HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

RAPPORT INTERMÉDIAIRE

Sémantique formelle des contraintes qualitatives

Loïc HERMAN

Rapport intermédiaire sur la théorie du langage

StandardQL

pour la réalisation du

Moteur de validation des contraintes de qualité du Registre fédéral des Bâtiments et des Logements (RegBL)

dans le cadre du

Développement de l'implémentation cantonale du registre dans le système d'information des bâtiments (SIBAT)

pour la

Direction du Cadastre et de la Géoinformation (DCG)

17 avril 2025

HEIG VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Table des matières

1 La définition des attributs et leur qualité dans un registre fédéral

1.1 Le Registre fédéral des Bâtiments et des Logements

Le Registre fédéral des bâtiments et des logements (RegBL) [1] a été établi par l'Office fédéral de la statistique (OFS) suite au recensement de la population de l'an 2000. Ce registre constitue aujourd'hui une infrastructure de données fondamentale pour la gestion du territoire suisse, répertoriant l'ensemble des bâtiments et des logements présents sur le territoire national.

Le RegBL s'inscrit dans un cadre juridique précis, fondé sur l'article 10, alinéa 3bis de la loi sur la statistique fédérale, l'ordonnance sur le Registre des bâtiments et des logements (ORegBL), ainsi que sur l'ordonnance sur les relevés statistiques. Ces bases légales définissent les objectifs, la structure et les obligations liées à la tenue de ce registre.

Selon l'article 2 de l'ORegBL, un bâtiment est défini comme "une construction immobilière durable couverte, bien ancrée dans le sol, pouvant accueillir des personnes et utilisée pour l'habitat, le travail, la formation, la culture, le sport ou pour toute autre activité humaine". Cette définition précise sert de fondement à l'identification et à la classification des objets répertoriés dans le registre.

Le RegBL joue un rôle essentiel dans la mise en œuvre du principe "once-only" de l'administration fédérale, qui vise à ce que la collecte de données par l'administration ne soit effectuée qu'une seule fois auprès d'une même personne ou entité. Cette approche permet d'éliminer les contradictions entre les différentes autorités et de garantir une cohérence dans les données utilisées à tous les niveaux administratifs (fédéral, cantonal et communal).

1.2 Ampleur et complexité du RegBL

L'envergure du RegBL est considérable. Le registre définit 7 entités principales (Service d'enquête, Projet de construction, Travaux, Bâtiment, Entrée de bâtiment, Logement, Rue) et environ 800 caractères (attributs) répartis sur ces entités. Ces caractères comprennent :

— Des attributs obligatoires : identifiants uniques, coordonnées géographiques, adresses
— Des attributs structurels : nombre d'étages, surface habitable, type de chauffage

1. https://www.housing-stat.ch/

Listing 1 Pseudocode de la règle BQ9980

```
GSTAT=1007
WHERE GABBJ NOT NULL
```

— Des attributs temporels : dates de mise à jour, historique des modifications
— Des relations spatiales : commune, quartier, parcelle cadastrale

La complexité du registre s'illustre également par la diversité des types de bâtiments qu'il répertorie. On distingue notamment :

— Les bâtiments exclusivement à usage d'habitation (maisons individuelles, immeubles résidentiels)
— Les bâtiments d'habitation avec usage annexe (commerces, bureaux)
— Les bâtiments partiellement à usage d'habitation (écoles avec logement de concierge)
— Les bâtiments sans usage d'habitation (bâtiments industriels, commerciaux)
— Les habitations provisoires (mobil-homes, roulottes)
— Les constructions particulières (hangars ouverts, arrêts de transports publics couverts)

1.3 Les contraintes de qualité : une nécessité pour la fiabilité des données

Face à l'ampleur et à la complexité de ce référentiel, l'OFS a défini un ensemble d'environ 500 règles de qualité. Ces règles sont essentielles pour garantir la cohérence, l'exactitude et la fiabilité des données enregistrées dans le RegBL. Elles se répartissent en quatre catégories principales :

— **Automatismes** : règles déclenchant des actions automatiques dans le système
— **Obligations d'annonce** : règles définissant les informations devant obligatoirement être renseignées selon certains critères
— **Exigences de qualité** : validations de cohérence et contraintes d'intégrité entre les différents attributs
— **Spécifications techniques** : définitions formelles du domaine des attributs

Ces règles répondent à l'article 3, alinéa 2 de l'ORegBL, qui stipule que « l'OFS, après avoir consulté les cantons, fixe les exigences minimales de qualité s'appliquant aux données du RegBL fédéral ».

