GRENOBLE-INP ENSIMAG

INGÉNIERIE DES SYSTÈMES D'INFORMATION

2A

GRENOBLE-INP ENSIMAG

INFORMATION SYSTEMS ENGINEERING

2A

ÉCOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET DE MATHÉMATIQUES APPLIQUÉES DE GRENOBLE

PROJET GL

MANUEL UTILISATEUR

Réalisé par :

DIAKITE Alpha Ousmane
DREUILLE Nathan
FLISS YASSINE
KOSSI Yao David
OUDRHIRI IDRISSI Safwane

CLIENT :

FREIRE MARCO

Année académique : 2025–2026

Table des matières

1 Construction et utilisation du langage Deca

1.1 Structure générale d'un programme

Bloc principal (`main`)

```
{
    int x;
    int y = 1;
}
```

Main :
- Déclaration de x comme entier
- Affectation de 1 à y.

```
class C {
    int y;
}
```

Classe simple :
- La classe `A` ne possède pas de super-classe
- Elle définit un champ public.

1.2 Types, visibilités et variables

Visibilités supportées

- public
- protected

Types supportés

- int
- float
- boolean
- void
- types de classes

Déclarations des variables

```
{
    int x;
    float y, z;
    int a = 1, b = 2;
    boolean ok;
}
```

1.3 Expressions

Une expression est une valeur calculée à partir de constantes, variables, opérateurs et éventuellement d'appels de méthodes. Les expressions peuvent être utilisées :

— dans une affectation : `x = expr;`

— comme condition : `if (expr) {...}` (expr doit être un `boolean`)

— comme argument d'affichage : `println(expr);`

Littéraux

```
42
3.14
true
false
"texte"
null   // uniquement en contexte objet
```

Affectation

```
int a,b,x;
boolean ok = true;
x = 3;
a = b = 4;
```

Opérateurs arithmétiques

```
x + 1
a - b
a * b
a / b
a % b
```

Comparaisons

```
a < b
a <= b
a > b
a >= b
a == b       // a EQUALS b
a != b       // a NOT EQUALS b
```

Opérateurs logiques (booléens)

```
!flag        // NOT flag
a && b       // a AND b
a || b       // a OR b
```

Parenthèses

Les parenthèses permettent de forcer l'ordre de calcul.

```
(2 + 3) * 4
(a < b) && !flag
```

Expressions orientées objet

Accès à un champ et appel de méthode :

```
obj.x
obj.f(1, 2)
this.x
this.f()
```

1.4 Éléments lexicaux utiles (syntaxe pratique)

Cette section décrit des constructions reconnues dès l'analyse lexicale et qui sont importantes pour écrire des programmes Deca lisibles.

Inclusion de fichiers : `#include`

```
#include "Math.decah"

{
  Math m = new Math();
  println(m.sin(1.0));
}
```

Remarques :

— La directive doit être écrite sous la forme `#include "nomDuFichier"`.

— Le nom de fichier peut contenir des lettres, chiffres, `.`, `-`, `_`.

Chaînes de caractères : `STRING`

```
{
  println("Bonjour");
  println("Ligne avec un \n saut de ligne");
}
```

Affiche :

```
>    Bonjour
>    Ligne avec un
>    saut de ligne
```

Une chaîne de caractères est écrite entre guillemets `"..."`. Elle ne peut pas contenir de saut de ligne.

Chaînes multi-lignes : `MULTI_LINE_STRING`

Le langage supporte également des chaînes qui peuvent contenir des sauts de ligne. Elles sont particulièrement utiles pour fournir un bloc de texte (par exemple dans un `asm("...")`).

```
{
  println("Premiere ligne
           Deuxieme ligne
           Troisieme ligne
          ");
}
```

Espaces et retours à la ligne : `WS`

Les espaces, tabulations et retours à la ligne sont ignorés par l'analyse lexicale (en dehors des chaînes de caractères). Ils peuvent donc être utilisés librement pour indenter le code et améliorer la lisibilité.

```
{
  int x=3;
  int y = 4;

  println( x + y );
}
```

Commentaires sur une ligne : `// ...`

Tout texte après `//` jusqu'à la fin de la ligne est ignoré.

```
{
  int x = 3; // ceci est un commentaire
  println(x);
}
```

Commentaires en bloc : `/* ... */`

Les commentaires en bloc commencent par `/*` et se terminent par `*/`. Ils peuvent s'étendre sur plusieurs lignes.

```
/*
  Commentaire multi-lignes
  utile pour documenter un bloc de code
*/
{
  println("ok");
}
```

1.5 Entrées / Sorties

Lecture

```
{
    int n = readInt();
    println(n);
}
```

readInt() : Lit un entier sur l'entrée standard.

```
{
    float x = readFloat();
    println(x);
}
```

readFloat() : Lit un flottant sur l'entrée standard.

