



Rapport de projet de compilation Interprète pour Kawa

Mohamed Abderraouf MEDJADJ
Zineddine KERMADJ

15 janvier 2025



Table des matières

1	Introduction	3
2	Organisation	3
3	Structure et composantes	4
3.1	Arborescence	4
3.2	Description des modules importants	5
4	Progression et Réalisations	6
4.1	Analyse Syntaxique	6
4.2	Vérification des Types	6
4.3	Interprétation	6
4.4	Extensions	6
5	Détails du développement	7
5.1	Extensions - Les plus “problématiques”	8
5.2	Justifications de certains choix :	12
5.2.1	Transtypage (TypeCast)	12
5.2.2	Structures (classes, méthodes, program)	12
5.2.3	Pourquoi pas <code>visibility</code> ?	13
6	Conclusion	15
7	Sources	16

1 Introduction

L'objectif de ce projet est de concevoir un interprète pour un langage objet simplifié, appelé Kawa, qui s'inspire des concepts fondamentaux de Java. Ce projet se structure en plusieurs étapes principales : l'analyse syntaxique, la vérification des types et l'interprétation, chacune visant à implémenter progressivement les éléments nécessaires au fonctionnement du langage.

Le projet permet de :

- Définir des variables et des classes.
- Utiliser des méthodes pour manipuler des objets.
- Exécuter un bloc principal de code.

2 Organisation

L'organisation du projet repose sur l'utilisation d'un dépôt GitHub dédié. Une stratégie rigoureuse de gestion de branches a été adoptée pour structurer le développement et assurer la qualité du code. Chaque fonctionnalité ou extension du projet est implémentée sur une branche distincte. Cette méthode permet d'isoler les développements et d'effectuer des tests indépendants, garantissant ainsi une meilleure traçabilité et une gestion efficace des modifications.

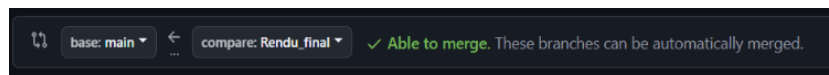
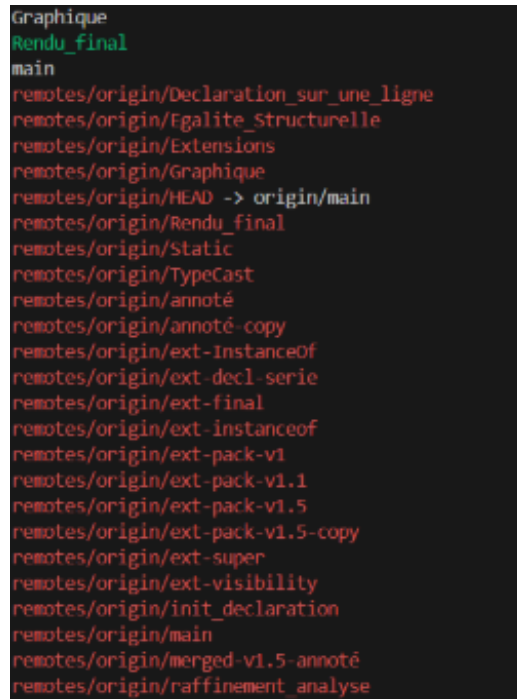


FIGURE 1 – Exemple d'utilisation du dépôt Git

Lorsqu'une extension est terminée, son code est soigneusement revu, et après validation, la branche correspondante est fusionnée (merged) dans la branche principale. Ce processus offre les avantages suivants :

- Clarté et organisation : Le découpage par branches facilite la compréhension du développement de chaque fonctionnalité et réduit les risques de conflits entre plusieurs tâches en cours.
- Collaboration améliorée : Cette stratégie nous a permis de travailler simultanément sur des fonctionnalités distinctes sans se gêner mutuellement.
- Contrôle de la qualité : Avant la fusion de tout nouveau code, des tests et une révision approfondie ont été réalisés pour s'assurer de l'absence de bugs ou de problèmes.



```
Graphique
Rendu_final
main
remotes/origin/Declaration_sur_une_ligne
remotes/origin/Egalite_Structurelle
remotes/origin/Extensions
remotes/origin/Graphique
remotes/origin/HEAD -> origin/main
remotes/origin/Rendu_final
remotes/origin/Static
remotes/origin/TypeCast
remotes/origin/annoté
remotes/origin/annoté-copy
remotes/origin/ext-InstanceOf
remotes/origin/ext-decl-serie
remotes/origin/ext-final
remotes/origin/ext-instanceof
remotes/origin/ext-pack-v1
remotes/origin/ext-pack-v1.1
remotes/origin/ext-pack-v1.5
remotes/origin/ext-pack-v1.5-copy
remotes/origin/ext-super
remotes/origin/ext-visibility
remotes/origin/init_declaration
remotes/origin/main
remotes/origin/merged-v1.5-annoté
remotes/origin/raffinement_analyse
```