Pour faciliter la compréhension et l'application de ces règles, l'OFS a développé un pseudocode comparable à du SQL, permettant une expression déclarative des contraintes. On affiche dans le listing 1 en exemple comment est décrite la règle BQ9980 dans les spécifications du registre. Cette règle indique que le statut d'un bâtiment (`GSTAT`) doit être "démoli" (1007) lorsqu'une année de démolition (`GABBJ`) est renseignée.

2 Formalisation d'une grammaire et des règles de typage du langage StandardQL

2.1 Introduction et objectifs du langage

Dans le chapitre précédent, nous avons exploré le contexte du Registre fédéral des bâtiments et des logements (RegBL) et la nécessité d'un système formalisé de contraintes de qualité. Nous introduisons maintenant StandardQL, un langage déclaratif conçu spécifiquement pour l'expression de ces contraintes de qualité.

Le paradigme développé pour la création de StandardQL s'inspire du paradigme de la programmation logique sous contraintes, tout en maintenant une syntaxe familière pour les utilisateurs habitués au pseudo-code SQL-like employé par l'OFS. Ce langage vise à combler le fossé entre l'expression informelle des règles de qualité et leur implémentation technique, en offrant un cadre formel, mais accessible.

2.1.1 Objectifs du langage

StandardQL est développé avec plusieurs objectifs clairs :

1. **Accessibilité pour les non-spécialistes** : Le langage doit être compréhensible et utilisable par des personnes ayant une connaissance minimale en informatique, permettant aux experts métier de formuler et d'ajuster les règles selon leurs besoins.
2. **Proximité avec le pseudo-code existant** : Pour faciliter l'adoption, la syntaxe de StandardQL est intentionnellement proche du pseudo-code SQL-like déjà utilisé par l'OFS pour documenter les contraintes de qualité.
3. **Expressivité ciblée** : Le langage offre des constructions syntaxiques spécifiques, comme les « Conditions », qui facilitent l'expression de contraintes sur un attribut unique, cas d'usage fréquent dans le contexte du RegBL.
4. **Extensibilité** : StandardQL est conçu pour être suffisamment flexible pour modéliser n'importe quelle contrainte requise, même si cela nécessite parfois une reformulation de la règle originale.

2.1.2 Non-objectifs du langage

Il est également important de clarifier ce que StandardQL ne cherche pas à accomplir :

1. **Traduction automatique complète** : Le langage ne vise pas à permettre une traduction directe et parfaite de toutes les règles existantes. Il est attendu que les utilisateurs ajustent ou reformulent certaines règles pour les adapter à la syntaxe du langage.
2. **Incorporation des règles d'automatisme** : Les règles classifiées comme « Automatisme » dans le RegBL, destinées à être exécutées en lots pour déclencher des actions spécifiques, ne sont pas intégrées dans la version actuelle du langage. Notons toutefois qu'ajouter ce concept serait relativement simple, en introduisant un nouveau type de déclaration.

2.2 Paradigme des contraintes de qualité

StandardQL s'inscrit dans un paradigme particulier de validation de données, où chaque règle exprime une contrainte que les données doivent satisfaire pour être considérées comme valides. Ce paradigme présente plusieurs avantages pour la vérification de la qualité des données du RegBL.

2.2.1 Approche déclarative

Le langage adopte une approche déclarative plutôt qu'impérative : au lieu de spécifier comment vérifier une contrainte, l'utilisateur déclare ce que la contrainte doit satisfaire. Cette approche permet de se concentrer sur la logique métier plutôt que sur les détails d'implémentation.

