Introduction à la compilation de Deca sans objet

version : 18 avril 2023 à 13:46

Exercice 1 (Votre premier programe Deca). Écrire un programme Deca qui demande un entier à l'utilisateur et compare cet entier avec le nombre 17. En sortie, il affiche la réponse sous la forme "`17 <` x" ou "x `<= 17`" (où x est la valeur du nombre tapé par l'utilisateur).

Pour écrire ce programme, vous devez vous appuyer sur la spécification formelle de la syntaxe d'entrée qui définie dans [**Lexicographie**] et [**Syntaxe**], en vous aidant des exemples qui apparaissent dans [**ExempleSansObjet**] et [**RendusIntermediaires**], etc.

Correction
Une réponse possible :

```
{
  int x ;
  print("Entrez un entier:");
  x = readInt();
  if (17 < x) {
    println("17 < ", x);
  } else {
    println(x, " <= 17");
  }
}
```

Exercice 2 (Votre premier programe IMA). Donner une traduction de votre programme de l'exo 1 dans l'assembleur IMA. On ne demande pas ici d'appliquer un algorithme de traduction (juste d'écrire ce programme directement en assembleur). Par contre, on demande d'insérer du code qui affiche le message "`Error: Input/Output error`" et termine la machine sur l'état d'erreur lorsque l'utilisateur ne saisit pas correctement un entier. [1]

Pour écrire ce programme, vous devez vous appuyer sur la spécification formelle de l'assembleur dans [**MachineAbstraite**], en vous aidant de l'exemple dans [**ExempleSansObjet**].

Vérifiez que ce programme fonctionne comme attendu en le testant avec `ima` (il faut avoir déjà suivi [**SeanceMachine**]). En particulier, vérifiez que vous pouvez obtenir les affichages
"`17 < 18`" "`17 <= 17`" "`16 <= 17`" "`Error: Input/Output error`"

Correction
Une réponse possible :

```
WSTR "Entrez un entier:"
RINT
BOV io_error
CMP #17, R1
BLE code.else
WSTR "17 < "
WINT
WNL
```

1. Dans l'idée, le compilateur `decac` doit réaliser une telle traduction à partir du programme de l'exo 1. Par contre, le code IMA produit par votre compilateur sera un peu plus complexe, car vous allez utiliser des procédés de compilation naïfs. En fait, c'est très compliqué d'écrire un compilateur *optimisé* qui génère du code aussi simple et efficace que possible. Dans le cadre du projet GL, on ne demande pas d'écrire un compilo *optimisé*, mais uniquement un compilo *correct* qui sait traiter le plus grand sous-ensemble possible du langage Deca. Et c'est déjà un *défi* dans le temps qui vous imparti.

```
HALT
code.else:
WINT
WSTR " <= 17"
WNL
HALT
io_error:
WSTR "Error: Input/Output error"
WNL
ERROR
```

Exercice 3 (Analyse lexicale). Pour chacun des programmes suivants, indiquez la suite de lexèmes que doit reconnaître par l'analyseur lexical d'après [**Lexicographie**] et en s'inspirant de [**ExempleSansObjet**]. Inutile de singer `text_lex` en donnant toutes les infos attachées aux lexèmes : suivre plutôt des conventions calquées sur [**Syntaxe**]. Par exemple, écrivez simplement `'println'` pour le mot-clef correspondant ou STRING↑`'bonjour'` pour le lexème `"bonjour"` du langage STRING.

1. le programme vide (sans bloc principal) :

```
// programme vide !
```

Correction
La suite de lexèmes est vide.

2. le programme avec un bloc principal qui ne contient aucune instruction :

```
{ }
```

Correction
Liste des lexèmes, un par ligne
'{'
'}'

3. le programme avec un bloc principal qui ne contient qu'une instruction qui ne fait rien :

```
{ ; }
```

Correction
Liste des lexèmes, un par ligne
'{'
';'
'}'

4. le programme qui dit bonjour :

```
{ println("Bonjour tout le monde !"); }
```

Correction
Liste des lexèmes, un par ligne

```
'{'
'println'
'('
STRING↑'Bonjour tout le monde !'
')'
';'
'}'
```

5. le programme qui affiche 17 (via une variable `x`) :

```
{ int x = 17; print(x); }
```

Correction
Liste des lexèmes, un par ligne

```
'{'
IDENT↑'int'
IDENT↑'x'
'='
INT↑'17'
';'
'print'
'('
IDENT↑'x'
')'
';'
'}'
```

6. le programme de l'exo 1.