1.6 Affichages / impression en Deca

Affichage décimal : `print` **et** `println`

```
{
    int x = 3;
    print("x=");
    println(x);
}
```

print/println : Affichent des `int`, `float` ou `string`.
`println` ajoute un saut de ligne.

Affichage hexadécimal : `printx` **et** `printlnx`

Les instructions `printx` et `printlnx` permettent d'afficher des valeurs de type `float` en notation hexadécimale. Elles sont principalement destinées au débogage et à la vérification précise des valeurs flottantes.

```
{
    float a = 3;
    printx(a);
}
```

Affiche : `0x1.8p+1`

```
{
    float b = 1.0;
    printlnx(b);
}
```

Affiche : `0x1.0p+0`

La notation affichée suit la forme :

`0xMANTISSE p EXPOSANT`

où la mantisse est exprimée en base 16 et l'exposant en base 2.

1.7 Conversion explicite : `(type) (expr)`

Cast : Conversion explicite entre types numériques (selon les règles de la spécification). On peut convertir un objet de type int en type float et inversement.

```
{
    int a = 3;
    float x = (float) (a);
    int b = (int) (3.7);
}
```

NB : Les parenthèses sont **obligatoires** pour éviter des erreurs syntaxique.

1.8 Structures de contrôle

Conditions

```
{
    if ( x < 0 ) {
        print("negatif");
    } else if ( x == 0 ) {
        print("nul");
    } else {
        print("positif");
    }
}
```

`NB` : `else if` peut être utilisé plusieurs fois dans un seul `if`.

Boucles

```
while (x > 0) {
    x = x - 1;
}
```

2 Création d'objets : `new`

```
class Point {
    int x;
    int y;
}

{
    Point p = new Point();
    p.x = 2;
    p.y = 3;
    println(p.x);
}
```

new : Instancie un objet. L'accès aux champs se fait avec `obj.champ`.

3 Héritage en Deca

La notion d'**héritage des classes** est permise en Déca avec le mot clé `extends`. Ce qui est important lors d'une redéfinition de méthodes ou la factorisation d'un code. En fin, le fait qu'une classe fille A hérite d'une classe mère (ou encore super classe) B se traduit de la manière suivante : `class A extends B`.

```
class B {}

class A extends B {
    int x;

    int getX() {
        return x;
    }
}
```

4 Méthodes en Deca

La signature d'une méthode en Deca implique :

1. La visibilité
2. Le type de retour
3. Le nom de la méthode
4. Le(s) paramètre(s) si elle en possède et son/leur type.

4.1 Méthodes simples

```
protected int methodX(int x
   ) {
    return x + 1;
}
```

- Visibilité : protected
- Type de retour : int
- nom de méthode : methodX
- Paramètre : x de type int

```
void methodY(int x) {
    print(x);
}
```

- Visibilité : public
- Type de retour : void
- nom de méthode : methodY
- Paramètre : x de type int

4.2 Méthodes avec corps assembleur : `asm`

Le mot-clé `asm` permet de définir une méthode dont le corps est écrit directement en assembleur IMA. Ce type de méthode est principalement utilisé pour implémenter des fonctionnalités de bas niveau ou des bibliothèques standard.

La syntaxe consiste à remplacer le corps classique de la méthode par un bloc `asm("...")` contenant les instructions assembleur.

```
class A {
    int foo()
        asm("LOAD #0, R0
             ADD #1, R0
        ");
}
```

Remarque : les méthodes utilisant `asm` sont généralement destinées aux bibliothèques et ne sont pas recommandées pour un usage courant dans les programmes utilisateurs.

4.3 Appel de méthode

```
class A {
    int f(int a) {
        return a + 1;
    }
}

{
    A obj = new A();
    println(obj.f(3));
}
```

Appel : `obj.methode(params)`. Le type retourné dépend de la signature.

