Manuel utilisateur

1 Description du compilateur

Le compilateur Decac est un programme informatique, qui prend en entrée un ou plusieurs fichiers ‘.deca’ et, si ces fichiers respectent les règles du langage Deca, alors le compilateur produit en sortie un ou des fichiers ‘.ass’ exécutables par la machine abstraite ima. Dans le cas contraire, la compilation sera arrêtée et un message d’erreur approprié affichera l’erreur. Decac produit, à partir de fichiers avec un haut niveau d’abstraction et compréhensibles par un être humain, des fichier en code machine afin de créer un programme exécutable par la machine abstraite ima. Pour cela, le programme du compilateur va réaliser plusieurs vérifications, ces vérifications sont réparties en trois parties. On nommera ces parties A, B et C et elles représenteront l’ensemble des vérifications qui sont effectuées sur chaque fichier.

Tout d’abord, la partie A s’occupe de vérifier la validité lexicale ainsi que la validité syntaxique de chacun des fichiers. Une vérification lexicale sera effectuée, c’est à dire que le ou les fichiers ‘.deca’ seront lu par le programme et chaque caractère lu sera analysé pour s’assurer qu’il appartient bien au langage Deca. Dans cette étape, il n’est vérifié que l’appartenance de chaque mot et non pas la logique de leur enchaînement. Si tous les mots sont reconnus alors ils sont associés à des jetons (tokens) pour l’étape suivante. La seconde et dernière partie de l’étape A est une vérification syntaxique, ce qui veut dire que l’on vérifie la validité syntaxique du programme. Par exemple, le fait qu’une instruction se finisse par un point virgule ‘;’, l’absence de ce point virgule dans un code source arrêtera la compilation et générera une erreur syntaxique appropriée pour aider l’utilisateur à résoudre au mieux le problème syntaxique qui a été relevé par le compilateur. De plus, cette dernière étape construit, lors de la vérification, un arbre représentant l’ensemble du programme source. Cet arbre détails les différentes parties qui composent le programme, chaque déclaration de variable, instruction, déclaration de classe … Finalement, si aucune erreur n’a été détecté lors des deux étapes de vérifications, alors l’arbre représentant le code source sera donné en sortie pour la partie suivante.

La partie B concerne d’une part la vérification contextuelle du code source et d’autre part la décoration de l’arbre construit lors de la partie précédente. Tout d’abord, la

vérification contextuelle s'effectue en trois passes, c'est à dire que l'on parcourt l'arbre obtenu à la partie A trois fois. Pour chacune de ces passes, de nouvelles vérifications sont effectuées. Lors de la première passe, on effectue une vérification du nom des classes ainsi que la hiérarchie de ces même classes. Ces vérifications contextuelles consistent à s'assurer qu'il n'y a pas de double déclaration de classe et lier les classes qui ont des liens d'héritages pour les prochaines passes. Pour la seconde passe, on vérifie l'ensemble des attributs de chaque classe en s'assurant qu'il n'y a pas de double déclaration ou d'erreur d'initialisation. De plus, les signatures des méthodes sont aussi vérifiées, c'est à dire leur nom et leurs paramètres. Cela permet donc de s'assurer qu'il n'y a pas de double définition de fonction et de vérifier les potentielles surcharges et redéfinitions de méthodes. Finalement, pour la dernière passe, les vérifications contextuelles portent sur les corps de méthodes, en s'assurant, par exemple, que chaque variable utilisée est visible et la priorité de visibilité, entre paramètres et attributs. Cette troisième passe, vérifie aussi le contexte dans le main, qui n'avait pas encore été parcourue lors des deux passes précédentes. Il s'agit aussi de vérifications contextuelles proche de celles effectuées plus tôt, il est vérifié que les types des expressions concordent lors d'affectations, d'opérations binaires et unaires, etc … En ce qui concerne la partie décoration de l'arbre abstrait, ces décorations consistent à associer un type et/ou une définition aux différents éléments de l'arbre. Cette partie est réalisée au cours des différentes passes, et décore les éléments qui sont vérifiés (passe 1: décoration des classes, passe 2: décorations des attributs et des signatures des différentes méthodes, passe 3: décoration des corps de méthode et du main).

