :

- de la taille de ces chaînes (attribut ou valeur) qui est fixée à 20 caractères maximum
- du nombre d'attributs d'une relation qui est fixé à 8
- de la taille d'une chaîne de caractères représentant une liste de valeurs ou d'attributs qui est fixée à 180 caractères

Les valeurs définies semblent raisonnables dans un objectif didactique mais elles peuvent bien entendu être modifiées pour permettre une adaptation de l'application, ce qui nécessitera la recompilation du code.

Le nombre de tuples dans une relation n'est pas limité, la réallocation mémoire s'effectue par tranches de 30 tuples afin d'éviter les réallocations multiples à chaque insertion.

Analyse top-down des fonctions

Les fonctions dont il est question ci dessous constituent les fonctions principales de l'application. Elles visent directement à manipuler les données. Elles sont appelées depuis la partie production de la spécification Bison.

Convention : *dans l'analyse ci-dessous, une action ne sera exécutée que si l'action précédente a obtenu une réponse positive ou s'est exécutée correctement. Dans le cas contraire et même si ce n'est pas explicitement indiqué, la séquence est interrompue par une erreur. Ces fonctions principales font également appel à des fonctions de gestion moins conséquentes, qui sont reprises dans les schémas d'interactions entre fonctions. Elles sont abondamment commentées dans le code source.*

<Créer une nouvelle relation>

nécessite : le nom de la relation, l'intitulé de chaque champ de la relation

- vérifier si la relation n'existe pas dans le dictionnaire : relation_existe(...)
- vérifier si chaque intitulé de champ possède un nom unique dans la relation
- créer la table avec les intitulés de champs correspondants
- insérer les références de la relation dans le dictionnaire

<Insérer des données dans une relation>

nécessite : le nom de la relation, les données correspondant à chaque champs de la relation

- vérifier si la relation existe déjà dans le dictionnaire : relation_existe(...)
- vérifier que le nombre de champs en paramètre est égal au nombre de champs de la relation
- insérer le tuple dans la table (champs par champs)
- enregistrer la taille du tuple
- comparer la taille obtenue à celle des enregistrement déjà stockés dans la table
- si un enregistrement possède la même taille : les comparer (champ à champ : comparer_tuples(tuple1,tuple2))
- si un tuple identique existe marquer le nouveau tuple comme inactif
- augmenter le nombre d'enregistrements
- augmenter le nombre d'enregistrements actifs (si nécessaire)

<Vérifier le schéma de deux relations>

nécessite : le nom des deux relations

- vérifier si les deux relations existent dans le dictionnaire
- vérifier le nombre de champs de la relation1 et de la relation2
- vérifier la correspondance de chaque champ (champ à champ)

<SÉLECTION>

nécessite : le nom de la relation initiale, le nom de la relation résultante et le prédicat de sélection

- vérifier l'existence de la relation initiale dans le dictionnaire
- vérifier l'inexistence de la relation résultante dans le dictionnaire
- vérifier l'existence du champ repris dans le prédicat de sélection
- créer la nouvelle relation de même schéma que la relation initiale
- insérer les tuples actifs répondant au prédicat de sélection dans la nouvelle table (inutile de vérifier les doublons car même schéma)

<PROJECTION>

nécessite : le nom de la relation initiale, le nom de la relation à générer, et le(s) nom(s) des champs à conserver dans la nouvelle relation

- vérifier l'existence de la relation de base dans le dictionnaire
- vérifier l'inexistence de la relation résultante dans le dictionnaire
- vérifier l'existence des champs sélectionnés dans la relation existante
- créer la nouvelle relation
- insérer les valeurs correspondant aux champs retenus de la relation initiale dans la nouvelle relation (en utilisant <insérer des données dans une relation> de façon à éliminer les doublons).