4.4 Redéfinition de méthode (override)

Une classe fille peut redéfinir une méthode héritée de sa classe mère en conservant **la même signature**. Pour se faire il suffit de caster l'objet fille `B` en objet mère `A`. Ce qui se fait selon cette syntaxe : `((A)(this)).<methodeARedefinir>`. Cette manière de faire est une représentation du mot clé `super` en Java.

L'exemple suivant illustre bien le critère de redéfinition

```
class Personne {
int num;

void init (int nb) {
  this.num = nb;
}
void afficherNb () {
  print(this.num) ;
}
}

class Etudiant extends Personne {
  int age ;

void init(int n) {
  ((Personne)(this)).init(n) ;  // cast
  this.age = 23;
}
void afficherAge () {
  println(age) ;
}
}
{
Etudiant e = new Etudiant () ;
e.init (1) ;
e . afficherNb () ; // Etudiant herite de afficheNb ()
print (" - toto : ") ;
e . afficherAge () ;
}
```

Retour attendu : `1 - toto : 23`

NB : La méthode appelée dépend du type dynamique de l'objet.

4.5 Retour de méthode : `return`

```
class A {
    int f(int a) {
        return a + 1;
    }
}
```

return : Termine la méthode et renvoie une valeur compatible avec le type de retour.

5 Test de type : `instanceof`

```
class A {}
class B extends A {}

{
    A obj = new B();
    if (obj instanceof B) {
        println("obj est une instance de B");
    }
}
```

instanceof : Teste dynamiquement si un objet appartient à une classe (ou sous-classe).

6 Mot-clé `this`

```
class A {
    int x;

    void setX(int x) {
        this.x = x;
    }
}
```

6.1 Utilisation implicite de `this` dans les classes

Dans notre compilateur, l'accès aux attributs et aux méthodes d'une classe peut, dans certains cas, se faire sans utiliser explicitement le mot-clé `this`. Cependant, cette omission n'est valide

que lorsqu'il n'existe aucune ambiguïté de nom.

Accès aux attributs sans ambiguïté Considérons l'exemple suivant :

```
class A {
    int x;
    void setX(int c) {
        x = c;
    }
}

{
    A a = new A();
    a.setX(5);
    println(a.x);
}
```

Dans la méthode `setX`, l'affectation `x = c` fonctionne sans préfixer `x` par `this`, car :

- `x` est un attribut de la classe,
- `c` est un paramètre de la méthode,
- il n'existe aucune ambiguïté entre les deux identificateurs.

Le compilateur résout donc automatiquement `x` comme étant `this.x`.

Cas d'ambiguïté avec les paramètres En revanche, lorsque le paramètre de la méthode porte le même nom qu'un attribut, l'utilisation explicite de `this` devient obligatoire.

```
    class A {
    int x;
    void setX(int x) {
        this.x = x;
    }
}
```

Dans ce cas, l'identificateur `x` fait référence par défaut au paramètre local. Sans l'utilisation de `this.x`, l'attribut de la classe serait masqué et la méthode ne fonctionnerait pas comme attendu.

Appel de méthodes au sein d'une même classe De manière similaire, lorsqu'une méthode appelle une autre méthode de la même classe, l'utilisation explicite de `this` est nécessaire.

```
class A {
    int p = 2;

    void init(int n) {
        p = n;
    }

    void reInit(int m) {
        this.init(m);
    }

    void affiche() {
        println(p);
    }
}

{
    A a = new A();
    a.init(3);
    a.reInit(5);
    a.affiche();
}
```

7 Messages d'erreur

Le compilateur peut produire différents types d'erreurs au cours de la compilation ou de l'exécution.