L'avantage principal d'une telle approche est qu'elle élimine la nécessité d'intervention des développeurs pour des tâches fondamentales comme la modification de valeurs ou l'ajustement de règles. Dans un contexte impératif, même des changements mineurs nécessiteraient du temps de développement, car on ne peut raisonnablement attendre des personnes orientées métier qu'elles maîtrisent la programmation traditionnelle.

2.2.2 Concept d'identifiants comme fonctions

Une caractéristique distinctive de StandardQL est sa conception des identifiants comme des fonctions. Lorsqu'un identifiant (comme `GSTAT` pour le statut d'un bâtiment) est mentionné dans une règle, il est interprété comme une fonction qui, appliquée à l'entité en cours d'évaluation, retourne la valeur correspondante.

Cette approche permet une gestion élégante des relations entre entités. Par exemple, si une règle concernant un bâtiment fait référence à un attribut d'une entité liée (comme un

Program	$\mathscr{P}$	::=	$\mathcal{S}$ `EOF`	constraint
Statement	$\mathcal{S}$	::=	e	simple rule
		|	e `WHERE` e	filtered rule

FIGURE 2.1 – Structure du langage externe de StandardQL

logement), l'identifiant peut retourner une liste de valeurs extraites de ces entités liées, qui peut ensuite être manipulée par des fonctions d'agrégation.

2.2.3 Extensibilité via le contexte d'exécution

Les fonctions disponibles dans un programme StandardQL sont entièrement déterminées par le contexte d'exécution. Une implémentation standard fournit des fonctions utilitaires communes (comme `COUNT`, `SUM`, `LENGTH`), mais le programme qui utilise le langage peut déclarer des fonctions supplémentaires selon ses besoins.

Cette approche garantit une grande flexibilité, permettant d'adapter le langage à différents contextes d'utilisation tout en maintenant une syntaxe cohérente.

2.3 Grammaire formelle du langage

La grammaire de StandardQL est conçue pour être rigoureuse mais intuitive, permettant autant une analyse syntaxique efficace qu'une utilisation aisée par des non-spécialistes. Nous présentons ci-dessous la grammaire formelle du langage, exprimée en notation EBNF.

2.3.1 Structure générale

Un programme StandardQL se compose d'une unique déclaration (Statement), qui représente une règle de qualité, suivie de la fin du fichier. Cette déclaration comprend une expression principale et, optionnellement, une expression de filtrage introduite par le mot-clé `WHERE`. La structure est donnée en figure 2.1.

2.3.2 Expressions

Les expressions dans StandardQL décrites en figure 2.2 suivent une hiérarchie classique avec des priorités d'opérateurs. Les opérateurs logiques (`AND`, `OR`) et relationnels (`=`, `!=`, $<$, $>$, etc.) permettent de combiner des expressions plus simples. Le langage introduit également le concept de ConditionExpr, qui permet d'exprimer des contraintes sur un attribut spécifique de manière concise. La table en figure 2.3 décrit les termes primaires du langage, ce sont les termes pour lesquels il n'est pas nécessaire de spécifier la préséance relative d'autres termes primaires, car la syntaxe ne sera jamais ambiguë.

Expression	e	::=	e_L	simple term
		|	e `OR` e_L	or expression
LogicalTerm	e_L	::=	e_R	simple term
		|	e_L `AND` e_R	and expression
RelationalExpr	e_R	::=	e_C	simple term
		|	e_C `=` e_C	equality
		|	e_C (`!=` | `<>`) e_C	inequality
		|	e_C (`<` | `<=` | `>` | `>=`) e_C	comparison
		|	e_C (`+` | `-`) e_C	arithmetic I
		|	e_C (`*` | `/`) e_C	arithmetic II
ConditionExpr	e_C	::=	$\mathcal{P}$	primary term
		|	$\mathcal{I}$ $\wedge_c$	condition on term