FIGURE 2 – Ensemble des branches (github)

En résumé, cette approche a assuré une progression ordonnée du projet tout en minimisant les risques liés à l'intégration de nouvelles fonctionnalités.

3 Structure et composantes

Dans cette section, nous présentons l'organisation structurelle du projet afin de mieux comprendre son architecture. Cette présentation est divisée en deux parties : une vue d'ensemble de l'arborescence complète du projet, suivie d'une description détaillée des modules essentiels qui composent l'application.

3.1 Arborescence

L'arborescence du projet offre une vue hiérarchique des fichiers et dossiers qui constituent l'application. Elle met en évidence les relations entre les différents composants. Voici l'arborescence complète de notre projet :

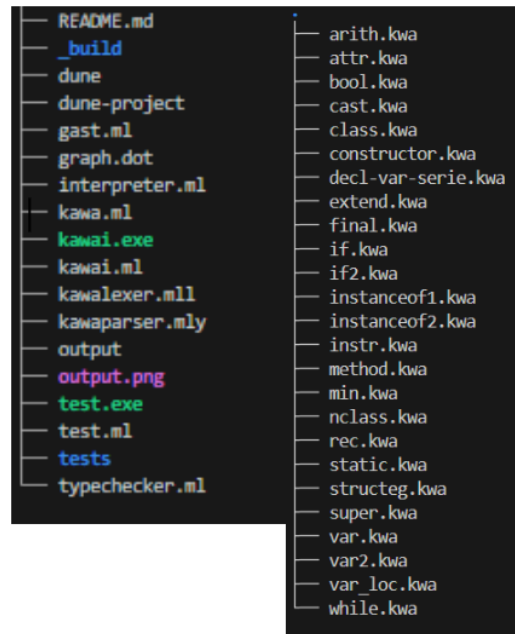


FIGURE 3 – Arborescence du projet - Codes, outputs et fichiers tests

3.2 Description des modules importants

Le projet est organisé en plusieurs fichiers essentiels :

Fichier	Description
kawa.ml	Définit la syntaxe abstraite du langage.
kawalexer.mll	Gère l'analyse lexicale.
kawaparser.mly	Assure la vérification des types.
typechecker.ml	Assure la vérification des types.
interpreter.ml	Implémente l'interprétation des programmes.
kawai.ml	Constitue le programme principal.
tests/*	Contient les fichiers de tests (N.B. certains tests fournis en squelette ont été modifiés).
test.ml	Code qui exécute tous les tests du dossier tests/*.
gast.ml	Génère un graphe illustrant tous les composants d'un programme et leurs types.
output.png	Image générée représentant le graphe AST.

TABLE 1 – Description des principaux modules

4 Progression et Réalisations

Durant le développement de ce projet, nous avons choisi une méthode de travail qui consistait à avancer par blocs fonctionnels complets. Contrairement à une approche linéaire (comme coder d'abord tout le lexer, puis tout le parser), nous avons préféré finaliser chaque fonctionnalité du projet dans son ensemble avant de passer à la suivante. Cela nous a permis d'avoir une vue d'ensemble plus cohérente et de valider régulièrement nos progrès. Chaque cycle suivait des étapes clés : analyse des besoins, conception, implémentation, test et validation. Les principales réalisations du projet peuvent être réparties en quatre grandes sections :

4.1 Analyse Syntaxique

D'abord, pour construire un arbre de syntaxe abstraite à partir d'un programme source, nous sommes passés par :

- Complétion des fichiers `kawalexer.ml1` et `kawaparser.mly`.
- Gestion des priorités des opérations afin d'éliminer les ambiguïtés (0 conflit détecté).