7.1 Erreurs lexicales

Ces erreurs surviennent lors de l'analyse lexicale, par exemple :

- **caractère non reconnu**

```
    {
        "chaine pas finie
    }
    Erreur :<filename>:2:4: token recognition error at: '"
       chaine pas finie\n}\n\n'
```

- **chaîne de caractères mal formée**

```
{
print("hello);
}
Erreur :<filename>:2:6: token recognition error at: '"
   hello);'
```

7.2 Erreurs syntaxiques

Les erreurs syntaxiques indiquent :

- **la ligne**
- **la colonne**
- **le symbole fautif**

```
{
    int x;
    iff (x = 0) {
        x = 2;
    } else {
        x = 3;
    }
}
Erreur :
<filename>:3:16: no viable alternative at input '{'
```

Certaines erreurs de syntaxe peuvent être signalées sur un symbole situé après la cause réelle de l'erreur. Cela est dû au fonctionnement de l'analyse syntaxique, qui détecte l'erreur au moment où aucune règle grammaticale valide ne peut être appliquée . Dans cet exemple, l'erreur provient de l'utilisation du mot-clé invalide `iff`. Le message est toutefois déclenché – lors de la lecture

du symbole `{` car il s'attendait à un `;` supposant que c'est une appelle de fonction `iff(args)`– lorsque le parseur ne trouve plus de règle valide.

7.3 Erreurs contextuelles

Ces erreurs sont détectées lors des vérifications contextuelles (typage, déclarations, portées).

Message d'erreur	**Règle violée**	**Configuration provoquant l'erreur**
Erreurs de typage arithmétique et conversions		
Incompatible types for arithmetic operation : [type1] and [type2]	Les opérateurs arithmétiques s'appliquent uniquement aux types numériques	Opération arithmétique entre types incompatibles (ex. `boolean + int`)
Incompatible types for modulo operation : [type1] and [type2]	L'opérateur modulo est réservé aux entiers	Utilisation de % avec des types non entiers
Modulo requires integer operands	Les opérandes de % doivent être de type int	Application du modulo à un float
Operator MINUS cannot be applied on type [type]	L'opérateur unaire - s'applique uniquement aux types numériques	moins unaire appliquée à un booléen
Operator CONV_FLOAT cannot be applied on type [type]	La conversion implicite en float est autorisée uniquement depuis int	Tentative de conversion depuis un type non numérique
Erreurs sur les expressions booléennes et conditions		
Condition expression must be of boolean type	Les conditions de contrôle doivent être booléennes	Condition de if ou while non booléenne
Boolean operations require boolean operands	Les opérateurs logiques nécessitent des booléens	Utilisation de `&&` ou `||` avec des types non booléens
Operator NOT cannot be applied on type [type]	L'opérateur ! s'applique uniquement aux booléens	Négation appliquée à un type non booléen
Erreurs sur les comparaisons		
Comparison operators require int or float operands	Les comparaisons sont définies sur les types numériques	Comparaison impliquant des booléens
Invalid types for equality comparison : [type1] and [type2]	Les comparaisons d'égalité nécessitent des types compatibles	Comparaison entre types incompatibles
Erreurs d'affectation et de conversion		
Type mismatch : cannot assign/-cast [exprType] to [targetType]	Compatibilité des types lors de l'affectation	Affectation d'un float à un int

Message d'erreur	**Règle violée**	**Configuration provoquant l'erreur**
Type mismatch : cannot cast [exprType] to [targetType]	Conversions explicites autorisées uniquement	Cast invalide entre types incompatibles
Identifier [name] is not assignable	Une affectation doit cibler une LValue	Affectation sur une expression non assignable
Erreurs sur les identificateurs et déclarations		
Identifier [name] is unknown	Tout identificateur doit être déclaré avant usage	Utilisation d'une variable non déclarée
Type [name] is unknown	Les types doivent être définis	Utilisation d'un type inexistant
Variable [name] is already defined in this scope	Un identificateur ne peut être redéfini dans le même environnement	Double déclaration dans un même bloc
Parameter [name] already defined	Les paramètres doivent avoir des noms distincts	Deux paramètres portant le même nom
A variable cannot have type void	Le type void est interdit pour les variables	Déclaration d'une variable de type void
A parameter cannot have type void	Les paramètres ne peuvent pas être de type void	Paramètre déclaré avec le type void
Erreurs liées aux entrées / sorties		
Only strings, ints and floats can be printed	Les fonctions d'affichage acceptent uniquement certains types	Tentative d'affichage d'une expression non autorisée
Erreurs d'exécution du code assembleur		
Erreur d'exécution (division par zéro, débordement, accès mémoire)	Contraintes non vérifiables statiquement	Dépend des valeurs manipulées à l'exécution

7.4 Erreurs à l'exécution

Ces erreurs sont détectées lors de l'exécution du code assembleur généré :

- **Débordement arithmétique**

```
{
    int a = 2147483647;
    a = a + 1;
    print(a);
}
Resultat :
-2147483648
```

On obtient un débordement arithmétique silencieux. En effet le programme compile mais ne fait pas le calcul attendu ($2147483647 = 2^{31} - 1$ est la plus grande valeur possible pour un int donc quand on ajoute 1 à a la valeur reboucle et on obtient -2^{31}).