<JOINTURE>

nécessite : les noms des deux relations initiales et le nom de la relation résultante

- vérifier l'inexistence de la relation résultante dans le dictionnaire
- vérifier l'existence des deux relations initiales dans le dictionnaire
- vérifier si des attributs sont communs aux deux relations initiales
- créer une relation temporaire qui est un produit cartésien entre les deux relations initiales
- conserver les tuples actifs pour lesquels les valeurs des attributs communs des deux relations sont identiques
- réaliser une projection pour éliminer les attributs redondants et les valeurs associées

<RENOMMER>

nécessite: le nom de la relation concernée, le nom de l'attribut à renommer et le nouveau nom de l'attribut

- vérifier l'existence de la relation dans le dictionnaire
- vérifier si le nouveau nom de l'attribut n'existe pas déjà dans la relation
- vérifier l'existence du champ à renommer dans la relation
- renommer le champ

<UNION>

nécessite: les noms des deux relations initiales et le nom de la relation résultante

- vérifier l'inexistence de la relation résultante dans le dictionnaire
- vérifier la correspondance du schéma des deux relations initiales
- créer la relation résultante avec le même schéma que les deux précédentes
- insérer dans la table résultante tous les tuples actifs de la première relation.
- insérer dans la table résultante tous les tuples actifs de la deuxième relation (en utilisant <insérer des données dans une relation> de façon à éliminer les doublons).

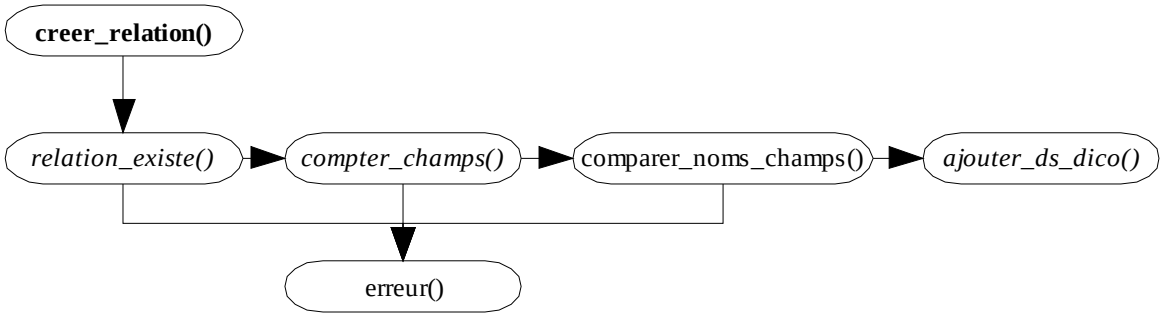
<DIFFÉRENCE>

nécessite: les noms des deux relations initiales et le nom de la relation résultante

- vérifier l'inexistence de la relation résultante dans le dictionnaire
- vérifier la correspondance du schéma des deux relations initiales (et par conséquent leur existence)
- créer la relation résultante avec le même schéma que les deux précédentes
- vérifier si chaque tuple de la première relation n'existe pas dans la deuxième relation
- s'il n'existe pas, l'insérer dans la relation résultante

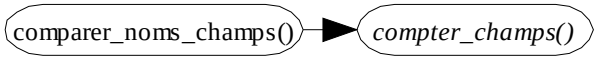
Schémas d'interactions des fonctions

Les fonctions dont le nom est mentionné en gras sont les fonctions principales appelées dans la section des productions du fichier Bison.
Les fonctions dont le nom est écrit en italique sont des fonctions secondaires qui ne recourent pas à d'autres procédures ou fonctions.



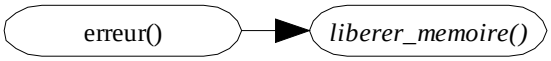
`relation* creer_relation(char * nom_relation, char * champs_rel);`

Crée une nouvelle relation et la référence dans le dictionnaire des relations.
Retourne l'adresse de la structure de la relation créée.



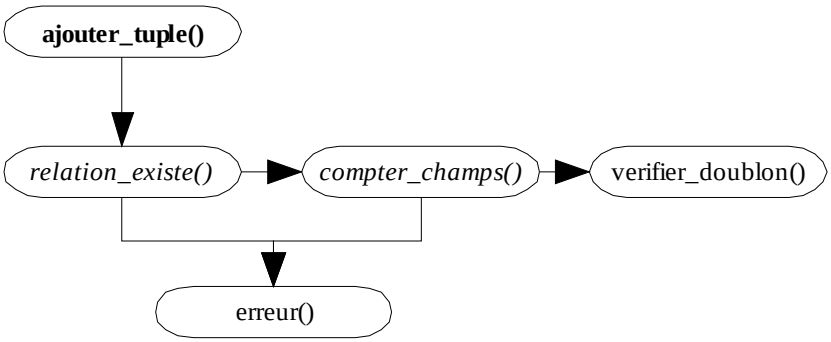
`int comparer_noms_champs(char * champs);`

Compare si des intitulés de champs d'une même relation sont tous différents.
Retourne 0 si les attributs sont tous différents sinon interruption.



