26/06/2020

TP Compilation - HEPIAL

Cours de Techniques de compilation

Ottavio Buonomo & Jean-Daniel Kuenzi
HEPIA

TP COMPILATION - HEPIAL

Introduction

Lors du cours de Techniques de compilation de 2ème année à l'Hepia, il nous a été demandé de créer notre propre compilateur avec notre propre langage de programmation afin de mettre en pratique les notions vues en cours. Le langage que nous avons conçu s'appelle HEPIAL. Ce travail était à réaliser par groupe de deux et il comportait l'écriture du code, un rapport et une présentation.

HEPIAL

Le langage HEPIAL est assez brut et composé essentiellement de trois parties, la déclaration du programme, la déclaration des variables et les instructions. Ces dernières sont des affectations, des conditions, des boucles des écritures sur la console ou des lectures d'entrées utilisateur.

Uniquement deux types de variable sont présent dans le langage HEPIAL, il s'agit du type entier (TypeInteger) et du type booléen (TypeBoolean) mais il est quand même possible d'écrire sur la console une chaîne de caractères.

Lors de la déclaration d'une constante, le type et la valeur de la constante seront figé et il ne sera pas possible de la réaffecter. Aussi il est impossible de déclarer des variables avec le même nom même si elles n'ont pas le même type.

Lors de la lecture d'une entrée utilisateur, il est uniquement possible d'affecter une valeur entière.

HEPIAL	Description
programme **<nom>**	Initialise le programme avec son nom
debutprg	Signal le début du programme
finprg	Signal la fin du programme
constante **<type>** **<nom>** = **<valeur>**;	Initialise une constante avec un type (entier ou booléen) un nom et sa valeur
entier **<nom>**;	Déclare une variable entière avec un nom
booleen **<nom>**;	Déclare une variable booléenne avec un nom
<variable> = **<variable>**|**<valeur>**|**<opérations>**;	Affecte une variable la valeur d'une autre variable, une valeur (vrai|faux pour les booléens, nombre pour les entiers) ou une opération (+, -, *, /, non, ~)

lire **<variable>**;	Lit une entrée clavier et la store dans une variable déclarée
ecrire **<variable>**;	Écrit dans le terminal la valeur de la variable
ecrire **<StringConst>**;	Écrit dans le terminal la chaîne de caractères
si **<condition>**	Signal le début d'un if avec une condition
si (**<condition>**) **et**|**ou** (**<condition>**) ...	Signal le début d'un if avec plusieurs conditions, les conditions doivent être entre **parenthèses** et reliées par les opérateurs binaires **et**|**ou**
alors **<instruction>**	Le/Les instruction(s) à effectuer si la condition est vraie
sinon **<instruction>**	La clause else du if (OPTIONNEL)
finsi	Signal la fin du if
tantque **<condition>**	Signal le début d'un while avec une condition
tantque (**<condition>**) **et**|**ou** (**<condition>**)	Signal le début d'un while avec plusieurs conditions, les conditions doivent être entre **parenthèses** et reliées par les opérateurs binaires **et**|**ou**
faire **<instruction>**	Le/Les instruction(s) à effectuer si la condition est vraie
fintantque	Signal la fin du while
pour **<variable>**	Signale le début du for avec un variable d'itération (déclaré auparavant)
allantde **<borne inférieure>**	Définit la borne inférieure du for
a **<borne supérieure>**	Définit la borne supérieure du for
faire **<instruction>**	Le/Les instruction(s) à effectuer
finpour	Signal la fin du for
<variable> >|**<**|**>=**|**<= <valeur>**|**<variable>**	Permet de comparer une variable de type entier à une autre variable ou un nombre. Cette opération retourne un booléen (vrai|faux). Sert surtout dans les conditions
<variable> <>|**== <variable>**|**<valeur>**	Permet de comparer une variable à une autre variable ou valeur du même type. Cette opération retourne un booléen (vrai|faux). Sert surtout dans les conditions

non **<variable>**|**<valeur>**	Opérateur booléen qui inverse une variable ou une valeur booléenne (non vrai => faux)
~ **<variable>**|**<valeur>**	Opérateur arithmétique qui inverse les bits d'un entier (~1001 => 0110)
<variable>|**<valeur>** **+|-|*|/** **<variable>**|**<valeur>**	Permet d'appliquer les opérateurs arithmétiques (addition, soustraction, multiplication et division) sur des variables ou valeurs entières

Exemple de code Hepial (Fibonacci)

```
programme Fibonacci
entier n1;
entier n2;
entier n3;
entier max;
entier i;
debutprg
    n1 = 1;
    n2 = 1;
    n3 = 1;
    ecrire "Veuillez entrer la valeur de la borne supérieure :";
    lire max;
    ecrire "--- Suite de Fibonacci ---";
    pour i allantde 0 a max faire
        ecrire n3;
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
    finpour
finprg
```

Figure 1 - Code de la suite de Fibonacci en Hepial

Analyse lexicale (FLEX)

L'analyse lexical est le début de la chaine de compilation. Elle permet de définir les mots du langage et vérifier que tous les mots écrits lors du codage sont bien connus par le langage. Cette analyse se fait avec le programme Jflex.
Le fichier source Jflex nous a été fournis pour ce TP.