4.2 Vérification des Types

Ensuite, dans le but de garantir la cohérence des types dans le programme, nous avons effectué les tâches suivantes :

- Implémentation des fonctions principales dans `typechecker.ml`.
- Vérification des sous-types ainsi que de la cohérence des retours des méthodes et des diverses instructions.

4.3 Interprétation

Enfin, pour permettre l'exécution des instructions principales du programme, il a fallu s'intéresser aux parties suivantes :

- Développement de la fonction `exec_prog` et de ses fonctions auxiliaires (`eval_expr`, `exec_seq`, `exec_instr`) dans le fichier `interpreter.ml`.
- Gestion de la mémoire pour les variables globales et locales, ainsi que pour les attributs des objets.

4.4 Extensions

Nous avons ajouté plusieurs fonctionnalités pour enrichir le projet :

- Support des champs immuables.
- Gestion des déclarations en série.
- Déclarations avec des valeurs initiales.
- Introduction des champs statiques.
- Test de type.
- Support du transtypage (syntaxe : `expr_obj : (typeclass)`, voir `./tests/cast.kwa`).
- Gestion de l'héritage via `super`.
- Ajout du support pour les tableaux et matrices.
- Égalité structurelle.
- Détection des erreurs comme :
 - Point-virgule manquant ("Missing semicolon").
 - Parenthèse non fermée ("Unclosed parenthesis").
- Suggestions pour certaines erreurs, par exemple : "Did you mean 'recursion'?".
- Prise en charge des structures de syntaxe abstraite typée.
- Génération graphique de l'AST pour un programme test (nécessite les installations suivantes) :

```
opam install ocamlgraph
sudo apt install graphviz
```
- Support des conditions `if` sans branche `else`.
- Automatisation des tests grâce au fichier `test.ml`

5 Détails du développement

Ce projet visait à concevoir un interprète fonctionnel et extensible, tout en privilégiant une approche modulaire. Nous avons avancé par blocs fonctionnels, finalisant chaque composante avant de passer à la suivante, garantissant ainsi une intégration cohérente.

La progression reposait sur quatre phases : conception, implémentation, tests automatisés et ajustements. Chaque étape a renforcé la stabilité du système et facilité l'ajout d'extensions. Ces extensions, parfois audacieuses comme le transtypage ou la gestion des tableaux, ont enrichi le projet en offrant des solutions à des problématiques complexes.

Malgré le choix de privilégier certains concepts avancés au détriment d'autres plus simples, ce projet témoigne de nos progrès techniques, méthodologiques et de notre capacité à résoudre des défis pratiques tout en garantissant la qualité et la cohérence globale.

5.1 Extensions - Les plus “problématiques”

Certaines extensions nous ont posé énormément de problèmes. En voici quelques unes :

- **Déclaration avec valeur initiale** : Cette extension est loin d’être la modification la plus complexe en termes de logique ou de complexité algorithmique. Cependant, elle s’est révélée particulièrement pénible en raison de la nécessité de gérer cette fonctionnalité de manière flexible tout en maintenant la structure et la cohérence du projet.

```
type class_def = {  
    class_name: string;  
    attributes: (string * typ * (expr option)) list;  
    methods: method_def list;  
    parent: string option;  
}
```

FIGURE 4 – Ancienne définition de class

Nous avons opté pour une structure spécifique, détaillée plus loin dans ce document, qui nous a permis de concilier à la fois la simplicité de l’extension et l’intégration fluide au reste du projet.

- **Tableaux et matrices** : Toujours plus avec cette extension ! Lorsqu’il ne s’agit pas de gérer l’accès aux données, c’est l’affectation qui pose problème. Lorsque ce n’est pas l’affectation, c’est la création de nouveaux éléments qui devenait délicate. Et lorsqu’enfin tout semble fonctionner comme prévu, voilà qu’une simple modification entraîne une mise à jour de toute la colonne d’une matrice... Bref, c’est un véritable calvaire ! Mais chaque étape nous a permis d’affiner notre approche pour rendre le tout plus robuste et flexible.


```

(* Fonction récursive pour créer un tableau multidimensionnel *)
let rec create_nested_array dims =
  match dims with
  | [] -> failwith "Dimensions list cannot be empty"
  | [dim] -> (
    match typ with
    | TInt -> VArray (Array.init dim (fun _ -> VInt 0))
    | TBool -> VArray (Array.init dim (fun _ -> VBool false))
    | TVoid -> VArray (Array.init dim (fun _ -> Null))
    | TClass _ -> VArray (Array.init dim (fun _ -> Null))
    | _ -> error ("Can't handle this data type")
  )
  | dim :: rest ->
    VArray (Array.init dim (fun _ -> create_nested_array rest ))

