MANUEL UTILISATEUR

EQUIPE 20

GROUPE 4

Table des matières

SECTION 1

Le compilateur **decac**

1.1 Utiliser decac

Deca est un langage dérivé du langage Java. C'est un langage qui permet de rester proche de la machine tout en utilisant des fonctionnalités de langage haut niveau. Le compilateur Decac permet de compiler ce langage vers le langage assembleur de la machine astraite IMA.
La compilation s'effectue sur des fichier .deca. La structure de fichier deca reste simple.

```
1   // Description:
2   //     Programme minimaliste qui affiche a l'ecran Hello, world.
3   //
4   // Resultats:
5   // Hello, world
6   //
7   // Historique:
8   //     cree le 01/01/2022
9
10  {
11          println("Hello, world");
12  }
```

Le compilateur decac permet de créer un fichier .ass en langage assembleur à partir d'un fichier .deca en écrivant la commande `decac file.deca` dans un terminal (file étant le nom du fichier .deca).

Il est possible de lancer l'exécution de decac sur plusieurs fichiers en une seule commande. Pour cela, il suffit d'écrire le nom de chaque fichier à la suite l'un de l'autre, sous la forme :

```
decac file1.deca file2.deca ... fileN.deca
```

ou plus simplement :

```
decac *.deca
```

pour lancer la compilation de tous les fichiers deca présents dans le répertoire courant.

1.2 Les paramètres

Plusieurs fonctionnalités sont disponibles avec la compilateur decac en tant qu'option de compilation.

- **-b** (banner) : affiche une bannière indiquant le nom de l'équipe à l'origine du projet
- **-p** (parse) : arrête decac après l'étape de construction de l'arbre, et affiche la décompilation de ce dernier (i.e. s'il n'y a qu'un fichier source à compiler, la sortie doit être un programme deca syntaxiquement correct)
- **-v** (verification) : arrête decac après l'étape de vérifications (ne produit aucune sortie en l'absence d'erreur)
- **-n** (no check) : supprime les tests de verification de debordement et de déréférencement null
- **-r X** (registers) : limite les registres banalisés disponibles à R0 ... RX-1, avec 4 <= X <= 16
- **-d** (debug) : active les traces de debug. Répéter l'option plusieurs fois pour avoir plus de traces.
- **-P** (parallel) : s'il y a plusieurs fichiers sources, lance la compilation des fichiers en parallèle (pour accélérer la compilation)"
- **-t** (tree) : arrête decac après l'étape de construction de l'arbre, et affiche ce dernier
- **-x** (enrichedTree) : arrête decac après l'étape de vérification et affiche l'arbre enrichi

1.3 Les étapes de la compilation

La compilation d'un programme deca se fait en 3 étapes qui seront détaillées dans la suite de ce manuel :

— Une étape d'analyse lexicale et syntaxique
— Une étape de vérification contextuelle
— Une étape de génération de code

Chacune de ces étape est susceptible de renvoyer des erreurs qui leur sont propres.

SECTION 2

Analyse lexicale et syntaxique

2.1 Analyse lexicale

L'implémentation de l'analyse lexicale est effectuée dans son intégralité, dans le fichier **DecaLexer.g4**.
Un caractère ou une séquence de caractères n'appartenant pas à la lexicographie de Deca renverra l'erreur :
`<file>:<lgn>:<col> : Caractère(s) non reconnu(s) par notre lexer`
Ensuite, on liste aussi trois autres messages d'erreurs possibles.

- Pour les littéraux entiers, une erreur de compilation est levée si un littéral entier n'est pas codable comme un entier signé positif sur 32 bits.
  `<file>:<lgn>:<col> : Le littéral <litt> est trop grand`
- Pour les littéraux flottants, une erreur de compilation est levée si un littéral est trop grand et que l'arrondi se fait vers l'infini, ou bien qu'un littéral non nul est trop petit et que l'arrondi se fait vers zéro.
  `<file>:<lgn>:<col> : Le littéral <litt> est trop grand et l'arrondi se fait vers l'infini`
  ou
  `<file>:<lgn>:<col> : Le littéral <litt> est trop petit et l'arrondi se fait vers 0`

On peut tester l'utilisation de ce lexer :

- en ligne de commande avec la commande `test_lex`
- avec l'exécution de `/src/test/script/basic-lex.sh`

2.2 Analyse syntaxique

L'implémentation de l'analyse syntaxique est effectuée pour la partie sans objet ici, dans le fichier **DecaParser.g4**.
L'exécution se fait en entrant `test_synt` ou `tests_synt`.