- **Erreur d'entrée/sortie**

```
{
int a = readInt();
print(a);

}
Saisie utilisateur : Hello
Resultat :
Erreur: D'entree incompatible avec la variable
```

- **Dépassement de pile**

```
class A{
    int f(){
        return this.f();
    }
}
{
A a = new A();
a.f();
}
Resultat :
Erreur:   ** IMA ** ERREUR ** Ligne 59 :
    BSR : Debordement de la pile
```

- **Erreurs internes du compilateur**

  Dans certains cas, une erreur de typage (par exemple l'utilisation d'un identifiant inconnu comme `prnt`) peut provoquer une erreur interne du compilateur au lieu d'un message contextuel approprié.

  Ce comportement correspond à une limitation actuelle de l'implémentation de la partie orientée objet.

8 Utilisation du compilateur

Le compilateur `decac` compile un ou plusieurs fichiers `.deca` en un programme assembleur pour la machine abstraite IMA (fichier `.ass`).

8.1 Syntaxe générale

```
decac [[-p | -v] [-n] [-r X] [-d]* [-P] <fichier.deca>...] | [-b]
```

— `<fichier.deca>...` : un ou plusieurs (valable pour -P) fichier(s) source Deca.

— Les options `-p` et `-v` sont **mutuellement exclusives**.

8.2 Configuration de la commande `decac`

Afin de faciliter l'utilisation du compilateur `decac` depuis n'importe quel répertoire, un script shell nommé `runDecac` est fourni à la racine du projet (répertoire `gl54`).

Objectif

Ce script permet d'ajouter automatiquement le répertoire contenant le binaire `decac` au `PATH` de l'utilisateur. Une fois configuré, le compilateur peut être invoqué depuis n'importe quel dossier sans avoir à spécifier son chemin absolu.

Procédure d'installation

1. Se placer à la racine du projet :

   ```
   cd gl54
   ```

2. Rendre le script exécutable (la première fois) :

   ```
   chmod +x runDecac
   ```

3. Lancer le script :

   ```
   ./runDecac
   ```

4. Appliquer immédiatement la modification du `PATH` selon le message affiché sur terminal d'exécution (de forme : `source /...`).

Fonctionnement interne

Le script :

— détecte automatiquement le shell utilisé (`bash`, `zsh`, ou autre),

— sélectionne le fichier de configuration correspondant,

— ajoute le répertoire `src/main/bin` au `PATH` si et seulement s'il n'est pas déjà présent.

Pour éviter toute duplication, le script vérifie la présence exacte de la ligne suivante dans le fichier de configuration :

```
export PATH="$PATH:/chemin/absolu/vers/src/main/bin"
```

Cette vérification empêche les faux positifs qui pourraient survenir si le chemin apparaissait ailleurs dans le fichier (commentaires, anciennes configurations, etc.).

Résultat

Une fois la configuration terminée, l'utilisateur peut compiler un fichier `.deca` depuis n'importe quel répertoire :

```
decac mon_fichier.deca
```

Remarque importante

Le script `runDecac` doit impérativement être exécuté depuis la racine du projet (`gl54`). Un lancement depuis un autre répertoire empêcherait la résolution correcte du chemin vers le binaire `decac`.

8.3 Outil complémentaire : `decac-energy`

Dans le cadre du projet GL, un script expérimental nommé `decac-energy` est fourni afin d'estimer l'impact d'une compilation et de l'exécution d'un programme Deca.

Objectif

L'objectif de cet outil est de fournir des indicateurs simples (performance, nombre d'instructions générées, temps d'exécution, consommation mémoire) permettant de raisonner sur le coût d'exécution d'un programme, et indirectement sur son impact énergétique.

Principe de fonctionnement

Le script :

— compile un fichier `.deca` à l'aide de `decac`,

— estime le nombre d'instructions IMA générées à partir du fichier assembleur,

— mesure le temps d'exécution et la consommation mémoire via l'outil `time`.

