

# Compilateur pour le langage **DECAF**

**But du projet :** Réaliser un compilateur pour le langage DECAF décrit ci-dessous. DECAF est un langage impératif simple similaire à C ou à Pascal. Ce compilateur produira en sortie du code assembleur MIPS qui devra s'exécuter sans erreurs à l'aide d'un simulateur de microprocesseur MIPS.

## 1 Présentation du langage

## 1.1 Considérations lexicales

Toutes les mots clés de DECAF sont en minuscules. Les mots clés et les identificateurs sont sensibles à la casse. Par exemple, `if` est un mot clé, alors que `IF` est un nom de variable; `foo` et `FOO` sont deux noms différents qui font référence à deux variables distinctes.

Les mots réservés sont :

```
boolean break class continue else false for if int return true void
```

Notons que **Program** n'est pas un mot clé (voir ci-dessous), mais un identificateur ayant un sens spécifique dans certaines circonstances.

Les commentaires commencent avec // et se terminent avec la fin de ligne.

Les chaînes de caractères sont composées de caractères (<char>s) et délimitées par des doubles quotes ("). Une constante caractère consiste en un <char> entouré de simples quotes (').

Les nombres en DECAF sont des entiers signés sur 32 bits, c'est-à-dire des valeurs décimales entre -2147483648 et 2147483647. Si une suite débute avec **0x**, alors ces deux premiers caractères et la plus longue suite de caractères issus de [0-9a-f-A-F] forment une constante hexadécimale. Si une suite débute avec un chiffre décimal, et pas **0x**, alors le plus long préfixe de chiffres décimaux forme une constante entière décimale. Notons que la vérification d'une valeur représentable est effectuée plus tard. Une longue suite de chiffres (par exemple 123456789123456789) est bien reconnue comme un token unique.

Un <char> est tout caractère ASCII imprimable (les valeurs ASCII entre les valeurs décimales 32 et 126, ou octales entre 40 et 176), autres que double quote ("), simple quote ('), ou backslash (\), mais incluant les suites de deux caractères “\ ” pour spécifier la double quote, “\ ’” pour spécifier la simple quote, “\\” pour spécifier le backslash, “\t” pour spécifier la tabulation, et “\n” pour spécifier le saut de ligne.

## 1.2 Grammaire du langage

Notation :

<foo>	signifie que foo est un symbole non terminal
<b>foo</b>	(en gras) signifie que <b>foo</b> est un symbole terminal
[ <i>x</i> ]	signifie zéro ou une occurrence de <i>x</i> ( <i>x</i> est optionnel)
<i>x</i> *	signifie zéro ou plusieurs occurrences de <i>x</i>
<i>x</i> <sup>+</sup> ,	une liste d'un ou plusieurs <i>x</i> séparés par des virgules
{ }	les accolades sont utilisées pour le regroupement
	est le séparateur d'alternatives

<program>	→ <b>class Program</b> '{' <field_decl>* <method_decl>* '}'
<field_decl>	→ <type> { <id>   <id> '[' <int_literal> ']' } <sup>+</sup> , ;
<method_decl>	→ { <type>   <b>void</b> } <id> ( [ { <type> <id> } <sup>+</sup> , ] ) <block>
<block>	→ '{' <var_decl>* <statement>* '}'
<var_decl>	→ <type> <id> <sup>+</sup> , ;
<type>	→ <b>int</b>   <b>boolean</b>
<statement>	→ <location> <assign_op> <expr> ;   <method_call> ;   <b>if</b> ( <expr> ) <block> [ <b>else</b> <block> ]   <b>for</b> <id> = <expr> , <expr> <block>   <b>return</b> [<expr>] ;   <b>break</b> ;   <b>continue</b> ;   <block>
<assign_op>	→ =   +=   -=
<method_call>	→ <method_name> ( [<expr> <sup>+</sup> ,] )
<method_name>	→ <id>
<location>	→ <id>   <id> '[' <expr> ']'
<expr>	→ <location>   <method_call>   <literal>   <expr> <bin_op> <expr>   - <expr>   ! <expr>   ( <expr> )
<bin_op>	→ <arith_op>   <rel_op>   <eq_op>   <cond_op>
<arith_op>	→ +   -   *   /   %
<rel_op>	→ <   >   <=   >=
<eq_op>	→ ==   !=