- `test_synt` permet de lancer l'analyse syntaxique dans le terminal et d'entrer manuellement le code deca à tester.
- `tests_synt` permet de lancer l'analyse syntaxique directement sur des fichiers tests .deca contenant le code à tester.

Une erreur dans la syntaxe renvoie les erreurs à l'exécution :

- Lorsqu'une expression n'est pas définie,
  `<unknown>:[location]: no viable alternative at input '...'`
- Lorsqu'un token est manquant,
  `<unknown>:[location]: missing '...' at '...'`

Lorsque l'exécution se déroule convenablement, `test_synt` renvoit alors un arbre sans décoration, représentant les déclarations des variables et chaque instanciation.

SECTION 3

Vérifications contextuelles

3.1 Partie contextuelle du projet

L'implémentation de cette partie est entièrement implémentée, principalement dans les méthodes verify* des fichiers **java/fr/ensimag/deca/tree/*.java**. Cette partie permet, après avoir reconnu les caractères à l'étape précédente, de vérifier la syntaxe contextuelle de chaque ligne du fichier avant de générer une décoration pour l'arbre obtenu auparavant. Ces décorations sont les définitions et les types des variables ou instances du fichier analysé.
Voici une liste des erreurs contextuelles pouvant être rencontrées lors de la compilation d'un fichier .deca.

3.2 Erreurs : déclarations dans une classe

Pour les déclarations de classes :

- `<file>:<lgn>:<col> : undefined super class` : la classe dont hérite la classe courante n'est pas définie.
- `<file>:<lgn>:<col> : not class extension` : la classe courante n'hérite pas d'une classe.

Pour les déclarations de champs :

- `<file>:<lgn>:<col> : <field> isn't a field` : ce nom a déjà été attribué dans une classe supérieure, mais ce n'est pas un champ.
- `<file>:<lgn>:<col> : field void` : un champ déclaré ne peut pas être de type *void*.

— `<file>:<lgn>:<col> : can't defined field identifier several times in a class` : le champ a déjà été déclaré dans la classe.
— `<file>:<lgn>:<col> : Incompatible extension of field <field>` : le champ hérité n'a pas le même type.

Pour les déclarations de méthodes :

— `<file>:<lgn>:<col> : <method> isn't a method` : ce nom a déjà été attribué dans une classe supérieure, mais ce n'est pas une méthode.
— `<file>:<lgn>:<col> : <method> must have same signature` : une méhode doit avoir la même signature que celle dont elle hérite.
— `<file>:<lgn>:<col> : <method> must have same type` : une méhode doit avoir le même type que celle dont elle hérite.
— `<file>:<lgn>:<col> : can't defined method identifier several times in a class` : la méthode a déjà été déclarée dans la classe.
— `<file>:<lgn>:<col> : can't have null type` : la classe mère ne peut pas être de type *null*.

Pour les déclarations de paramètres :

— `<file>:<lgn>:<col> : parameter void` : un paramètre déclaré ne peut pas être de type *void*.
— `<file>:<lgn>:<col> : can't defined parameter identifier several times in a class` : le paramètre a déjà été déclaré dans la méthode.

Pour l'utilisation de **this** :

— `<file>:<lgn>:<col> : Current class must be not null` : on ne peut pas utiliser **this** hors d'une classe.

3.3 Erreurs : déclarations dans le Main

Pour la déclaration d'une variable :

— `<file>:<lgn>:<col> : type void` : un identifier déclaré ne peut pas être de type *void*.
— `<file>:<lgn>:<col> : L'identificateur ne peut etre defini plus qu'une fois` : l'identifier a déjà été déclaré dans le main.