Les valeurs obtenues constituent des indicateurs approximatifs et ne représentent pas une mesure directe de consommation énergétique.

Utilisation

Le script s'utilise sur un fichier `.deca` de la manière suivante :

1. Exécuter le script `runDecac` et recharger la configuration du shell depuis la racine du projet comme décrit dans la section Procédure d'installation si ce n'est pas encore fait
2. Lancer `decac-energy` sur un programme `.deca` comme tel :
   `decac-energy mon_programme.deca`

Remarque

Cet outil est fourni à titre expérimental et pédagogique. Il ne modifie en aucun cas le comportement du compilateur `decac`.

8.4 Options disponibles

— `-b` (banner) : affiche la bannière du compilateur. Dans ce mode, aucun fichier n'est compilé.

— `-p` (parse) : mode *parse*. Affiche le programme après décompilation (sortie texte) et n'effectue pas la génération de code.

— `-v` (verification) : mode *verify*. Effectue les vérifications contextuelles et affiche les informations associées (diagnostics / traces), sans lancer la génération de code.

— `-n` (no check) : désactive certaines vérifications à l'exécution lors de la génération du code (par exemple certaines protections ou tests runtime).

— `-r` X (registers) : fixe le nombre de registres utilisables à X, avec $4 \leq X \leq 16$. **Une valeur en dehors de cet intervalle provoque une erreur.**

— `-d` (debug) : augmente le niveau de traces (option répétable). Plusieurs occurrences augmentent la verbosité : `-d`, `-dd`, `-ddd`, etc.

— `-P` (parallel) : lance la compilation des fichiers en parallèle (pour accélérer la compilation) s'il y a plusieurs fichiers sources.

8.5 Exemples d'utilisation

Compilation standard :

```
decac prog.deca
```

Compilation de plusieurs fichiers :

```
decac prog1.deca prog2.deca
```

Affichage du programme décompilé (sans génération de code) :

```
decac -p prog.deca
```

Vérifications contextuelles (sans génération de code) :

```
decac -v prog.deca
```

Compilation en désactivant certaines vérifications à l'exécution :

```
decac -n prog.deca
```

Compilation en limitant le nombre de registres à 8 :

```
decac -r 8 prog.deca
```

Augmentation du niveau de traces :

```
decac -d prog.deca
decac -dd prog.deca
```

8.6 Exécution des programmes compilés

Après la compilation d'un programme `.deca` à l'aide du compilateur `decac`, un fichier assembleur `.ass` est généré. Ce fichier contient le code IMA correspondant au programme source et constitue l'entrée du simulateur IMA pour l'exécution.

Commande d'exécution

L'exécution d'un programme compilé se fait à l'aide de la commande suivante :

```
ima nom-fichier.ass
```

Pour avoir les statistiques de l'exécution (nombre d'instructions et temps d'exécution) :

```
ima -s nom-fichier.ass
```

où `nom-fichier.ass` est le fichier assembleur produit par `decac`. Le simulateur `ima` interprète ce fichier et exécute les instructions IMA qu'il contient.

Remarque sur l'extension `.ass`

Il est nécessaire de fournir explicitement l'extension `.ass` lors de l'appel à `ima`. En effet, le simulateur IMA ne possède pas de mécanisme de résolution automatique des extensions, contrairement à certains environnements comme Java où la commande `java` se base sur le nom de la classe.

Par exemple, la commande suivante est incorrecte et ne fonctionnera pas :

```
ima nom-fichier
```

8.7 Compilation et exécution simplifiées

Afin d'améliorer l'expérience utilisateur, nous avons ajouté un script Bash permettant de simplifier la compilation et l'exécution des programmes Deca.

Script `exec-decac` Le script `exec-decac`, situé dans `src/main/bin/`, permet de compiler et d'exécuter un programme Deca en une seule commande. Il automatise les étapes suivantes :

- Compilation du fichier `.deca` avec `decac`
- Exécution du fichier assembleur généré avec l'interpréteur `ima`

Son utilisation est la suivante :

```
exec-decac nom_programme
```

Test de la fonctionnalité Pour tester cette amélioration, il suffit de :

1. Exécuter le script `runDecac` et recharger la configuration du shell depuis la racine du projet comme décrit dans la section Procédure d'installation si ce n'est pas encore fait
2. Lancer `exec-decac` sur un fichier `.deca`

8.8 Erreurs de ligne de commande

Le compilateur peut refuser l'exécution dans les cas suivants :

- option inconnue (toute option commençant par `-` non supportée) ;
- options `-p` et `-v` utilisées simultanément ;
- utilisation de `-r` sans argument ;
- argument de `-r` non entier ou hors de l'intervalle `[4,16]` ;
- aucun fichier fourni (hors cas `-b`) : affichage de l'aide.