<cond_op>	→	&&
<literal>	→	<int_literal>   <char_literal>   <bool_literal>
<id>	→	<alpha> <alpha_num>*
<alpha_num>	→	<alpha>   <digit>
<alpha>	→	<b>a</b>   <b>b</b>   ...   <b>z</b>   <b>A</b>   <b>B</b>   ...   <b>Z</b>   <b>_</b>
<digit>	→	<b>0</b>   <b>1</b>   <b>2</b>   ...   <b>9</b>
<hex_digit>	→	<digit>   <b>a</b>   <b>b</b>   <b>c</b>   <b>d</b>   <b>e</b>   <b>f</b>   <b>A</b>   <b>B</b>   <b>C</b>   <b>D</b>   <b>E</b>   <b>F</b>
<int_literal>	→	<decimal_literal>   <hex_literal>
<decimal_literal>	→	<digit> <digit>*
<hex_literal>	→	<b>0x</b> <hex_digit> <hex_digit>*
<bool_literal>	→	<b>true</b>   <b>false</b>
<char_literal>	→	' <char> '
<string_literal>	→	" <char>* "

### 1.3 Structure d'un programme en DECAF

Un programme en DECAF consiste en une déclaration unique de classe, pour une classe appelée **Program**. Cette déclaration de classe consiste en un nombre quelconque de <field\_decl> et un nombre quelconque de <method\_decl>. Les <field\_decl> permettent de déclarer des variables qui peuvent être accédées globalement par toutes les méthodes du programme. Les <method\_decl> permettent de déclarer des fonctions ou procédures. Le programme doit contenir une déclaration d'une méthode appelée **main** qui n'a pas de paramètres. L'exécution d'un programme en DECAF démarre avec la méthode **main**.

### 1.4 Types

Il y a deux types de base en DECAF, **int** et **boolean**. De plus, il y a des tableaux d'entiers (**int [ N ]**) et des tableaux de booléens (**boolean [ N ]**).

Les tableaux ne peuvent être déclarés qu'en portée globale (<field\_decl>). Tous les tableaux sont uni-dimensionnels et ont une taille fixée à la compilation. Ils sont indexés de 0 à  $N - 1$ , où  $N > 0$  est la taille du tableau. La notation usuelle utilisant les crochets est utilisée pour indiquer les tableaux.

### 1.5 Règles de portée

DECAF possède des règles de portée qui sont simples et très restrictives. Tous les identificateurs doivent être définis (textuellement) avant d'être utilisés. Par exemple :

- une variable doit être déclarée avant d'être utilisée.
- une méthode ne peut être appelée qu'à partir d'une instruction située après son entête (notons que les appels récursifs sont autorisés).

Il y a au moins deux portées valides en tout point d'un programme en DECAF : la portée globale et la portée de la méthode. La portée globale concerne les variables et les méthodes introduites dans la déclaration de classe du programme. La portée d'une méthode concernent les variables et les paramètres introduits dans la déclaration de la méthode. Des portées locales supplémentaires existent à l'intérieur de chaque <block> dans le programme. Ceux-ci peuvent survenir après des instructions **if** ou **for**, ou insérés partout où un <statement> est possible. Un identificateur de la portée d'une méthode supplante

le même identificateur de portée globale. De même, les identificateurs de portées locales supplantent les identificateurs de portées moins profondes, de portée d'une méthode et de portée globale.

Les noms de variables définis dans la portée d'une méthode, ou de portée locale, supplantent les noms de méthodes de portée globale. Dans ce cas, l'identificateur ne peut être utilisé que pour une variable, jusqu'à que la variable quitte la portée.