Pour la lecture d'un identifier :

— `<file>:<lgn>:<col> : <identifier>: identifier not defined` : l'identifier n'a pas été déclaré au préalable.
— `<file>:<lgn>:<col> : Identifier-type error` : le type d'identifier n'est pas reconnu.

Pour l'utilisation de **new** :

— `<file>:<lgn>:<col> : class <class> not defined` : le constructeur appelé n'est pas défini, car sa classe n'est pas définie.
— `<file>:<lgn>:<col> : the type of the class must be a class` : le constructeur appelé n'est pas un constructeur de classe.

3.4 Erreurs : affectations

Pour l'utilisation d'un assign "=" :

- `<file>:<lgn>:<col> : Can't assign a null value` : on ne peut affecter à une variable la valeur *null*.
- `<file>:<lgn>:<col> : Assignment error` : le type de l'expression du membre de droite n'est pas compatible avec celui de la variable.

3.5 Erreurs : opérations

Pour l'utilisation d'expressions arithmétiques, booléennes ou de comparateurs :

- `<file>:<lgn>:<col> : NotIntAndNotFloatType` : la ou les expressions attendues auraient dues être de type *Int* ou *Float*.
- `<file>:<lgn>:<col> : NotBooleanType` : la ou les expressions attendues auraient dues être de type *Boolean*.
- `<file>:<lgn>:<col> : NotIntType` : la ou les expressions attendues auraient dues être de type *Int*.

Pour l'utilisation de **instanceof** :

- `<file>:<lgn>:<col> : Incorrect types` : l'opérateur instanceof n'est pas défini pour ces types, il est défini pour *null* instanceof <*type_quelconque*> ou pour deux classes.

Pour l'appel d'un champ "_._" :

- `<file>:<lgn>:<col> : <class> isn't a class` : le champ sélectionné doit appartenir à une classe.
- `<file>:<lgn>:<col> : <field> isn't defined in this class` : le champ sélectionné n'est pas défini dans la classe choisie.
- `<file>:<lgn>:<col> : <field> isn't a field` : l'objet sélectionné dans la classe existe, mais ce n'est pas un champ.
- `<file>:<lgn>:<col> : can't access to protected field <field>` : le champ sélectionné a un attribut *PROTECTED* dans sa classe, on ne peut y accéder.

Pour l'appel d'une méthode "_._(_)"

- `<file>:<lgn>:<col> : object isn't a class` : la méthode sélectionnée doit appartenir à une classe.
- `<file>:<lgn>:<col> : <method> isn't defined` : la méthode sélectionnée n'est pas définie dans la classe choisie.
- `<file>:<lgn>:<col> : MethodCall: too many args` : trop d'arguments ont été passés en entrée de la méthode.
- `<file>:<lgn>:<col> : MethodCall: too few args` : le nombre d'arguments en entrée de la méthode est insuffisant.

Pour la conversion d'une expression "(_)(_)" :

- `<file>:<lgn>:<col> : Expression is null` : on ne peut pas convertir une expression de type *null*.
- `<file>:<lgn>:<col> : Type is null` ou `cannot cast null type` : on ne peut

pas convertir une expression en un type *null*.

— `<file>:<lgn>:<col> : cannot cast void type` : on ne peut pas convertir une expression en un type *void*.

— `<file>:<lgn>:<col> : impossible cast` : on ne peut pas convertir l'expression en ce nouveau type.

3.6 Erreurs : sortie

Pour l'utilisation de **print**, **println**, **printx** et **printlnx** :

— `<file>:<lgn>:<col> : Print contextual Error` : on ne peut pas afficher une expression de ce type.

Pour l'utilisation de **return** :

— `<file>:<lgn>:<col> : unexpected return type` : la méthode définie ne retourne pas le type attendu.

— `<file>:<lgn>:<col> : returned type can't be void` : une méthode comportant un return ne peut pas avoir un type *void*.