FIGURE 2.2 – Expressions du langage externe de StandardQL

Primary	$\mathcal{P}$	::=	$\supset_{\mathrm{E}}$	function call
		|	v	valid value
		|	`(` e `)`	inner expression
FunctionCall	$\supset_{\mathrm{E}}$	::=	$\mathcal{I}$ `(` [ArgumentList] `)`	function call
ArgumentList		::=	($\mathcal{I}$ | v | $\supset_{\mathrm{E}}$)	single argument
		|	($\mathcal{I}$ | v | $\supset_{\mathrm{E}}$) `,` ArgumentList	multiple arguments

FIGURE 2.3 – Termes primaires du langage externe de StandardQL

ConditionList	$\wedge_c$	::=	c	simple condition
		|	c `AND` $\wedge_c$	and condition
		|	c `OR` $\wedge_c$	or condition
Condition	c	::=	[`IS`] `NULL`	nullity
		|	[`IS`] `NOT NULL`	non-nullity
		|	`=` v	equality
		|	(`!=` | `<>`) v	inequality
		|	(`<` | `<=` | `>` | `>=`) v	comparison
		|	`BETWEEN` v `AND` v	simple domain
		|	`IN (` $\cup_v$ `)`	value domain
		|	`NOT IN (` $\cup_v$ `)`	inverse value domain
		|	[`MATCHES`] `(` e `)`	closure-bound inner expression
ValueList	$\cup_v$	::=	v [`-` v]	single range
		|	v [`-` v] `'|'` $\cup_v$	multiple ranges

FIGURE 2.4 – Conditions du langage externe de StandardQL

$$
\begin{array}{lcll}
\text{Value} & v & ::= & \text{NumericLiteral} \\
 & & \mid & \text{StringLiteral} \\
\text{Identifier} & \mathcal{I} & ::= & \text{Letter } \{ \text{ Letter } \mid \text{ Digit } \mid \text{ \_ } \}
\end{array}
$$

FIGURE 2.5 – Identifiants et littéraux du langage externe de StandardQL

2.3.3 Conditions

Les conditions représentent des contraintes spécifiques appliquées à un identifiant (attribut). Elles peuvent exprimer des tests de nullité, des comparaisons avec des valeurs, ou des tests d'appartenance à un ensemble ou à un intervalle. Cela constitue un sucre syntaxique par rapport aux expressions de base. Leur rôle est de simplifier l'écriture des contraintes qui s'appliquent à un même attribut, réduisant ainsi la duplication de code. La table en figure 2.4 en donne un exemple prévisionnel.

Ce mécanisme peut être étendu avec des syntaxes supplémentaires tant qu'elles peuvent être réduites à des expressions simples sur l'identifiant concerné. Par exemple, la condition `IN <value-range>` se traduit en arrière-plan par une série d'expressions de comparaison avec des opérateurs logiques.

2.3.4 Identifiants et littéraux

La figure 2.5 définit pour finir la syntaxe des identifiants (noms d'attributs et de fonctions) et des littéraux (valeurs numériques et chaînes de caractères). On ne répète pas dans la syntaxe la notion triviale de nombres et de strings (les littéraux sous forme de strings sont entourés de guillemets et les nombres sont représentés tels quels).

2.3.5 Analysabilité du langage

La grammaire présentée jusqu'ici a été construite pour qu'un parser puisse analyser l'entrée de gauche à droite, en effectuant la dérivation la plus à gauche d'une phrase donnée. Autrement dit, la grammaire fait partie de la classe $\mathrm{LL}(k)$ et peut être analysée par un parser descendant récursif avec k tokens de lookahead.

Dans la version du langage telle que décrite ci-avant, on pourra analyser une phrase avec $k = 2$ tokens de lookahead. La grammaire s'inscrit alors dans la classe $\mathrm{LL}(2)$. Pour le prouver, nous devons vérifier que pour chaque non-terminal avec plusieurs productions, le premier ou le second token de chaque production permet de déterminer de manière unique quelle production utiliser.