```

FIGURE 5 – Fonction `create_nested_array` qui s'occupe de la création de tableaux et de matrices

Nous avons finalement réussi à tout bien implémenter pour cette structure de données en initialisant chaque tableau créé de dimension >0 et avec au moins un élément avec des valeurs par défaut qui seront, bien évidemment, modifiées à la guise du programmeur.

Les tests sont effectués dans `./tests/array.kwa`.

- **Transtypage** : Cette extension, qui semblait particulièrement complexe à implémenter, a été retardée jusqu'à la fin de l'annotation. Toutefois, ce délai nous a finalement été bénéfique, car l'annotation a offert un regard neuf et objectif sur le problème. Ce point de vue externe a permis d'identifier rapidement les difficultés que nous rencontrions au niveau du typage/interprétation et a contribué à résoudre efficacement tous les problèmes qui auraient pu autrement entraver l'avancée du projet.

Nous détaillons les choix effectués pour le parsing dans la section suivante.

- **"Did you mean 'recursion'?"** : Cette extension a été particulièrement agréable à implémenter. En effet, c'était un véritable plaisir d'explorer et de mettre en œuvre différentes méthodes de mesure de similarité, telles que la distance de Sørensen-Dice, la distance de Monge-Elkan, ainsi que la distance de Levenshtein (aussi bien dans sa

version classique qu'avec pénalités additionnelles). Chaque approche a apporté des perspectives nouvelles sur la comparaison de chaînes, offrant une occasion unique d'expérimenter avec des concepts mathématiques intéressants tout en les appliquant à un cas concret.

```
❯ zier@LAPTOP-DK4MF1K2:~/finaltest/kawa-interpretor$ ./kawai.exe ./tests/nclass.kwa
type error: Class not found: sefment, did you mean: segment?
```

FIGURE 6 – Erreur de frappe (utilisation de la distance de Levenshtein)

Nous avons finalement choisi l'approche de Levenshtein dans sa forme classique qui offre un équilibre optimal entre performance et précision.

- **Syntaxe abstraite typée** : Au départ, cette extension n'était pas censée en être une, car nous pensions que la tâche serait trop difficile. En réalité, c'était plutôt une question de procrastination. Cependant, une fois que nous nous y sommes mis, non seulement elle s'est avérée bien moins complexe que prévu, mais elle a également inspiré deux autres extensions. En fin de compte, cette initiative s'est révélée très bénéfique pour l'évolution du projet.

Nous typons donc le maximum pendant le parsing sans en abuser afin de récupérer le plus d'informations possibles sur les types. Ensuite, le reste du typage est effectué pendant l'étape de typage proprement dite, là où les vérifications sont plus approfondies et les types sont définitivement déterminés.

```
| Unop (Opp, e1) ->
  (match type_expr e1 tenv with
  | TInt -> TInt
  | ty -> type_error ty TInt)
```

FIGURE 7 – Ast

```
| Unop (u, e) -> (
  let typed_e = type_expr e tenv in
  match u with
  | Opp ->
    check_eq_type TInt typed_e.annot;
    {annot = TInt; expr = Unop(u, typed_e); loc = e.loc})
```

FIGURE 8 – Ast annoté

- **Dessin graphique de l'AST d'un programme** : Une extension supplémentaire que nous avons trouvée particulièrement intéressante consiste à dessiner un graphe représentant tous les composants d'un programme. Ce graphe sert à la fois de moyen de visualisation et de vérification, permettant ainsi de s'assurer que tous les éléments du programme interagissent comme prévu. Cette approche facilite la

détection d'éventuelles erreurs logiques et aide à mieux comprendre la structure globale du programme.