Dans le cas particulier de l'instruction de boucle **for**, la variable compteur de boucle `<id>` est déclarée implicitement dans l'entête de la boucle (voir la sous-section 1.9).

Aucun identificateur ne peut être défini plus d'une fois dans la même portée. Ainsi, les noms de variables et de méthodes doivent être distincts dans la portée globale, et les noms de variables locales et de paramètres doivent être distincts dans chaque portée locale.

## 1.6 Emplacements

DECAF possède deux types d'emplacements : les variables scalaires locales/globales et les éléments de tableaux (globaux). Chaque emplacement possède son type. Les emplacements de types **int** et **boolean** contiennent des valeurs entières et booléennes, respectivement. Les emplacements de types **int** [ *N* ] et **boolean** [ *N* ] dénotent des éléments de tableaux. Comme les tableaux possèdent des tailles fixées à la compilation, ils peuvent être alloués dans la zone de données statiques du programme, et n'ont pas besoin d'être alloués sur le tas.

Chaque emplacement est initialisée à une valeur par défaut lorsqu'il est déclaré. Les entiers ont la valeur zéro par défaut, et les booléens ont la valeur **false** par défaut. Les variables locales doivent être initialisées lorsque la portée de déclaration est entrée. Les éléments de tableaux sont initialisés lorsque le programme démarre.

## 1.7 Affectations

Les affectations ne sont autorisées que pour les valeurs scalaires. Pour les types **int** et **boolean**, une affectation `<location> = <expr>` copie la valeur résultant de l'évaluation de `<expr>` dans `<location>`. L'affectation `<location> += <expr>` incrémente la valeur stockée dans `<location>` par `<expr>`, et est uniquement valide si `<location>` et `<expr>` sont de type **int**. L'affectation `<location> -= <expr>` décrémente la valeur stockée dans `<location>` par `<expr>`, et est uniquement valide si `<location>` et `<expr>` sont de type **int**.

Les symboles `<location>` et `<expr>` dans une affectation doivent être de mêmes types. Pour le type tableau, `<location>` et `<expr>` doivent faire référence à un unique élément de tableau qui est aussi une valeur scalaire.

Il est autorisé d'affecter une valeur à une variable paramètre à l'intérieur du corps d'une méthode. Une telle affectation est limitée à la portée de la méthode.

## 1.8 Invocation de méthode et retour

L'invocation d'une méthode implique (1) le passage de valeurs d'arguments de l'appelant vers l'appelé, (2) l'exécution du corps de l'appelé, et (3) le retour vers l'appelant, éventuellement avec un résultat.

Le passage d'arguments est défini dans les termes de l'affectation : les arguments formels d'une méthode sont considérés de la même manière que les variables locales de la méthode, et sont initialisés, par des affectations, aux valeurs résultant de l'évaluation des expressions formant les arguments d'appel. Les arguments sont évalués de gauche à droite.

Le corps de l'appelé est ensuite exécuté en exécutant ses instructions en séquence.

Une méthode qui n'a pas de résultat, c'est-à-dire dont le type de résultat est **void**, peut uniquement être appelée comme une instruction, c'est-à-dire qu'elle ne peut pas être utilisée dans une expression. Une telle méthode redonne le contrôle à l'appelant lorsque **return** est exécuté (aucune expression résultat n'est autorisée), ou lorsque la fin textuelle de l'appelé est atteinte.

Une méthode qui renvoie un résultat peut être appelée en faisant partie d'une expression, au quel cas le résultat de l'appel est le résultat de l'évaluation de l'expression située dans l'instruction **return**, lorsque cette instruction est atteinte. Il est illégal que le contrôle atteigne la fin textuelle d'une méthode qui retourne un résultat.

Une méthode qui retourne un résultat peut aussi être appelée comme une instruction. Dans ce cas, le résultat est ignoré.