4

SECTION

Génération de code

4.1 Limitations/Points propres à l'implémentation du compilateur

Tout d'abord, il s'agit de précisé que la génération de code de notre compilateur fonctionne comme voulu sur la quasi-totalité des programmes Deca. Nous allons lister ici les derniers comportements involontaires qui résultent en une erreur de compilation :

— lors d'implémentation de corps de méthode avec `asm(...)`, il n'est pas possible d'afficher le caractère backslash (\).

— notre gestion des registres et de la pile induit une erreur lors du calcul d'expression longue qui mélange opérateur binaire et unaire. Il est donc conseillé à l'utilisateur d'effectuer les opérations unaires au préalable du calcul de la longue expression. Par exemple, il est déconseillé de faire :

```
1        int x = 0;
2        x = a + -(-(b*b)+c*(-(a+(b*c*(-a+b+(c*(c-d)*-(a*(a+b*(-c+(-a*b+(a*b+c))))
     )))))));
3
```

— Les réalisations de cast lorsque les registres sont limités à 4 provoquent des comportements inatendus sur l'objet resultant de cette opération. L'exemple suivant induit une erreur à l'exécution lorque la compilation est lancée avec le paramètre `-r 4`

```
class A {
    int x = 1;
}
class B extends A {
    int x = 2;
}
{
    B b = new B();
    A a = (A)(b);
    print(a.x);
}
```

4.2 Liste des message d'erreur et les configurations qui les provoquent

Toute les configurations provoquant un messages d'erreurs sont repertoriées dans le dossier `/src/test/deca/codegen/invalid`. Nous reprenons ici leur description :

• Définition d'une classe dans une autre classe.

```
// Error: stack_overflow_error
class A {
    B b = new B();
}
class B {
    A a = new A();
}
{
    B b = new B();
}

```

• Débordement arithmétique des flottants.

```
// Error: overflow_error

{
    float num = 10000000000000000000000000000000.0;
    float denom = 0.00000000000000000000000000000001;
    float res = num / denom;
}

```

- Division flottante par zéro

```
// Error: overflow_error

{
    float res = 1.0 / 0.0;
}

```

- Division entière par zéro

```
// Error: zero_division

{
    int num = (1+1*(1+1*(1+1)))/(2-2);
}

```

- Cast impossible

```
// Error: cast_error

class A { }
class B extends A {}

{
   A a = new A();
   B b = (B)(a);
}

```

- Modulo par zéro

```
// Error: zero_division

{
    int num = 1 % 0;
}

```

- Absence de return pour une méthode de type autre que `void`

```
// Error: method A.getX needs a return

class A {
    int x = 1;
    int getX() {
        int a = 2;
    }
}
{
    A a = new A();
    print(a.getX());
}

```

- Déférencement d'un objet null

```
// Error: null_dereference

class A {
    int x = 1;
    int getX() {
        int b = 2;
        return x;
    }
}
{
    A a;
    print(a.getX());
}

```

5

SECTION

Extension TRIGO

5.1 Pré-implémentation en Java

L'extension Trigo contient des méthodes pour effectuer des opérations numériques de base telles que les fonctions élémentaires trigonométriques.

Les fonctions cosinus (cos), sinus (sin), arctan (atan), arcsin (arcsin) et enfin la fonction ulp y ont été implémentées.
Les spécifications de qualité d'implémentation concernent une propriétés, la précision du résultat retourné de la méthode. La précision des méthodes mathématiques à virgule flottante est mesurée en termes d'ulps, unités à la dernière place. Pour un format à virgule flottante donné, un ulp d'une valeur de nombre réel spécifique est la distance entre les deux valeurs à virgule flottante entre parenthèses cette valeur numérique.
Les fonctions trigonométriques se basent sur des approximations polynomiales et plus précisément leurs développements de Taylor. La fonction arcsinus est elle calculées grâce à la fonction arctangente. Toutes ces fonctions fonctionnement correctement sur leurs intervalles respectifs.
La fonction ulp(float x) quand à elle fonctionne correctement et semble obtenir les même résultats que que la fonction upl(float x) de la classe java.lang.Math.
Les écarts de précisions des fonctions trigonométrique sont un facteur essentiel. Pour cela, toutes nos fonctions ont été comparés à celles de la classe java.lang.Math. En général, ces fonctions ne dépassent pas d'erreurs supérieurs à 3 ULP. Cependant, plus on s'éloigne de l'intervalle [-π, π], plus l'erreur augmente (comme le Unit in the Last Place). Les courbes d'erreurs sont disponibles sur la Figure 1. La figure 2 montre bien que les

fonctions disponibles ont bien les qu'elles devraient avoir en théorie.