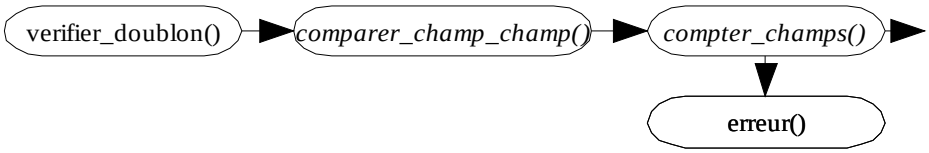
`void erreur(int code);`

Affiche le message d'erreur correspondant au code de l'erreur et précise le numéro de la ligne.
Elle libère la mémoire, ferme éventuellement les fichiers ouverts et interrompt le programme.



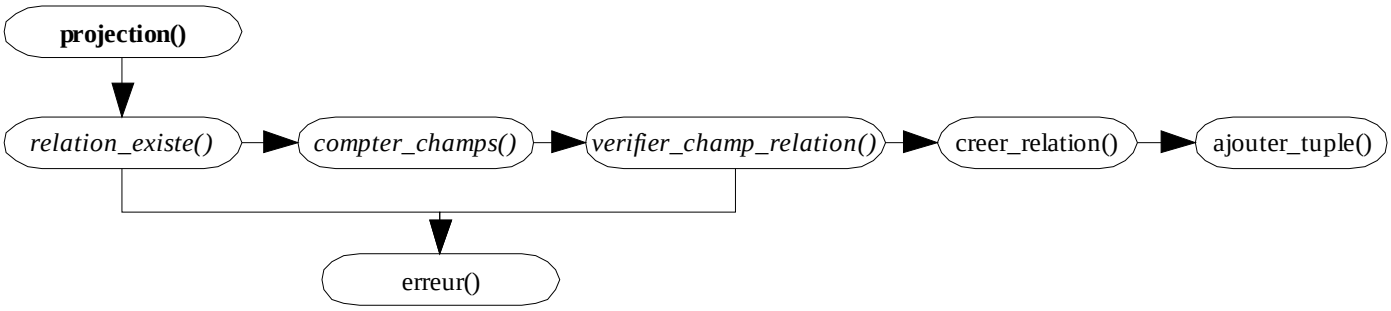
`int ajouter_tuple(char * nom_relation, char* champs);`

Ajoute un tuple dans la relation, retourne 1 en cas de réussite, interruption si erreur.



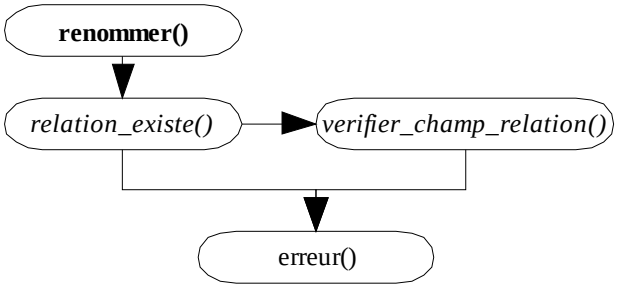
`int verifier_doublon(relation * relation_courante);`

Vérifie si le dernier tuple inséré est en double dans la relation, si oui un flag le rend inactif.



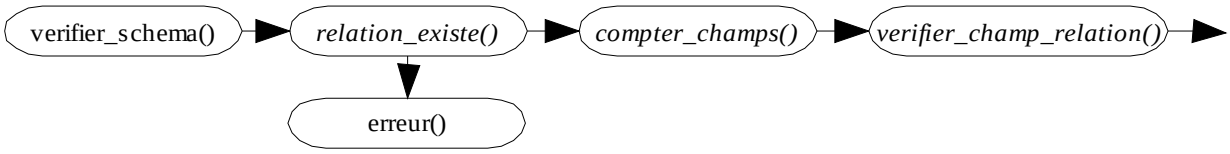
`void projection(char * nom_relation_init, char * nom_relation_res, char* champs_rel);`

Réalise la projection



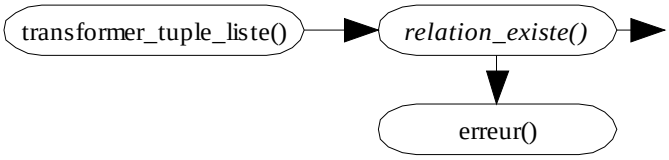
`void renommer(char * nom_relation, char * attribut_init, char * attribut_res);`