## 1.9 Structures de contrôle

### if

L'instruction **if** possède la sémantique habituelle. Tout d'abord, `<expr>` est évalué. Si le résultat est **true**, alors la branche **true** est exécutée. Sinon, la branche **else** est exécutée, si elle existe. Puisque DECAF demande à ce que les branches **true** et **else** soit délimitées par des accolades, il n'y a pas d'ambiguïté à faire correspondre une branche **else** avec son instruction **if** correspondante.

### for

Le symbole `<id>` est la variable d'index qui supplante toute variable de même nom déclarée dans une portée englobante, si elle existe. La variable d'index déclare une variable entière dont la portée est limitée au corps de la boucle. Le premier `<expr>` est la valeur initiale de la variable d'index et le deuxième `<expr>` est sa valeur finale. Chacune de ces expressions est évaluée une seule fois, à l'entrée dans la boucle. Chaque expression doit résulter en une valeur entière. Le corps de boucle est exécuté si la valeur courante de la variable d'index est inférieure ou égale à la valeur finale. Après une exécution du corps de boucle, la variable d'index est incrémentée de 1, et la nouvelle valeur est comparée à la valeur finale, pour savoir si une nouvelle itération doit être exécutée.

## 1.10 Expressions

Les expressions suivent des règles normales d'évaluation. En l'absence d'autres contraintes, les opérateurs de mêmes priorités sont évalués de la gauche vers la droite. Les parenthèses peuvent être utilisées pour modifier cet ordre.

Un emplacement dans une expression est évalué à la valeur contenue dans l'emplacement.

Les invocations de méthodes dans les expressions sont discutées ci-dessus (voir «Invocation de méthode et retour»).

Les constantes entières sont évaluées à leur valeur entière. Les constantes caractères sont évaluées à leur valeur entière ASCII (par exemple, **'A'** représente l'entier 65). Le type d'une constante caractère est **int**.

Les opérateurs arithmétiques (`<arith_op>` et le moins unaire) ont les significations et priorités habituelles, ainsi que les opérateurs relationnels (`<rel_op>`). L'opérateur `%` calcule le reste de la division de ses opérandes.

Les opérateurs relationnels sont utilisés pour comparer des expressions entières. Les opérateurs d'égalité, `==` et `!=` sont définis pour les types **int** et **boolean**, et peuvent être utilisés pour comparer deux expressions de mêmes types.

Le résultat d'un opérateur relationnel ou d'un opérateur d'égalité est de type **boolean**.

Les expressions avec connecteurs booléens `&&` et `||` sont évaluées en court-circuit : Le deuxième opérande n'est pas évalué si le résultat du premier opérande détermine la valeur de toute l'expression, c'est-à-dire si le résultat est faux pour `&&` et vrai pour `||`.

Priorités des opérateurs, de la plus forte à la plus faible :

Opérateurs	Commentaires
-	moins unaire
!	non logique
* / %	multiplication, division, reste
+ -	addition, soustraction
< <= >= >	relationnel
== !=	égalité
&&	conditionnel et
	conditionnel ou

Notons que la grammaire du langage DECAF donnée ci-dessus ne reflète pas ces priorités.

## 1.11 Entrées/Sorties

Des méthodes permettant d'interagir avec les programmes devront obligatoirement être implémentées. Celles-ci seront considérées comme des routines d'une bibliothèque externe.

- **void** WriteInt(**int** <expr>) : affiche la valeur entière de <expr>.
- **void** ReadInt(**int** <location>) : permet la saisie au clavier d'une valeur entière, stockée dans <location>.
- **void** WriteBool(**boolean** <expr>) : affiche la valeur booléenne, **true** ou **false**, de <expr>.
- **void** WriteString(<string\_literal>) : affiche la chaîne de caractères <string\_literal>.

## 1.12 Règles sémantiques

Ces règles s'ajoutent aux contraintes imposées par la grammaire du langage DECAF. Un programme qui est grammaticalement bien formé et qui ne viole aucune des règles qui suivent est un programme légal. Un compilateur robuste doit vérifier explicitement chacune de ces règles, et doit générer un message d'erreur décrivant chaque violation qu'il est capable de détecter. Un compilateur robuste doit générer au moins un message d'erreur pour chaque programme illégal, et aucun message pour un programme légal.