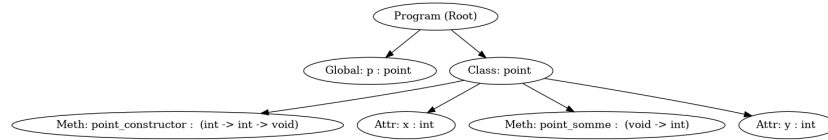


FIGURE 9 – Output du test method.kwa

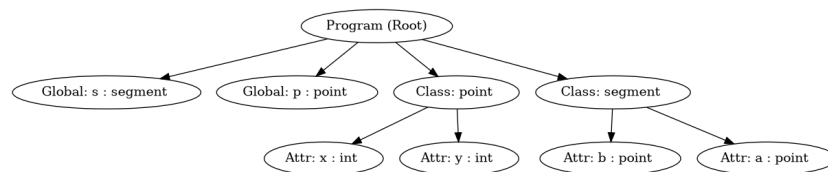


FIGURE 10 – Output du test nclass.kwa

Attention, vous devrez exécuter ces lignes de commandes : `| opam install ocamlgraph | | oo apt install graphviz`

Une légère modification a été apportée au fichier `dune` afin de faire fonctionner le code `gast.ml`.

- **Le processus ne peut pas aboutir en raison d'un problème technique** : Résultat quasi-direct de l'annotation de l'ast, c'est exactement le même principe sauf qu'on sauvegarde l'emplacement du buffer au moment du parsing dans un champ au lieu d'un type. C'était tout de même très frustrant de comprendre qu'il fallait rajouter l'instruction : `set filename lb file` dans `kawai`.
- **Fichier test.ml** : Probablement notre extension préférée, elle est discrète tout en nous faisant gagner des heures de travail et préservant notre santé mentale. Ce code s'occupe de l'ensemble des tests à notre place, nous permettant ainsi de nous concentrer sur l'essentiel du projet, sans être distraits par des vérifications répétitives et parfois fastidieuses. Grâce à cette automatisation, nous avons gagné en efficacité et réduit considérablement les risques d'erreur. De plus, elle nous permet de garantir qu'en ajoutant de nouvelles fonctionnalités, nous ne corrompons pas celles qui existent déjà, en validant que tout reste opérationnel après chaque modification. Cette

sécurité nous donne une grande tranquillité d'esprit, nous assurant que chaque évolution du projet ne porte pas atteinte aux fonctionnalités précédemment implémentées. Pour cela, il faut exécuter la ligne de commande : `ocamlc -o test.exe unix.cma test.ml; rm test.cmi test.cmo; ./test.exe`

5.2 Justifications de certains choix :

Dans ce projet, certains choix de conception pourraient sembler controversés, mais ils reposent sur des considérations réfléchies pour maximiser l'efficacité et la pérennité du système. Voici nos justifications :

5.2.1 Transtypage (TypeCast)

Le typecast a été codé de manière assez particulière. La syntaxe choisie pour ce dernier est : `expr_obj : (typeclass) print((pt : (triple)).z) ;`

Cette approche offre une distinction explicite et simplifie considérablement l'analyse syntaxique, en éliminant le risque de confusion entre un typecasting et d'autres constructions telles que des expressions parenthésées.

Avant de finaliser ce choix, nous avons réalisé des recherches approfondies et étudié différentes solutions documentées pour gérer cette ambiguïté courante dans les langages similaires. Voici un résumé des pistes envisagées :

- Lexer Hack : Modification du lexer pour identifier si le contenu entre parenthèses est un type ou une expression, au prix d'une complexité accrue.
- Distinction dans la grammaire : Création de règles distinctes pour les casts et les parenthèses, mais avec une grammaire plus lourde.
- Priorités dans le parseur : Attribution de priorités et règles de précedence pour lever les conflits, efficace mais exigeant des ajustements subtils.

La syntaxe a été retenue pour sa simplicité, sa clarté syntaxique et son efficacité dans la prévention des ambiguïtés, sans alourdir le parseur.