9 Limitations connues

9.1 Limitation concernant les méthodes `asm`

Notre compilateur permet d'intégrer du code assembleur IMA directement dans une méthode Deca via la construction `asm("...")`. Cependant, cette fonctionnalité présente une limitation concernant l'utilisation de chaînes de caractères avec l'instruction d'affichage `WSTR`.

En assembleur IMA, l'instruction `WSTR` attend une chaîne de caractères écrite sous la forme suivante :

```
WSTR "message"
```

Or, dans le langage Deca, les chaînes de caractères sont également délimitées par des guillemets (`"`). Ainsi, écrire directement l'instruction suivante dans un bloc `asm` :

```
WSTR "message"
```

entraîne une erreur de syntaxe lors de l'analyse du programme Deca, car les guillemets internes ne sont pas échappés.

Une tentative de correction consiste à échapper les guillemets :

```
WSTR \"message\"
```

Cette écriture permet au compilateur Deca de réussir l'analyse syntaxique. Cependant, lors de la génération du code assembleur IMA, les caractères d'échappement (\) sont conservés, ce qui produit le code suivant :

```
WSTR \"message\"
```

Or, l'assembleur IMA ne reconnaît pas le caractère \, ce qui provoque une erreur à l'exécution.

Cette limitation ne peut pas être corrigée à l'étape de génération de code (étape C), car l'information sur l'échappement des chaînes est déjà figée lors des phases précédentes d'analyse lexicale et syntaxique. Une correction nécessiterait une modification du traitement des chaînes de caractères dans la grammaire ou dans l'analyse lexicale.

En conséquence, l'utilisation de l'instruction `WSTR` avec une chaîne de caractères littérale n'est pas supportée dans les blocs `asm` de notre compilateur.

9.2 Appel de méthodes au sein d'une même classe

La version finale de notre compilateur n'implémente pas l'appel de méthodes –même issue de la même classe– sans le mot clé `this`. IL faut obligatoirement précéder l'appel à la méthode d'un `this`. Ainsi le programme suivant déclenchera une erreur à l'exécution et demandera de fournir un objet sur lequel appeler la méthode à la ligne 9.

```
1 class A {
2     int p = 2;
3
4     void init(int n) {
5         p = n;
6     }
7
8     void reInit(int m) {
9         init(m);
10     }
11
12     void affiche() {
13         println(p);
14     }
15 }
16
17 {
18     A a = new A();
19     a.init(3);
20     a.reInit(5);
21     a.affiche();
22 }
```

10 Utilisation de l'extensions trigo

10.1 Activation de l'extension

L'extension `TRIGO` est intégrée à la bibliothèque standard du compilateur. Aucune option supplémentaire n'est nécessaire pour l'activer.

La compilation d'un programme utilisant les fonctions trigonométriques se fait de la même manière qu'un programme Deca standard :

```
decac programme.deca
```

10.2 Utilisation dans un programme Deca

Les fonctions trigonométriques sont fournies par la classe `Math`. Cette classe est rendue disponible via l'inclusion du fichier correspondant en début de programme :

```
#include "Math.decah"
```

Les attributs constantes disponibles :

— `PI` : valeur de PI

Les fonctions disponibles sont les suivantes :

— `sin(float)` : sinus

— `cos(float)` : cosinus

— `atan(float)` : arc tangente

— `asin(float)` : arc sinus

— `ulp(float)` : précision du flottant

Toutes ces fonctions prennent un paramètre de type `float` et renvoient une valeur de type `float`.

Pour utiliser une fonction, il est nécessaire de créer une instance de la classe `Math`, puis d'appeler la méthode souhaitée.

```
{
  Math m = new Math();
  float y = m.cos(m.PI/2);
  println(x);
}
```

10.3 Options de compilation

Aucune option spécifique de la commande `decac` n'est requise pour utiliser cette extension.