Details des méthodes

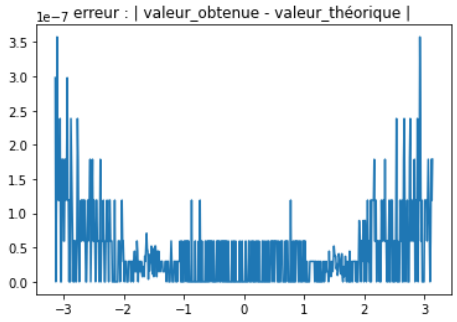
cos (float a)
Retourne le cosinus d'un angle en radians. Cas spéciaux :
•Si l'argument est NaN ou l'infini, alors le résultat est Nan.
Entrée : a - un angle, en radians.
Sortie : le cosinus de l'argument.

sin (float a)
Retourne le sinus d'un angle en radians. Cas spéciaux :
•Si l'argument est NaN ou l'infini, alors le résultat est Nan.
•Si l'argument est zéro, alors le resultat sera .
Entrée : a - un angle, en radians.
Sortie : le sinus de l'argument.

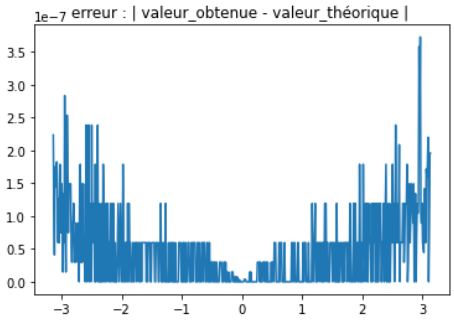
arctan (float a)
Retourne l'arctangente d'un angle en radians. Cas spéciaux :
•Si l'argument est NaN ou l'infini, alors le résultat est Nan.
•Si l'argument est zéro, alors le resultat sera .
Entrée : a - un angle, en radians.
Sortie : l'arctangente de l'argument.

arcsin (float a)
Retourne l'arcsinus d'un angle en radians. Cas spéciaux :
•Si l'argument est NaN ou l'infini, alors le résultat est Nan.
•Si l'argument est zéro, alors le resultat sera .
Entrée : a - un angle, en radians.
Sortie : l'arcsinus de l'argument.

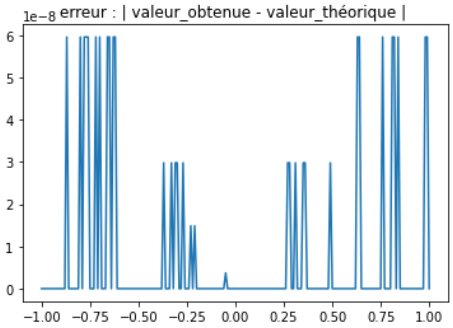
ulp (float a)
Retourne l'ulp (Unit in the Last Place) de l'argument. Cas spéciaux :
•Si l'argument est NaN, alors le résultat est Nan.
•Si l'argument est l'infini, alors le résultat est l'infini.
Entrée : a - un flottant.
Sortie : l'ulp de l'argument.