Correction
Liste des lexèmes, un par ligne

```
'{'
IDENT↑'int'
IDENT↑'x'
';'
'print'
'('
STRING↑'Entrez un entier:'
')'
';'
IDENT↑'x'
'='
'readInt'
'('
')'
';'
'if'
'('
INT↑'17'
'<'
IDENT↑'x'
')'
'{'
```

```
'println'
'('
STRING↑'17 < '
','
IDENT↑'x'
')'
';'
'}'
'else'
'{'
'println'
'('
IDENT↑'x'
','
STRING↑' <= 17'
')'
';'
'}'
'}'
```

Exercice 4 (Analyse syntaxique). Reprendre les programmes de l'exo 3, et donner l'arbre abstrait que doit retourner l'analyseur syntaxique, d'après la spécification donnée en [**Decompilation**]. Ce document décrit une traduction du langage **PROGRAM** de la [**SyntaxeAbstraite**] vers le langage **prog** de [**Syntaxe**]. Vous allez en fait ici devoir appliquer la traduction *inverse* pour passer de **prog** à **PROGRAM** sur ces programmes. [2] Là encore, on demande d'écrire cet arbre en appliquant les conventions de la grammaire [**SyntaxeAbstraite**], plutôt que du programme `text_synt` apparaissant dans [**ExempleSansObjet**] (voir l'exemple de la figure 1 page 60).

Correction

1. le programme vide :
   `Program [ [] EmptyMain ]`
2. le programme avec un bloc principal qui ne contient aucune instruction :
   ```
   Program [
      []
      Main [ [] [] ]
   ]
   ```
3. le programme avec un bloc principal qui ne contient qu'une instruction qui ne fait rien :
   ```
   Program [
      []
      Main [
         []
         [ NoOperation ]
      ]
   ]
   ```

2. Implémenter cette traduction *inverse* constitue en fait la *majeure* partie de ce que vous devez réaliser pour l'étape A du compilateur. En effet, l'essentiel de votre travail en étape A consiste à étendre la BNF de [**Syntaxe**] en une BNF attribuée [**ANTLR**] qui réalise cette traduction. Dans votre implémentation, les arbres abstraits seront des *objets Java*, codés dans le style de [**ConventionsCodage**], qui dérivent d'une hiérarchie de classes calquée sur la BNF de [**SyntaxeAbstraite**].

4. le programme qui dit bonjour :

```
Program [
   []
   Main [
      []
      [ Println↑false [ StringLiteral↑'Bonjour tout le monde !' ]  ]
   ]
]
```

5. le programme qui affiche 17 (via une variable x) :

```
Program [
   []
   Main [
      [ DeclVar [
          Identifier↑'int'
          Identifier↑'x'
          Initialization[ IntLiteral↑17 ]
        ]
      ]
      [ Print↑false [ Identifier↑'x' ]  ]
   ]
]
```

6. le programme de l'exo 1.

```
Program [
   []
   Main [
      [ DeclVar [
          Identifier↑'int'
          Identifier↑'x'
          NoInitialization
        ]
      ]
      [
        Print↑false [ StringLiteral↑'Entrez un entier:' ]
        Assign [ Identifier↑'x' ReadInt ]
        IfThenElse [
          Lower [ IntLiteral↑17 Identifier↑'x' ]
          [ Println↑false [ StringLiteral↑'17 < '  Identifier↑'x' ] ]
          [ Println↑false [ Identifier↑'x'  StringLiteral↑' <= 17' ] ]
        ]
      ]
   ]
]
```