Finalement, la partie C, la génération de code machine, tout comme la partie précédente, s'effectue avec des passes, en l'occurrence deux parcours de l'arbre abstrait décoré. Lors de la première passe, uniquement les classes sont parcourues et plus précisément les méthodes de ces classes dans le but de produire le code machine permettant de construire la table contenant les méthodes des différentes classes. Cela permet donc d'associer à chaque classe les méthodes qui lui sont propres, mais aussi les méthodes potentiellement héritées. A la suite, lors de la seconde passe, on s'intéresse au reste du programme. Dans un premier temps, on associe à chaque classe, en plus de ses méthodes, chacun de ses champs avec leur potentielle initialisation en générant le code machine associé. Par la suite, pour chacune des méthodes, on associe à son label la génération de code machine correspondant à son corps de méthode (déclarations et instructions), cela permettra de faire uniquement référence au label pour exécuter la méthode qui lui est associée. Finalement, on parcourt le programme principal (main) pour générer le code machine qui lui correspond. Au cours de cette étape, une pile d'exécution avec une limite est utilisée, pour s'assurer et prévenir qu'il n'y a pas de débordement de

cette pile, on génère aussi l'instruction TSTO au début de chaque bloc. Ce qui permettra d'afficher des messages d'erreur appropriés en cas de débordement de la pile d'exécution.

2 Commandes et options

Pour utiliser le compilateur "decac", il faut utiliser utiliser un terminal dans un environnement Linux. Et respecter l'usage :

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

Chaque lettre entre '[ ]' représente une option de compilation pour le compilateur "Decac". Ces options ont pour but d'améliorer l'expérience de l'utilisateur avec le compilateur en lui proposant d'autres fonctionnalités utiles à la production d'un programme informatique.

- L'option '-b' (banner), permet d'afficher une bannière indiquant le nom de l'équipe ayant produit le compilateur. Cette option ne peut être utilisé avec d'autres options ou avec des fichiers sources.

- L'option '-p' (parse), permet d'arrêter la compilateur après les étapes de vérification lexicale et de vérification syntaxique. Si celles-ci se sont déroulées sans détection d'erreur pour le code source, alors cela affichera en sortie, la décompilation de ce qui a été analysé précédemment par le compilateur. Plus précisément, cela affichera un programme deca syntaxiquement correct correspondant à l'arbre construit lors de la partie A.

- L'option '-v' (verification), permet d'arrêter la compilation du programme source après les différentes étapes de vérifications. Cela implique les étapes de vérification lexicale et syntaxique, mais aussi l'étape de vérification contextuelle. C'est à dire que le compilateur s'arrête après l'étape B, si une erreur est détectée alors elle est affichée, sinon le compilateur ne produit aucune sortie si le programme source est lexicalement, syntaxiquement et contextuellement correct.

- L’option ‘-n’ (no check), a pour effet de supprimer les tests de débordements à l'exécution, lors de la partie de génération de code machine, les instructions de test de débordements ne sont pas générées:
  - débordement arithmétique
  - débordement mémoire
  - déréférencement de null

- L’option ‘-r X’ (registers), limite le nombre de registres disponibles pour la génération de code machine ima, cette option nécessite d’être suivi par un entier X symbolisant le nombre de registre maximum. Cette valeur doit être compris entre 4 et 16, dans le cas contraire une erreur sera levée, sachant que les registres disponibles vont de R0 à R15.

- L’option ‘-d’ (debug), produit des traces de debug au cours de la compilation, ce qui permet de suivre le cheminement de la compilation. Répéter cette option plusieurs fois permet d’augmenter le nombre de traces de debug, avec au maximum trois répétitions de l’option ‘-d’.

- L’option ‘-P’ (parallel) permet, dans le cas où il y a plusieurs fichiers sources à compiler, de lancer la compilation des fichiers en parallèle, ce qui a pour effet d’accélérer la compilation en effectuant celle-ci sur un thread différent pour chaque fichier source.

Parmis les options décrit ci-dessus, les options ‘-p’ (parse) et ‘-v’ (verification) ne sont pas compatibles et ne peuvent donc pas être utilisées en même temps. De plus, l’option ‘-b’ (banner), ne peut être utilisé avec aucune autre option et aucun fichier source. Si ces conditions ne sont pas respectées, une erreur sera affichée et le compilateur s’’arrêtera.

En ce qui concerne les fichiers sources, ils peuvent être désignés par leur chemin absolue, de la forme “<répertoires/fichier.deca>”. De plus, chaque fichier source doit obligatoirement avoir un nom formé avec le suffixe “.deca”, et le compilateur produira, si le fichier source passe toutes les vérifications, une fichier du même nom avec le suffixe “.ass” dans le même répertoire que le fichier source.

3 Messages d’erreurs

Le programme du compilateur decac a pour but de transformer un code source en un code machine, tout en vérifiant que ce même code source respecte les règles du langage Deca, à travers de multiples vérifications (lexicales, syntaxiques et contextuelles). Lorsque une erreur est détectée par l’une des étapes, le compilateur est donc arrêté et une message avertissant l’utilisateur est affiché en sortie.