Renomme un attribut d'une relation



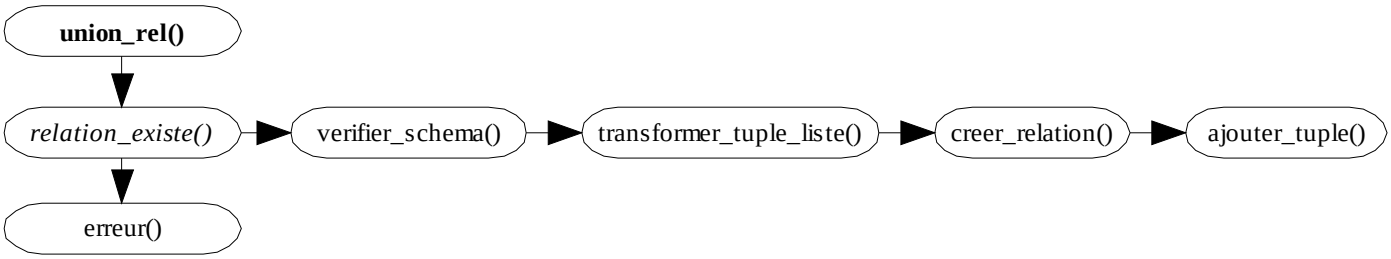

```
int verifier_schema(char * nom_relation1, char * nom_relation2);
```

Vérifie le schéma de deux relations, retourne 1 si le schéma est identique, sinon retourne 0.



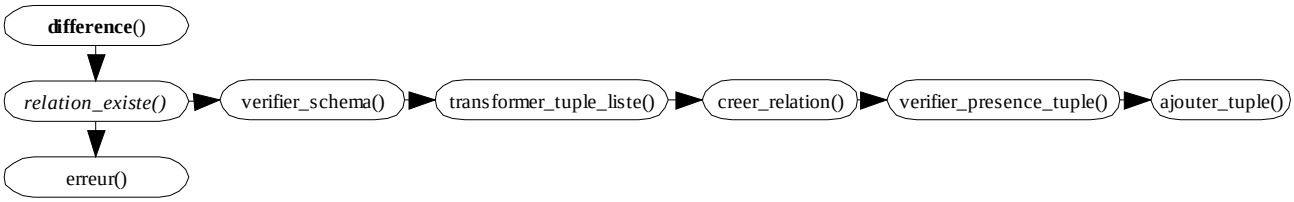

```
char * transformer_tuple_liste(char * nom_relation, int no_tuple);
```

Transforme les valeurs d'un tuple en une liste de valeurs séparées par des virgules.
Retourne l'adresse de la chaîne de caractères.



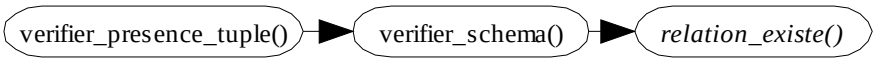

```
void union_rel(char * nom_relation1, char * nom_relation2,char * nom_relation_res);
```

Réalise l'union de deux relations.



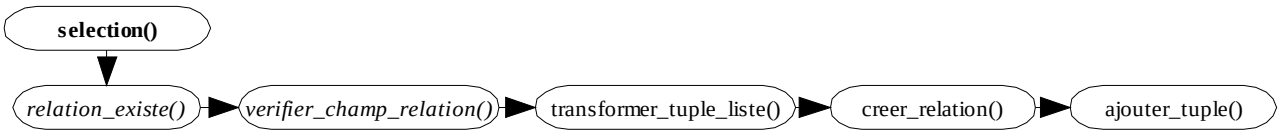

```
void difference(char * nom_relation1, char * nom_relation2, char * nom_relation_res);
```

Réalise la différence entre deux relations.