Exercice 5 (Analyse contextuelle). Le document [**SyntaxeContextuelle**] définit la vérification statique (contextuelle) du compilateur pour l'ensemble du langage Deca (qui a lieu en étape B) : tout arbre abstrait accepté par ce vérificateur doit être compilable en un programme assembleur. Une tâche essentielle de ce vérificateur est de trouver la *signification* des identificateurs du programme : il relie l'*utilisation* de tout identificateur à sa *définition* (ou *déclaration*) pour en vérifier la cohérence. Formellement, il exprime cette signification des identificateurs à l'aide d'*environnements*, qui associent à chaque *symbole* d'identificateur une *définition* (dans l'implémentation Java, un tel environnement est par exemple une sorte de table de hachage, associant

chaque objet de type `Symbol` à un objet qui représente une définition). Pour préparer la génération de code (en étape C), l'étape B va aussi mémoriser en partie les environnements dans l'arbre abstrait (pendant la vérification contextuelle) : c'est la notion d'[**ArbreEnrichi**]. Autrement dit, le document [**ArbreEnrichi**] (à l'aide de [**Exemple**]) indique comment on doit enrichir l'arbre abstrait, à partir des vérifications décrites par [**SyntaxeContextuelle**] en vue de l'étape C.

Si on se limite au Deca sans objet, la vérification contextuelle est relativement élémentaire. Elle est décrite par *une grammaire attribuée* – appelée passe 3 du vérificateur final (avec objet). Elle commence donc à la régle (3.1) (p. 75) dans lequel l'attribut *env_types* hérité correspond à *l'environnement des types prédéfinis sans objet*. Autrement dit, c'est env_types_predef (p. 69) sans la définition de l'identificateur Object. Cet environnement ne contient que les types (de base) prédéfinis. Et l'utilisateur du Deca sans objet n'a en fait pas la possibilité d'introduire d'autres types que ceux prédéfinis (au contraire, chaque définition de classe va étendre l'environnement des types, comme cela est spécifié dans les passes 1 et 2 de la vérification contextuelle). Ainsi, dans toutes les règles de la passe 3, *env_types* est la version sans Object de env_types_predef. De plus, en Deca sans objet, les identificateurs n'ont que deux natures possibles : soit des types de base (nature notée type dans la doc), soit des variables (nature notée var). Précisons que la vérification contextuelle du Deca sans objet inclut aussi les règles (0.1) et (0.2) (p. 72) qui concernent le traitement des identificateurs, qui est commun à toutes les passes du vérificateur.

Ainsi, on peut spécialiser les règles de [**SyntaxeContextuelle**] pour traiter uniquement le Deca sans-objet, et c'est ce que vous allez devoir implémenter au démarrage du projet. Par exemple, dans le cas du Deca sans-objet, la règle (3.2) est toujours valide (la liste de définition de classes est vide). Et la règle (3.4) dit qu'on peut passer directement au traitement fait par la règle (3.18) avec *env_exp_sup* et *env_exp* des environnements vides, *class* qui vaut 0 et *return* qui représente le type void.

▷ **Question 1.** On considère le programme Deca "`{ print(x); }`" correspondant à l'arbre abstrait ci-dessous.

```
Program [
   []
   Main [
      [ ]
      [ Print↑false [ Identifier↑'x' ]  ]
   ]
]
```

Expliquez pourquoi ce programme est rejeté par le vérificateur contextuel (en indiquant les calculs effectués sur la grammaire attribuée).

Correction
On commence par enchaîner (3.1), (3.4), (3.18). La règle (3.16) est valide car la liste de déclarations de variable est vide : elle synthétise un environnement *env_exp_r* vide. On rentre donc dans (3.19) avec un *env_exp* vide, puis dans (3.21) (en validant (3.26)), puis dans (3.30), puis dans (3.31), puis dans (3.34), puis dans (3.64). Là, en Deca sans-objet, le traitement du non-terminal **lvalue_ident** passe nécessairement par la règle (3.69), en invoquant la règle (0.1). Et c'est ici que la vérification échoue. En effet, dans la règle (0.1), l'attribut env_exp est toujours vide et l'attribut *name* vaut `'x'`. Donc la condition échoue car *env_exp*(*name*) n'est pas défini (cf. explication page 70).

▷ **Question 2.** On considère le programme "`{ int x="Bonjour"; }`" correspondant à l'arbre abstrait ci-dessous.

```
Program [
   []
```

```
   Main [
      [ DeclVar [
          Identifier↑'int'
          Identifier↑'x'
          Initialization[ StringLiteral↑'Bonjour']
        ]
      ]
      [  ]
   ]
]
```

Expliquez pourquoi ce programme est rejeté par le vérificateur contextuel (en indiquant les calculs effectués sur la grammaire attribuée).