Le format des messages d’erreurs est fait pour que l’utilisateur puisse trouver l’origine de l’erreur le plus simplement possible. Quelque soit l’étape où l’erreur est détectée, le format reste le même, de la forme:

<nom de fichier.deca>:<ligne>:<colonne>: <description informelle du problème>

Ce format permet de pouvoir, dans un premier temps, localiser le fichier source où l’erreur est apparue, puis de savoir quelle partie du fichier est en cause avec la localisation de la ligne et de la colonne. Une fois l’erreur localisée, la description du problème fournis par le compilateur permet de comprendre ce qui a créé l’erreur à l’endroit donné.

Par exemple, ce programme “chaine_incomplète.deca”, qui a une erreur lexicale au caractère 0 de la ligne 10 affichera le message d’erreur:

chaine_incomplete.deca:10:0: token recognition error at: "'chaine pas finie\n'

Ou encore, une erreur contextuelle, plus précisément d’affectation, au caractère 7 de la ligne 15, affichera le message d’erreur suivant:

affect-incompatible.deca:15:7: Types incompatibles pour affectation

En ce qui concerne les erreurs liées au débordement de la pile, aucune indication sur l’endroit où se situe l’erreur n’est donnée, car au niveau du compilateur, on ne peut pas situé la cause de ce débordement. Un simple message d’erreur est affiché:

Error: Overflow during arithmetic operation

4 Limitations

Le compilateur decac ne supporte pas l'ensemble du langage de programmation Deca, en effet, certaines fonctionnalitées ne sont pas utilisable. Cela n'empêche pas le programme decac de fonctionner sur le langage Deca essentiel, c'est à dire le langage Deca sans les conversions de types (cast) et les tests d'appartenance à une classe (instanceof). Ces limitations impliquent qu'il est impossible d'utiliser le mot clé "*instanceof*" ainsi que la conversion "*(Type)".*

Le mot clé "instanceof" est un opérateur binaire, permettant de vérifier si l'opérande de gauche est du type de classe de l'opérande de droite. Les conséquences de l'absence de cette fonctionnalitée implique des difficultés dans l'implémentation de méthodes, telle que la méthode "*equals"* de la classe "*Object"* qui utilise cet opérateur pour exclure des cas de comparaison.

La conversion de types d'objets est une expression permettant de convertir un objet en un autre sous certaines conditions. L'absence de cette fonctionnalitée implique une limitation du langage, au niveau de l'héritage, où l'abstraction peut nécessiter une conversion de type pour utiliser des méthodes particulière à une classe, ou encore dans l'utilisation de la méthode "equals" de la classe "Object", où l'unique argument est une objet de type "Object" qui doit être converti en l'objet avec lequel on veut le comparer, pour avoir accès à l'ensemble des méthodes et attributs qui composent cette classe.

Ces limitations sont dues à un manque de temps dans l'implémentation du compilateur decac. L'objectif premier étant de fournir un compilateur fonctionnel avec le moins de bug possible, la priorité a donc été de produire un compilateur complètement utilisable pour les fonctionnalités implémentées plutôt que d'ajouter des fonctionnalités supplémentaires qui ne sont pas stables et risqueraient de réduire le confiance de l'utilisateur vis à vis de la sécurité du compilateur decac.

5 Utilisation de l'extension

Le compilateur decac fournit une extension inclut dans la bibliothèque standard, cette extension trigonométrique permet de réaliser des calculs avec des fonctions mathématiques complexes et donc d'abstraire tout l'aspect technique de ces fonctions.

L’ensemble de cette extension est fournis via une classe “*Math”* possédant l’ensemble de méthodes suivant:

- *`float cos(float f)`*
- *`float sin(float f)`*
- *`float atan(float f)`*
- *`float acos(float f)`*
- *`float asin(float f)`*
- *`float ulp(float f)`*
- *`float power(float x, int p)`*
- *`float sqrt(float f)`*

L’ensemble de ces méthodes est accessibles en incluant la classe “Math” au code source, pour cela, il suffit d’écrire ‘ *#include “Math.decah” ’* en en-tête du fichier où l’on souhaite utiliser les fonctions mathématiques. De plus, il faudra instancier un objet de classe mathématique fournit pour appeler les méthodes qui lui sont associées.

Par exemple, pour calculer le sinus de la valeur 0.32, dans un bloc d’instruction situé dans un fichier source incluant le fichier de la classe “*Math”* :

```
Math math = new Math();
float result = math.sin(0.32);
println(“sinus(0.32) = “, result);
```

6 Limite de l’extension [TRIGO]

Les différentes méthodes implémentant les fonction mathématiques ont une limite de précision imposé par la machine, cette précision est propre à chaque méthode :

- La méthode “*float cos(float f)”*, implémentant la fonction mathématique cosinus, permet une précision de l'ordre de 10^-7 lorsque son paramètre a une valeur situé dans l’intervalle [-π ; π]. Cependant, lorsque la valeur du paramètre de la méthode sort de cet intervalle et est situé dans l’intervalle [-100 000 ; -π] U [π ; 100 000], la précision de la fonction est moindre et réduit à l’ordre de 10^-3,

rendant les calculs beaucoup moins précis sur les grand nombres. Finalement, il est impossible de réaliser un calcul, avec la méthode “*cos”*, sur un paramètre qui n’est pas dans un des deux intervalles utilisé ci-dessus. La valeur retournée est située entre -1 et 1.