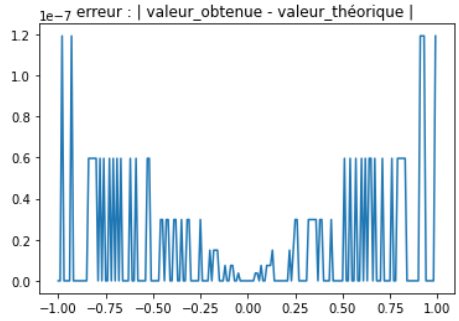
(a) erreur de la fonction cosinus



(b) erreur de la fonction sinus

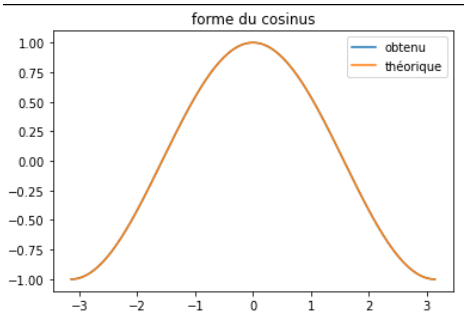


(c) erreur de la fonction arctangente

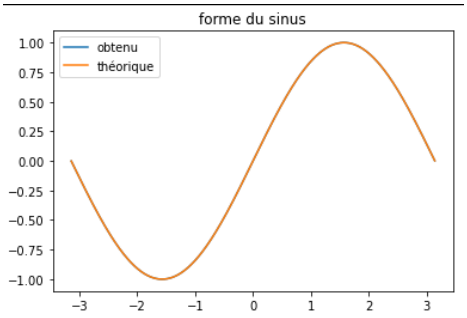


(d) erreur de la fonction arcsinus

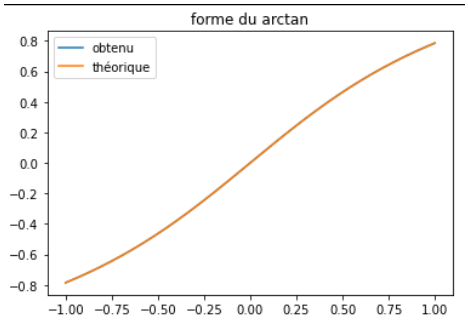
FIGURE 1 – Erreur des fonctions trigonométrique



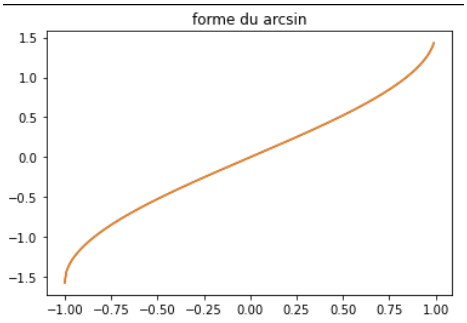
(a) forme de la fonction cosinus



(b) forme de la fonction cosinus



(c) forme de la fonction arctangente



(d) forme de la fonction arcsinus

FIGURE 2 – Erreur des fonctions trigonométrique

5.2 Implémentation en Deca et limitation

L'ensemble des fonction enumérées ci-dessus ont été adaptées et implémentées en deca et sont présentes à l'emplacement `src/main/resources/include/` dans l'en-tête Polynomial.decah. Certaines parties sont directement écrites en assembleur pour gagner

en performance. Cependant, seules les fonctions `cos`, `sin` et `ulp` sont exécutables avec notre compilateur `decac`. En effet les fonctions `arcsin` et `arctan` produisent des erreurs d'assembleur à l'exécution.
De plus, il peut être nécessaire de lancer la compilation d'un programme utilisant l'extension avec le paramètre `-n` qui permet de supprimer les tests de débordement arithmétique sur les flottants. Si les tests sont conservés, l'exécution peut s'arrêter à cause d'une `overflow_error`.
Les résultats obtenus sont donc moins précis que ceux espérés et à ce jour aucun test de précision n'a pu être effectué sur les résultats obtenus par l'extension Trigo.

5.3 Utilisation de l'extension

Pour utliser l'extension Trigo dans un programme deca, il suffit d'inclure l'en-tête Math.decah au début de votre fichier et d'instancier un objet de la class Math sur lequel on peut appeler les différentes méthodes. Voici un exemple d'utilisation :

```
1   #include "Math.decah"
2
3   // Déclarations des classes du programme
4   Class A {
5   }
6   ...
7
8   // Programme principal
9   {
10      Math math = new Math();
11
12      // utilisation des fonction de Math
13      print(math.sin(0));
14  }
```