On examine alors les non-terminaux avec plusieurs productions :

— Statement : La distinction entre les deux productions se fait sur la présence du mot-clé `WHERE` après la première expression, ce qui est détectable avec un lookahead de 1.

$$\begin{aligned}\tau \quad ::= \quad & \textbf{int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{bool} \mid \textbf{date} \\ \mid \quad & \texttt{[}\tau\texttt{]} \mid \textbf{entity}\texttt{(e)} \mid \textbf{null}\end{aligned}$$

FIGURE 2.6 – Types utilisés dans le typage dynamique de StandardQL

— Expression, LogicalTerm, RelationalExpr : Les opérateurs `OR`, `AND`, et les opérateurs de comparaison permettent de distinguer les productions.
— ConditionExpr : La présence d'une liste de conditions après un identifiant permet de distinguer les deux productions.
— Primary : Le type du token (identifiant, parenthèse ouvrante, littéral) permet de déterminer la production. Il faudra quand même 2 tokens pour distinguer l'appel d'une fonction avec arguments d'une sous-expression parenthésée.
— Condition : Les mots-clés `IS`, `IN`, `BETWEEN` ou les opérateurs de comparaison permettent de distinguer les productions. Les expressions internes aux conditions sont déterminables par la présence de parenthèses sur le deuxième token.

Cette analyse confirme que la grammaire est $\mathrm{LL}(2)$, ce qui garantit une analyse syntaxique efficace et sans ambiguïté par un parser descendant récursif.

2.4 Système de types

Le système de types de StandardQL est conçu pour être simple mais expressif, permettant la manipulation des différents types de données présents dans le RegBL. La figure 2.6 définit les types premiers utilisés par le langage.

Cette section décrira le système prévisionnel de types qui devra être mis en place pour répondre aux exigences placées ci-avant.

2.4.1 Identifiants comme fonctions

Le système de types de StandardQL considère chaque identifiant comme une fonction qui s'applique à l'entité contextuelle courante. Nous formalisons cette notion à travers un système de jugement de types où le contexte d'évaluation joue un rôle central.

$$\frac{\Gamma(\mathrm{id}) = \textbf{entity}(e) \to \tau}{\Gamma \vdash \mathrm{id} : \textbf{entity}(e) \to \tau} \quad \text{(ID)} \tag{2.1}$$

Dans un contexte d'évaluation Γ, un identifiant id est typé comme une fonction qui prend une entité de type e et retourne une valeur de type τ. Le type τ et l'entité e sont déterminés dynamiquement au runtime avec le contexte d'évaluation.

Par exemple, si nous traitons une règle dont l'attribut principal référence un bâtiment. Le jugement suivant s'applique :

$$\Gamma \vdash \texttt{GSTAT} : \textbf{entity}(\texttt{BUILDING}) \to \textbf{int} \tag{2.2}$$

Sous forme textuelle, on dira que l'attribut représentant le statut du bâtiment (`GSTAT`) est disponible dans le contexte Γ et sa fonction associée retourne le statut sous forme d'entier lorsqu'elle est appelée avec le bâtiment en cours de traitement.

Pour les identifiants qui référencent des attributs d'entités liées (notées e_k), nous introduisons une règle d'inférence supplémentaire qui modélise la navigation relationnelle :

$$\frac{\Gamma(\text{id}) = \textbf{entity}(e) \to \textbf{entity}(e_k) \to \tau}{\Gamma \vdash \text{id} : \textbf{entity}(e) \to [\tau]} \quad \text{(REL-ID)} \tag{2.3}$$

Cette règle capture le fait que si un identifiant id est lié dans le contexte Γ à une fonction d'ordre supérieur déclarant que l'attribut mentionné fait partie d'une relation sur une entité liée e_k de l'entité principale e, le programme à l'exécution retournera une liste $[\tau]$ de valeurs correspondant à l'attribut donné. La norme de la liste dépend de la cardinalité de la relation entre les deux entités.

Dans le même exemple qu'avant, le jugement suivant s'applique :

$$\frac{\Gamma(\texttt{WSTAT}) = \textbf{entity}(\texttt{BUILDING}) \to \textbf{entity}(\texttt{DWELLING}) \to \textbf{int}}{\Gamma \vdash \texttt{WSTAT} : \textbf{entity}(\texttt{BUILDING}) \to [\textbf{int}]} \tag{2.4}$$

Cette liaison indique que `WSTAT` est une fonction qui, appliquée à un bâtiment, retourne une fonction qui, appliquée à un logement, retourne un nombre (le statut du logement). Dans le contexte d'une règle qui s'applique aux bâtiments, l'appel à `WSTAT` retourne donc une liste de valeurs numériques représentant les statuts de tous les logements associés au bâtiment.

