Gestion des risques et gestion des rendus

Projet GL — Compilateur Deca

7 janvier 2026

I Gestion des risques

La gestion des risques a été prise en compte dès le début du projet afin de limiter les erreurs ayant un impact critique sur la qualité du compilateur ou sur le respect des contraintes académiques. Les risques identifiés concernent à la fois l'organisation du travail en groupe et les aspects techniques liés au développement du compilateur.

I.1 Risque 1 — Retard ou oubli de rendu

Description : Un oubli ou un retard dans les rendus intermédiaires ou finaux constitue un risque majeur du projet.

Impact :

— Pénalités académiques importantes

— Dégradation de l'évaluation globale du projet

Actions mises en place :

— Élaboration d'un planning prévisionnel détaillé

— Désignation d'un responsable pour chaque rendu

— Suivi régulier de l'avancement par le responsable du groupe

I.2 Risque 2 — Conflits Git et perte de code

Description : Le travail collaboratif peut entraîner des conflits Git ou des pertes de modifications.

Impact :

— Perte de travail

— Retards dans le développement

— Instabilité du projet

Actions mises en place :

— Travail collaboratif via Visual Studio Code Live Share

— Synchronisation systématique du dépôt avant toute session de travail

— Utilisation de `git stash` pour éviter les conflits

— Communication préalable avant modification de fichiers critiques

I.3 Risque 3 — Mauvaise interprétation de la grammaire ou des règles contextuelles

Description : Une interprétation incorrecte de la grammaire ou des règles contextuelles peut introduire des erreurs subtiles, difficiles à détecter immédiatement.

Impact :

— Comportements incorrects sur certains programmes Deca

— Erreurs contextuelles non détectées précocement

Actions mises en place :

— Lecture approfondie de la documentation officielle Deca

— Implémentation progressive des règles contextuelles

— Mise en place de tests `.deca` spécifiques aux étapes A et B

— Isolation claire des tests des étapes A et B indépendamment de l'étape C

I.4 Risque 4 — Bugs critiques détectés tardivement

Description : Des bugs critiques découverts tardivement peuvent rendre le compilateur inutilisable au moment du rendu.

Impact :

— Échec du rendu

— Compilateur non fonctionnel

Actions mises en place :

— Exécution régulière des tests automatisés à l'aide de la commande `mvn test`, permettant de détecter les régressions fonctionnelles et contextuelles du compilateur.

— Compilation systématique du projet avec la commande `mvn compile` après toute modification significative du code source. Cette étape permet de vérifier immédiatement la cohérence structurelle du projet. L'enchaînement `mvn compile` puis `mvn test` permet de réduire la zone de détection des erreurs : les problèmes de compilation sont identifiés indépendamment des erreurs fonctionnelles détectées par les tests, ce qui facilite leur localisation et leur correction.

— Ajout progressif de tests unitaires Java (JUnit)

— Utilisation de JaCoCo pour analyser la couverture des tests

I.5 Risque 5 — Régressions non détectées avant push (CI/CD)

Description : Une modification poussée sur le dépôt peut introduire une régression sans être détectée immédiatement (tests cassés, rapports incohérents), ce qui déstabilise le travail du groupe et augmente le risque d'échec en fin de projet.

Impact :

— Dépôt distant instable (pipeline en échec)

— Blocage ou ralentissement du travail collaboratif

— Risque de livrer un rendu avec des tests non validés

Actions mises en place :

— Mise en place d'une intégration continue via `.gitlab-ci.yml` déclenchée à chaque `push`.

— Exécution systématique des tests Maven (`mvn test`) et des batteries de tests par étape : lexing, syntaxe, contexte, génération de code et tests différentiels.

— Utilisation de scripts Bash spécialisés (`test_lex_energy.sh`, `test_syntax_energy.sh`, `test_context_energy.sh`, `test_gencode_energy.sh`, `test_differential_energy.sh`) qui déterminent de manière intelligente quels tests relancer selon les fichiers modifiés (relance partielle si seuls des tests changent, relance complète si le compilateur change).

— Génération et publication automatique de rapports (`src/test/script/report`) sous forme de pages GitLab.

II Gestion des rendus

Même avec une base de tests automatisée, certaines actions doivent être réalisées manuellement avant chaque rendu afin de limiter les risques liés aux erreurs de manipulation ou aux oublis.

II.1 Procédure avant rendu

— Synchronisation du dépôt avec `git pull` et vérification de l'état du dépôt distant

— Exécution complète de `mvn clean test` pour valider le comportement fonctionnel du compilateur

— Vérification des tests `.deca` (programmes valides et invalides)

— Test du projet sur une version fraîchement clonée du dépôt

— Validation finale par le responsable du rendu

II.2 Validation CI avant rendu

En complément des vérifications locales, nous utilisons une intégration continue déclenchée à chaque `push`. Le pipeline exécute `mvn test` ainsi que des scripts Bash dédiés à chaque étape (lexing, syntaxe, contexte, génération de code, différentiel). Ces scripts appliquent une stratégie *économe* : ils identifient les tests à relancer en fonction des fichiers modifiés (relance partielle si seuls des tests changent, relance complète si le code du compilateur change). Les rapports générés sont archivés et publiés automatiquement via les pages GitLab.

Cette procédure vise à garantir que le projet est cohérent, compilable et fonctionnel au moment du rendu, indépendamment de l'environnement de développement utilisé.