Correction
On commence par enchaîner (3.1), (3.4), (3.18). On rentre dans (3.16), puis dans (3.17). La vérification sur le non-terminal **type** passe par la règle (0.2) qui réussit en sythétisant la valeur int pour l'attribut *type*. Ensuite, au niveau de la règle (3.17), on vérifie le non-terminal **initialization**. On rentre alors dans la règle (3.8) puis (3.28). On valide alors les règles (3.38) et (3.46) qui synthétisent la valeur string. On remonte donc au niveau de (3.28) où l'on teste la condition

$$\text{assign\_compatible}(\text{env\_types\_predef}, \text{int}, \text{string})$$

Dans le cas du sans-objet, assign_compatible(env, T_1, T_2), défini page 67, vaut

$$T_1 = T_2 \quad \text{ou} \quad (T_1 = \text{float} \text{ et } T_2 = \text{int})$$

Donc, elle n'est pas vérifiée ici.

▷ **Question 3.** On considère le programme "`{ void x; }`" correspondant à l'arbre abstrait ci-dessous.

```
Program [
   []
   Main [
      [ DeclVar [
          Identifier↑'void'
          Identifier↑'x'
          NoInitialization
        ]
      ]
      [  ]
   ]
]
```

Ce programme est-il accepté ou rejeté par le vérificateur contextuel ? Justifier la réponse.

Correction
On commence par enchaîner (3.1), (3.4), (3.18). On rentre dans (3.16), puis dans (3.17). La vérification sur le non-terminal **type** passe par la règle (0.2) qui réussit en sythétisant la valeur void pour l'attribut *type*. Ensuite, au niveau de la règle (3.17), on vérifie le non-terminal **initialization**. On valide alors dans la règle (3.9) (sans échec possible). En revenant dans (3.17), la condition

"*type* ≠ void" échoue. Ce programme est donc rejeté : on ne peut pas déclarer des variables de type `void` en Deca.

▷ **Question 4.** On considère le programme "`{ int int = 17; print(int); }`" correspondant à l'arbre abstrait ci-dessous.

```
Program [
   []
   Main [
      [ DeclVar [
          Identifier↑'int'
          Identifier↑'int'
          Initialization[ IntLiteral↑17 ]
        ]
      ]
      [ Print↑false [ Identifier↑'int' ]  ]
   ]
]
```

Ce programme est-il accepté ou rejeté par le vérificateur contextuel ? Justifier la réponse.

Correction
On commence par enchaîner (3.1), (3.4), (3.18). On valide la règle (3.16) – et toutes les sous-règles impliquées : (3.17), (3.8), (3.28), (3.38) et (3.44) – en synthétisant "$env\_exp\_r = \{\text{int} \mapsto (\underline{\text{var}}, \underline{\text{int}})\}$". Ensuite, on rentre donc dans (3.19), puis dans (3.21) – en validant (3.26) – puis dans (3.30), puis dans (3.31), puis dans (3.34), puis dans (3.64), puis (3.69), puis règle (0.1) qui réussit en synthétisant "$def = (\underline{\text{var}}, \underline{\text{int}})$". On remonte alors dans (3.69) qui réussit en synthétisant "$type = \underline{\text{int}}$". Puis, on remonte dans (3.64), (3.34) et (3.31). La condition dans (3.31) est bien validée, et toutes les règles au-dessus aussi. Ce programme est accepté (et est donc compilable par `decac`) : à l'exécution, il affiche 17. Donc, le langage Deca accepte que l'identificateur "`int`" représente autre chose que le type int (sous certaines conditions), même si ce n'est sans doute pas une bonne utilisation du langage.

▷ **Question 5.** On considère le programme "`{ x x = 17; print(x); }`" correspondant à l'arbre abstrait ci-dessous.

```
Program [
   []
   Main [
      [ DeclVar [
          Identifier↑'x'
          Identifier↑'x'
          Initialization[ IntLiteral↑17 ]
        ]
      ]
      [ Print↑false [ Identifier↑'x' ]  ]
   ]
]
```

Ce programme est-il accepté ou rejeté par le vérificateur contextuel ? Justifier la réponse.

Correction
On commence par enchaîner (3.1), (3.4), (3.18). On rentre dans (3.16), puis dans (3.17). La vérification sur le non-terminal **type** passe par la règle (0.2) qui rejette le programme. En effet env_types_predef(x) n'est pas défini.