11 Limitations des extensions

Cette section analyse les marges d'erreur obtenues lors de la validation de l'extension `TRIGO` (`sin`, `cos`, `atan`, `asin`). Elle démontre que les erreurs observées sont les conséquences inévitables de l'architecture 32 bits et de la segmentation des intervalles de calcul.

11.1 La Barrière de Précision : Le Mur des 23 bits

La contrainte fondamentale de notre implémentation est l'utilisation exclusive du format `float` IEEE 754. Ce format alloue **23 bits** à la mantisse, ce qui fixe la précision relative théorique à :

$$\epsilon = 2^{-23} \approx 1,19 \times 10^{-7}$$

Toute opération arithmétique élémentaire introduit une erreur d'arrondi potentielle de 0,5 ULP (*Unit in the Last Place*). Dans un polynôme de degré élevé, ces erreurs s'accumulent. En conséquence, même avec un algorithme mathématiquement optimal, il est impossible de garantir 23

bits de précision sur le résultat final. Nous visons une précision effective de **21 à 22 bits**, ce qui correspond à une perte de 1 à 2 bits due au "bruit" de calcul.

11.2 Analyse Spécifique des Fonctions Inverses (`arctan`, `asin`)

Nos implémentations des fonctions inverses reposent sur une découpe du domaine de définition pour assurer la convergence des séries. Cette méthode introduit structurellement des discontinuités de précision aux points de raccordement.

L'Arc-Tangente (`arctan`) : Le Pivot de $\pi/4$

Pour la fonction `arctan`, nous changeons de formule d'approximation autour de la valeur $|x| = 1$ (soit un angle de $\pm\pi/4$).

- **Phénomène observé :** Une légère dégradation locale de la précision est visible exactement à ce point de bascule.
- **Cause technique :** Le raccordement entre les deux polynômes n'est jamais parfait en arithmétique flottante. Les erreurs d'arrondi des deux méthodes s'additionnent de manière imprévisible à la frontière, créant un pic d'erreur localisé.

Arcsinus (`asin`) : Le Pivot de 0.5

Pour la fonction `asin`, la segmentation se fait à $|x| = 0.5$. Comme illustré par nos mesures, l'erreur maximale atteint **3.0 ULP** spécifiquement à ce point de transition.

- **Traduction en bits :** Une erreur de 3 ULP signifie que l'incertitude porte sur les $\log_2(3) \approx 1.58$ derniers bits.
- **Conclusion :** Sur les 23 bits disponibles, nous en perdons environ 1.6 à cause de la discontinuité algorithmique. La précision réelle garantie est donc de **21.4 bits**, ce qui reste une performance très satisfaisante pour une bibliothèque sans type `double`.

11.3 Le Problème de la Réduction d'Argument (`sin`, `cos`)

Pour les fonctions périodiques, la limitation majeure provient du stockage des constantes. Pour calculer $\sin(x)$ sur un grand angle, nous devons effectuer la réduction $r = x - k\pi$. Or, la spécification nous contraint à stocker π sur 32 bits.

- **Catastrophic Cancellation :** Lorsque l'entrée x est grande (ex : $x > 10^4$), sa partie entière occupe la majorité des bits de la mantisse. Lors de la soustraction, les bits significatifs s'annulent et le résultat ne contient plus que les bits de poids faible, qui sont pollués par l'erreur d'approximation de π.
- **Conséquence :** La précision s'effondre linéairement avec la grandeur de x. C'est une limitation physique de la machine *ima* : sans type 64 bits pour stocker une valeur précise de π, il est impossible de maintenir la précision trigonométrique pour les grands angles.

11.4 L'Observabilité Tronquée : Le Masque WFLOAT

Enfin, la validation fine de nos résultats se heurte à l'instruction d'affichage `WFLOAT`, qui tronque la sortie à **5 décimales**.

— Une erreur de 2 ou 3 ULP (notre marge critique) affecte généralement la 7ème décimale. Elle est donc **invisible** via l'affichage standard.

Note aux évaluateurs : Pour une analyse plus approfondie incluant les courbes d'erreur détaillées (graphes) et les justifications mathématiques complètes des polynômes utilisés, nous vous invitons à consulter la section *Optimisation et Justification des choix d'approximation* dans la documentation de l'extension.