1. Aucun identificateur ne doit être déclaré deux fois dans la même portée.
2. Aucun identificateur ne doit être utilisé avant d'avoir été déclaré.
3. Le programme doit contenir la définition d'une méthode appelée **main** qui n'a pas de paramètres. Notons que comme l'exécution du programme démarre par la méthode **main**, toute méthode définie après **main** ne sera jamais exécutée.
4. Le symbole <int\_literal> dans une déclaration de tableau doit être supérieur à 0.
5. Le nombre et types des arguments dans un appel de méthode doivent être les mêmes que le nombre et types des paramètres formels dans la déclaration de la méthode : les signatures doivent être identiques.
6. Si un appel de méthode est utilisé dans une expression, la méthode doit retourner un résultat.
7. Une instruction **return** ne doit pas posséder de valeur de retour, à moins qu'elle apparaisse dans le corps d'une méthode qui est déclarée comme retournant un résultat.
8. L'expression dans une instruction **return** doit être de même type que le type du résultat déclaré dans la définition de la méthode englobante.
9. Un <id> utilisé comme une <location> doit correspondre à une variable locale/globale ou à un paramètre formel.
10. Pour tous les emplacements de la forme <id>[<expr>]
  - (a) <id> doit être une variable tableau, et
  - (b) le type de <expr> doit être **int**.
11. Le symbole <expr> dans une instruction **if** doit être de type **boolean**.
12. Les opérandes des opérateurs <arith\_op> et <rel\_op> doivent être de type **int**.
13. Les opérandes des opérateurs <eq\_op> doivent être de mêmes types, soit **int**, soit **boolean**.
14. Les opérandes des opérateurs <cond\_op> et du non logique (!) doivent être de type **boolean**.
15. Les symboles <location> et <expr> dans une affectation, <location> = <expr>, doivent être de mêmes types.
16. Les symboles <location> et <expr> dans une affectation d'incréméntation/décréméntation, <location> += <expr> et <location> -= <expr>, doivent être de type **int**.
17. La valeur initiale <expr> et la valeur finale <expr> du **for** doivent être de type **int**.
18. Toutes les instructions **break** et **continue** doivent être contenues dans le corps d'une boucle **for**.

## 1.13 Vérifications dynamiques

Le générateur de code du compilateur doit générer des instructions de détection de violations découvertes à l'exécution :

1. La valeur d'indice d'un élément de tableau doit être entre la borne inférieure (0) et la borne supérieure.

2. Le contrôle ne doit pas atteindre la fin textuelle d'une méthode (sans exécution d'un **return** avec valeur de retour) qui est déclarée comme retournant un résultat.

Lorsqu'une erreur d'exécution survient, un message d'erreur approprié doit être affiché, et le programme doit se terminer. Ces messages doivent aider le programmeur à corriger son erreur.

### 1.14 Exemple de programme

```
// Exemple de programme en DECAF
class Program {
    int compteur, diviseur ;
    int valeurs[4] ;
    boolean test;

    boolean lecture_valeurs(int nombre) // Saisie des valeurs entières
    {
        boolean test;
        WriteString("Il faut saisir ");
        WriteInt(nombre);
        WriteString("valeurs entières\n");
        for i = 0, nombre-1
        {
            WriteString("Entrez la valeur ");
            WriteInt(i+1);
            WriteString(" : \n");
            ReadInt(valeurs[i]);
            test = valeurs[i] > 0; // vérification
            if (!test) {
                WriteString("Les valeurs doivent être strictement positives !\n");
                return false;
            }
        }
        return true;
    }

    int moyenne(int nombre) { // Calcul de la moyenne des valeurs
        int somme;
        for i = 0, nombre-1 {
            somme += valeurs[i];
        }
        return somme/nombre;
    }

    void main() {
        int nombre;
        WriteString("Nombre de valeurs : ");
        ReadInt(nombre);
        if (lecture_valeurs(nombre)) {
            WriteString("Moyenne = ");
            WriteInt(moyenne(nombre));
            WriteString("\n");
        }
        else { WriteString("Erreur\n"); }
    }
}
```