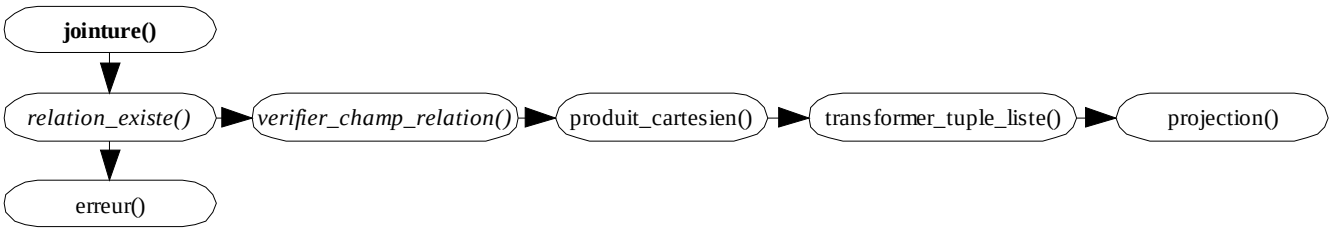

```
int verifier_presence_tuple(char * nom_relation1,char * nom_relation2, int no_tuple);
```

Vérifie l'existence d'un tuple d'une relation1 dans une relation2,
s'il existe retourne le numéro du tuple dans la seconde relation, dans le cas contraire retourne 0.



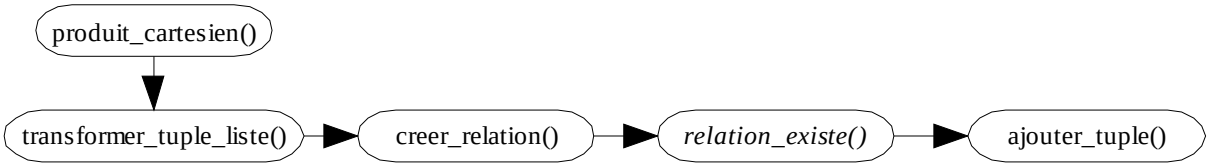

```
void selection(char * nom_relation_init,char * nom_relation_res,
      char * attribut,char operateur,char * valeur_attr) ;
```

Réalise la sélection sur une relation



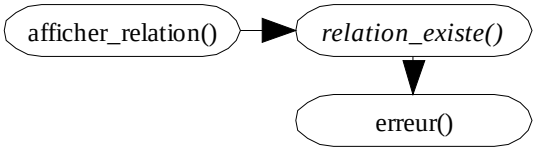

```
void jointure(char * nom_relation1, char * nom_relation2, char * nom_relation_res);
```

Réalise une jointure entre deux relations.




```
void produit_cartesien(char * nom_relation1,char * nom_relation2,char * nom_relation_res);
```

Réalise le produit cartésien entre deux relations.




```
void afficher_relation(char * nom_relation);
```

Affiche la relation dont le nom a été passe en paramètre

Codes sources et fichiers exécutables

Les codes sources joints en annexes évoluent encore et sont disponibles de même que le code compilé et la documentation à l'adresse : www.christiaen.org/sqf/

Guide pour l'utilisateur

Un guide d'utilisation est disponible à l'adresse www.christiaen.org/sqf/ .
Il comprend d'une part les explications relatives à l'utilisation courante de l'application et d'autre part les directives de (re)compilation pour un autre usage.

Table des annexes

Bibliographie

[01] Techniques et outils pour la compilation – Henri Garreta – Faculté des sciences de Luminy – Janvier 2001 - http://www.dil.univ-mrs.fr/~garreta/Polys/PolyCompil.pdf

[02] SQL et Algèbre relationnelle : notions de base – Jérôme Gabillaud – 2004 - Editions ENI

[03] Best of Langage C – Claude Delannoy - 2002 - Ed Eyrolles

[04] Programming Language Concepts – C. Ghezzi & M. Jazayeri – 3th Ed – Ed Wiley

[05] Bases de données - Jef Wijsen – UMH – Sep 2001

[06] Lex & Yacc – JR. Levine, T. Mason & D. Brown – Ed O'Reilly

[07] Compilateurs : principes, techniques et outils – A. Aho, R. Sethi, J Ullman – Ed Dunod

Annexe 1 :

énoncé du projet

Annexe 2

code source Flex

sqf.l

Ce code évolue encore et est disponible à l'adresse : www.christiaen.org/sqf

Annexe 3

code source Bison

sqf.y

Ce code évolue encore et est disponible à l'adresse : www.christiaen.org/sqf

Annexe 4

code source du fichier de définitions des structures de données et fonctions

sqf.h

Ce code évolue encore et est disponible à l'adresse : www.christiaen.org/sqf

Annexe 5

code source des fonctions de manipulations de données

sqf.c

Ce code évolue encore et est disponible à l'adresse : www.christiaen.org/sqf