- La méthode “*float sin(float f)”*, implémentant la fonction mathématique sinus, qui est calculée en fonction de la méthode “*cos”*, permet aussi une précision de l'ordre de 10^-7 lorsque son paramètre a une valeur situé dans l’intervalle [-π ; π]. Cependant, tout comme la méthode de laquelle elle dépend, les valeurs de son paramètre se limitent à l’intervalle [-100 000 ; 100 000], avec une réduction de la précision à l’ordre de 10^-3 lorsque l’argument sort de l’intervalle [-π ; π]. La valeur retournée est située entre -1 et 1.

- La méthode “*float atan(float f)”*, implémente la fonction mathématique arc tangente, et permet une précision à l’ordre de 10^-7 pour toutes valeurs de flottant fournis en paramètre. La grande précision de cette méthode permet un bon degré de précision pour les méthodes qui en dépendent (asin et acos). La valeur retournée est un nombre flottant entre -π/2 et π/2.

- La méthode “*float acos(float f)”*, implémente la fonction mathématique arc cosinus, et est calculée à partir de la méthode “*atan”*. Cette méthode ne permet de prendre en entrée qu’une valeur située dans l’intervalle [-1 ; 1], comme le veut la fonction mathématique arc cosinus. De plus la précision associée est de 10^-6 pour toute valeur dans l’intervalle d’entrée, la légère différence avec la fonction “*atan”,* dont elle dépend, est dûe aux calculs arithmétiques. La valeur de retour est un nombre flottant entre 0 et pi.

- La méthode “*float asin(float f)”*, implémente la fonction mathématique arc sinus, et est calculée à partir de la méthode “*atan”*. Cette méthode ne permet de prendre en entrée qu’une valeur située dans l’intervalle [-1 ; 1], comme le veut la fonction mathématique arc sinus. De plus la précision associée est de 10^-6 pour toute valeur dans l’intervalle d’entrée, la légère différence avec la fonction “*atan”*, dont elle dépend, est dûe aux calculs arithmétiques. La valeur de retour est un nombre flottant entre -π/2 et π/2.

- La méthode “*float ulp(float f)”*, implémente une fonction nommée “unit in the last place”, qui, pour un flottant retourne la distance avec le flottant le plus proche,

cela représente l'écart minimum entre deux valeurs représentables. Le but est de déterminer une précision dans les calculs numériques. La précision fournis pour cette méthode est proche de celle du langage Java, ce qui est plutôt convenable. La valeur retournée est un flottant égal à la distance.

- La méthode "*float power(float x, int p)"*, implémente la fonction mathématique de la puissance. Le premier paramètre "*float x"*, correspond à la valeur devant être multipliée, et le second paramètre "*int p"*, correspond à la puissance à laquelle sera élevé le premier paramètre, c'est à dire le nombre de fois où il sera multiplié par lui même. La précision de cette méthode est semblable à la méthode "*sqrt*" en langage Java, ce qui permet un degré de précision convenable. La valeur retournée est un flottant égal à x^p.

- La méthode "*float sqrt(float f)"*, implémente la fonction mathématique racine carré. Le paramètre doit être un nombre positif, c'est à dire appartenir à l'intervalle [0 ; +∞]. La précision de cette méthode est semblable à la méthode "*sqrt*" en langage Java, ce qui représente un niveau de précision convenable pour des calculs non professionnels.

La principale limitation de l'extension trigonométrique, proposant des fonctions mathématiques complexes, est la marge d'erreur, précision, de certaines de ses méthodes. En l'occurrence, les méthodes "*float sin(float f)"* et "*float cos(float f)"* ont une marge d'erreur convenable lorsque l'on utilise un paramètre dans l'intervalle [-π ; π], cependant, le paramètre ne peut pas excéder la valeur de 100 000 et -100 000, ce qui réduit considérablement le champs des possibilités de calculs, d'autant plus que la précision pour un paramètre dans l'intervalle [-100 000 ; -π] U [π ; 100 000], la précision est bien plus faible (de l'ordre de 10^-3), ce qui augment aussi la marge d'erreur pour ces deux méthode "*sin"* et "*cos"*.

Pour le reste des méthodes mathématiques, la précision reste satisfaisante avec un ordre de 10^-7 pour toutes les valeurs d'entrée. Cela ne permet pas une précision maximale et reste une limitation importante de cette extension.