## 2 Génération de code

Le code généré devra être en assembleur MIPS R2000. L'assembleur est décrit dans les documents fournis. Le code assembleur devra être exécuté à l'aide du simulateur de processeur MIPS *SPIM*<sup>1</sup> (il

---

1. <http://spimsimulator.sourceforge.net>

existe un package debian/ubuntu) ou *Mars*<sup>2</sup>.

### 3 Aspects pratiques et techniques

Le compilateur devra être écrit en C à l'aide des outils Lex (flex) et Yacc (bison).

Ce travail est à réaliser en équipe composée de quatre étudiant.e.s dans le cadre du cours de *Compilation* et du cours de *Conduite de Projets*, et à rendre à la date indiquée par vos enseignants en cours et sur Moodle. Une démonstration finale de votre compilateur sera faite durant la dernière séance de TP. Vous devrez rendre sur Moodle dans une archive :

- Le code source de votre projet complet dont la compilation devra se faire simplement par la commande « make ». Le nom de l'exécutable produit doit être « decaf »
- Un document détaillant les capacités de votre compilateur, c'est-à-dire ce qu'il sait faire ou non. Soyez honnêtes, indiquez bien les points intéressants que vous souhaitez que le correcteur prenne en compte car il ne pourra sans doute pas tout voir dans le code.
- Un jeu de tests.

Votre compilateur devra fournir les options suivantes :

- `-version` devra indiquer les membres du projet.
- `-tos` devra afficher la table des symboles.
- `-o <name>` devra écrire le code résultat dans le fichier `name`.

### 4 Recommandations importantes

Écrire un compilateur est un projet conséquent, il doit donc impérativement être construit incrémentalement en validant chaque étape sur un plus petit langage et en ajoutant progressivement des fonctionnalités ou optimisations. Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis de passer au débogage ! Le résultat de cette démarche serait très probablement nul, c'est-à-dire un compilateur qui ne fonctionne pas du tout ou alors qui reste très bogué.

Par conséquent, nous vous conseillons de développer tout d'abord un compilateur *fonctionnel* mais *limité* à la traduction d'expressions arithmétiques simples, sans structures de contrôle. À partir d'une telle version fonctionnelle, il vous sera plus aisé de la faire évoluer en intégrant telle ou telle fonctionnalité, ou en considérant des expressions plus complexes, ou en intégrant telle ou telle structure de contrôle. De plus, même si votre compilateur ne remplira finalement pas tous les objectifs, il sera néanmoins capable de générer des programmes corrects et qui « marchent » !

### 5 Précisions concernant la notation

- Si votre projet ne compile pas ou plante directement, la note 0 (zéro) sera appliquée : l'évaluateur n'a absolument pas vocation à aller chercher ce qui pourrait éventuellement ressembler à quelque chose de correct dans votre code. Il faut que votre compilateur s'exécute et qu'il fasse quelque chose de correct, même si c'est peu.
- Si vous manquez de temps, préférez faire moins de choses mais en le faisant bien et de bout en bout : on préférera un compilateur incomplet mais qui génère un programme MIPS exécutable plutôt qu'une analyse syntaxique seule.
- Élaborez des tests car cela fait partie de votre travail et ce sera donc évalué.
- Faites les choses dans l'ordre et focalisez sur ce qui est demandé. L'évaluateur pourra tenir compte du travail fait en plus (par exemple des optimisations de code) seulement si ce qui a été demandé a été fait et bien fait.
- Une conception modulaire et lisible sera fortement appréciée (et inversement).

---

2. <http://courses.missouristate.edu/kenvollmar/mars>