2.4.2 Fonctions d'agrégation et autres fonctions utilitaires

Les fonctions d'agrégation et utilitaires sont essentielles pour manipuler les données dans StandardQL. Nous formalisons leurs types à travers des règles d'inférence qui spécifient les types d'entrée et de sortie.

$$\frac{\Gamma \vdash f : \tau_1 \times \tau_2 \times \ldots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash f(e_1, e_2, \ldots, e_n) : \tau} \quad \text{(APP)} \tag{2.5}$$

$$\frac{\Gamma \vdash v : \textbf{string}}{\Gamma \vdash \text{LENGTH}(v) : \textbf{int}} \quad \text{(LENGTH)} \tag{2.6}$$

$$\frac{\Gamma \vdash v : [\tau]}{\Gamma \vdash \text{COUNT}(v) : \textbf{int}} \quad \text{(COUNT)} \tag{2.7}$$

$$\frac{\Gamma \vdash v : [\textbf{int}]}{\Gamma \vdash \text{SUM}(v) : \textbf{int}} \quad \text{(SUM)} \qquad \frac{\Gamma \vdash v : [\textbf{float}]}{\Gamma \vdash \text{SUM}(v) : \textbf{float}} \quad \text{(SUMF)} \tag{2.8}$$

$$\frac{\Gamma \vdash v : [\textbf{int}]}{\Gamma \vdash \text{MAX}(v) : \textbf{int}} \quad \text{(MAX)} \qquad \frac{\Gamma \vdash v : [\textbf{float}]}{\Gamma \vdash \text{MAX}(v) : \textbf{float}} \quad \text{(MAXF)} \tag{2.9}$$

$$\frac{\Gamma \vdash v : [\textbf{int}]}{\Gamma \vdash \text{MIN}(v) : \textbf{int}} \quad \text{(MIN)} \qquad \frac{\Gamma \vdash v : [\textbf{float}]}{\Gamma \vdash \text{MIN}(v) : \textbf{float}} \quad \text{(MINF)} \tag{2.10}$$

Les jugements 2.7 à 2.10 permettent les opérations d'agrégation sur les listes. Le jugement 2.6 définit la fonction sur les chaînes de caractères. Enfin, le jugement 2.5 permet la définition d'autres fonctions génériques comme le produit de types vers un type résultant.

Dans le cas particulier des conditions `MATCHES`, une fermeture lexicale est créée pour les appels de fonction qui apparaissent dans l'expression de correspondance. Lorsqu'un identifiant est utilisé avec une condition `MATCHES`, cet identifiant (qui est lui-même une fonction appliquée à l'entité propriétaire) est implicitement fourni comme premier paramètre à tout appel de fonction lié dans l'expression de correspondance qui n'a pas d'argument explicite. Cette règle peut être formalisée comme suit :

$$\frac{\begin{array}{l}\Gamma \vdash \text{id} : \textbf{entity}(e) \to \tau_1 \\ \Gamma \vdash f : \tau_1 \to \tau_2 \\ \Gamma \vdash \text{expr}[f() \mapsto f(id)] : \textbf{bool}\end{array}}{\Gamma \vdash \text{id MATCHES expr} : \textbf{bool}} \quad \text{(MATCHES)} \tag{2.11}$$

Par exemple, dans une expression comme `LPARZ NULL OR (LENGTH<=12)`, la fonction `LENGTH` est appelée avec l'attribut `LPARZ` comme argument, équivalent à `LENGTH(LPARZ)`.

2.5 Optimisation de l'évaluation

Le langage StandardQL et son compilateur utilisent quelques optimisations pour permettre une évaluation plus rapide des règles.