5.2.2 Structures (classes, méthodes, program)

Dans le but d'étendre la fonctionnalité du projet et de gérer de manière plus flexible les déclarations, avec la prise en charge des valeurs initiales des attributs, des champs statiques et de l'immuabilité de certains attributs, nous

avons opté pour une approche qui évite de modifier les types de listes d'attributs au niveau des classes et méthodes existantes. Ainsi, plutôt que d'effectuer des changements structurels profonds à chaque ajout de fonctionnalité ou modification, nous avons choisi d'introduire de nouveaux attributs dans la définition des classes. Cela permet non seulement d'assurer une plus grande extensibilité du projet, mais également de préserver l'intégrité de l'environnement initial sans avoir à en modifier continuellement la structure. Pour ce faire, plusieurs nouveaux champs ont été ajoutés à la définition de classe, par exemple, chacun ayant un rôle spécifique. En choisissant cette approche, nous avons garanti une évolutivité du projet sans remettre en cause la structure existante. Par exemple, chaque ajout ou modification d'attributs (comme les valeurs d'initialisation ou les attributs statiques) peut se faire sans avoir à modifier les types ou les méthodes de la classe de manière globale. Cela offre une meilleure gestion des évolutions futures du projet, tout en maintenant la compatibilité avec les versions précédentes du système.

```
type class_def = {  
  class_name: string;  
  attributes: (string * typ) list;  
  methods: method_def list;  
  parent: string option;  
  is_attr_final: (string * bool) list;  
  static_attribut : (string * bool) list;  
  attr_init_vals: (string * expr option) list;  
}
```

FIGURE 11 – Structure de classes

5.2.3 Pourquoi pas visibility ?

Une des extensions les plus simples consistait à reprendre les concepts déjà abordés précédemment, comme les notions de static ou final. Alors pourquoi ne pas l'avoir implémentée ? Tout simplement à cause du manque de temps. Nous avons préféré concentrer nos efforts sur des concepts plus complexes et approfondir nos connaissances dans ces domaines, plutôt que de nous attarder sur une tâche plus simple. C'était peut-être une erreur, et nous en serons conscients au moment de la notation, mais c'est une erreur que nous assumons pleinement.

```
%token PUBLIC PRIVATE PROTECTED

type class_def = {
  class_name: string;
  attributes: (string * typ) list;
  methods: method_def list;
  parent: string option;
  is_attr_final: (string * bool) list;
  static_attribut : (string * bool) list;
  attr_init_vals: (string * expr option) list;
  attr_visibility: (string * int option) list;
}

visibility :
| PUBLIC      {Some(2)}
| PRIVATE     {Some(0)}
| PROTECTED   {Some(1)}
| (*void *)   {Some(2)}
;
```

FIGURE 12 – Codes pour implémenter visibility

6 Conclusion

En somme, ce projet nous a offert une occasion unique d'explorer des concepts avancés en gestion de types et de développer une approche modulaire et robuste pour l'interprétation de programmes. Grâce à l'ajout d'extensions comme l'automatisation des tests, l'amélioration des mécanismes de typage, et la génération de graphes pour visualiser les relations entre les composants, notre travail s'est non seulement simplifié mais aussi amélioré en termes de fiabilité. Ces outils ont vraiment optimisé notre workflow, réduisant le temps de développement tout en assurant que chaque fonctionnalité soit bien testée et fonctionne comme prévu. Ce projet a été une véritable aventure technique, nous permettant d'approfondir nos connaissances tout en garantissant la qualité du code grâce à une automatisation accrue. La collaboration en équipe, la gestion des extensions et l'attention portée aux besoins réels du projet ont été des éléments clés de notre succès. Nous sommes convaincus que les solutions que nous avons développées seront non seulement utiles pour ce projet, mais également extensibles à des initiatives futures. En gros, ce fut une expérience enrichissante et nous sommes fiers des résultats obtenus et impatients de voir comment ces solutions pourront être appliquées à d'autres projets.

7 Sources

- Menhir Reference Manual (version 20090505)
- Gitlab - François POITTIER
- Github - let-def/menhir
- Formal Grammar for Java - Jim Alves-Foss
- Discussions scientifiques sur StackOverflow, discuss.ocamm et Reddit
- Youtube and cs3110 - @MichaelRyanClarkson
- Youtube - @TJ DeVries
- Gitlab- vincenzopalazzo/languages-compilers-and-interpreters
- Cours L3 Langages, interprétation, compilation - université paris sa-clay
- Ruslans blog-let's build a simple interpreter
- Articles scientifiques - Wikipedia