2.5.1 Compilation en représentation intermédiaire

Les règles StandardQL peuvent être compilées en une représentation intermédiaire qui facilite leur évaluation efficace. Une telle représentation intermédiaire éliminera alors, comme mentionné précédemment, toute la notion de Condition dans un processus d'élaboration pour les remplacer en un ensemble facilement interprétable d'expressions. Cela permettra de modéliser dans cette notation les fermetures et la gestion de la navigation relationnelle.

3 Étapes d'implémentation du compilateur pour l'évaluation dynamique des contraintes

L'implémentation technique du langage StandardQL nécessite une approche méthodique pour traduire la grammaire formelle et le système de types définis dans le chapitre précédent en un compilateur fonctionnel. Ce chapitre présente une feuille de route pour le développement de ce compilateur, en détaillant les étapes clés du processus, de l'analyse syntaxique à l'intégration dans un système imitant le projet SIBAT en cours de développement au canton de Vaud.

3.1 Développement du parser en Haskell

La première étape consiste à implémenter un parser pour le langage StandardQL. Le langage Haskell avec la bibliothèque Megaparsec sera utilisé. Plusieurs facteurs motivent ce choix. Haskell, en tant que langage fonctionnel pur, offre un cadre élégant pour l'implémentation d'analyseurs syntaxiques établis à partir des combinateurs de parsers. Megaparsec, en particulier, est une bibliothèque moderne et performante qui facilite la création de parsers robustes avec des messages d'erreur informatifs.

L'approche par combinateurs de parsers permet de construire le parser de manière modulaire, en reflétant directement la structure de la grammaire formelle. Chaque règle de production de la grammaire sera implémentée comme une fonction de parsing distincte, et ces fonctions seront combinées pour former le parser complet.

3.2 Élaboration vers un langage interne

Une fois l'analyse syntaxique réalisée, les constructions de l'AST (Abstract Syntax Tree) du langage externe doivent être transformées en un langage interne plus simple. Ce processus, appelé élaboration, consiste à réduire les sucres syntaxiques à des constructions plus fondamentales.

Par exemple, les conditions comme `BETWEEN` et `IN` seront traduites en expressions booléennes utilisant des opérateurs de comparaison et des conjonctions. De même, les expressions de condition sur un identifiant unique seront normalisées en expressions relationnelles standard. Cette étape simplifie considérablement les phases ultérieures de compilation et d'interprétation, en réduisant le nombre de cas à traiter.

L'élaboration inclura également la vérification des types, en s'assurant que les expressions sont bien typées selon le système formel défini dans le chapitre précédent. Les erreurs de type détectées à ce stade seront signalées avec des messages clairs et précis pour aider les utilisateurs à corriger leurs règles.

3.3 Intégration avec le système SIBAT via une interface Java

Dans le cadre plus agrandi du développement de ce langage, le compilateur sera intégré au moteur de validation des règles existant développé en Java pour remplacer une grande partie du code impératif déjà présent. Cela ne sera pas couvert dans l'implémentation au sein de ce cours.

En revanche, pour pouvoir démontrer une utilisation concrète de ce langage, nous utiliserons une maquette simplifiée de l'API existante dans SIBAT pour tester cette intégration, en simulant les structures de données et les opérations du système réel. Cette maquette servira de banc d'essai pour valider l'approche avant son déploiement dans l'environnement de production.

Une approche pratique pour le transfert des résultats de l'élaboration consiste à sérialiser les représentations intermédiaires générées par le compilateur Haskell dans un format que Java peut facilement consommer. Le système Java pourrait alors interpréter ces représentations pour évaluer les contraintes sur les données réelles.

3.4 Extensibilité et couverture des cas d'utilisation

L'approche proposée consiste à implémenter d'abord un noyau minimal du langage qui couvre les contraintes les plus courantes et les plus simples. Ce noyau sera ensuite étendu progressivement pour incorporer des fonctionnalités plus avancées, en fonction des besoins identifiés dans les règles existantes.

Le système sera conçu pour permettre l'ajout de nouvelles fonctions et opérateurs sans modification majeure du compilateur. Cela sera réalisé en définissant une interface claire pour les fonctions externes et en mettant en place un mécanisme d'enregistrement des fonctions dans le contexte d'exécution.