Analyse syntaxique (CUP)

L'analyse syntaxique va nous permettre de créer nos classes et notre table des symboles nécessaire au bon fonctionnement de notre compilateur. Elle va aussi nous permettre d'effectuer une première vérification du code comme la vérification du respect de la syntaxe (d'où le nom analyse syntaxique) mais également de vérifier si on ne déclare pas plusieurs variables avec le même nom.
C'est notamment cette partie qui va générer notre arbre abstrait que l'on pourra parcourir plus tard avec le design pattern Visiteur afin de vérifier la bonne sémantique du code.
Pour l'architecture de l'arbre abstrait et la grammaire voir les annexes[1].

Création de la table des symboles

Durant l'analyse syntaxique, il est impératif de stocker les variables du programme HEPIAL dans une table qui les répertorie de façon unique. Nous avons donc utilisé la classe "HashMap" de Java qui nous a permis de stocker le nom de l'identificateur de la variable ainsi que son type (Integer ou Boolean). Nous avons créé la classe "TDS" qui se compose ainsi :

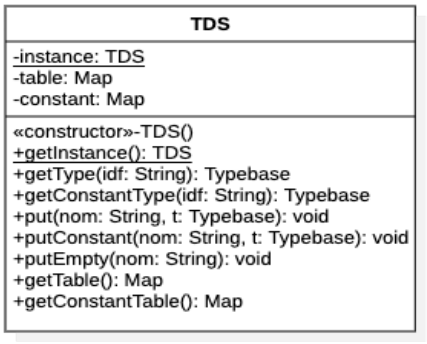


Figure 2 - Diagramme de classe de TDS

Cette dernière nous permet de garder les valeurs au sein de toute la compilation car il s'agit d'un singleton. Mais nous avons aussi créé une HashMap pour stocker les constantes ainsi nous avons une table des symboles contenant les variables et les constantes mais dans des tables séparées.
Dans le fichier "hepial.cup", nous créons une instance de TDS au moment où le fichier est exécuté et nous le remplissons au sein des déclarations de variables et déclaration du programme.

Analyse sémantique

Une fois l'analyse lexicale et syntaxique complétées, puis la table des symboles créée, nous nous sommes attaqués à l'analyse sémantique du code en HEPIAL. Cette opération consiste en vérifiant les différentes opérations qui se font au niveau du code (ex : addition, condition, etc..). Plus particulièrement, nous avons vérifié deux choses, le type des variables et si les variables étaient

[1] Sur notre répo git

déclarées. Si l'une de ces deux opérations devait ne pas passer, une exception sera levée, la compilation s'arrête et un message d'erreur sera affiché indiquant le problème.
L'analyseur sémantique implémente la classe ASTNode afin de pouvoir utiliser le pattern Visiteur et ainsi contrôler l'arbre abstrait. L'utilisation du design pattern est très intéressants cars elle nous permet de parcourir tout l'arbre abstrait jusqu'au feuilles sans avoir besoin de développer une méthode de vérification pour chaque classe.
Aussi en visitant par exemple une Expression, nous allons aussi pouvoir vérifier tous les attributs de l'expression et ainsi descendre jusqu'au bas de l'arbre abstrait. Par exemple dans le cas où l'on visite une instruction d'affectation et d'addition (ex: a = a + c):

Structure de l'instruction

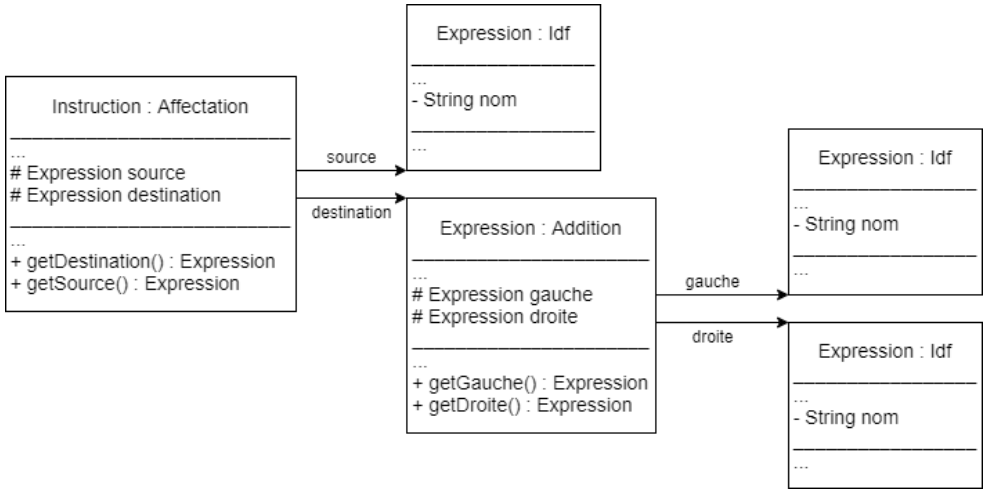


Figure 3 – Structure de l'instruction

Séquence du visiteur

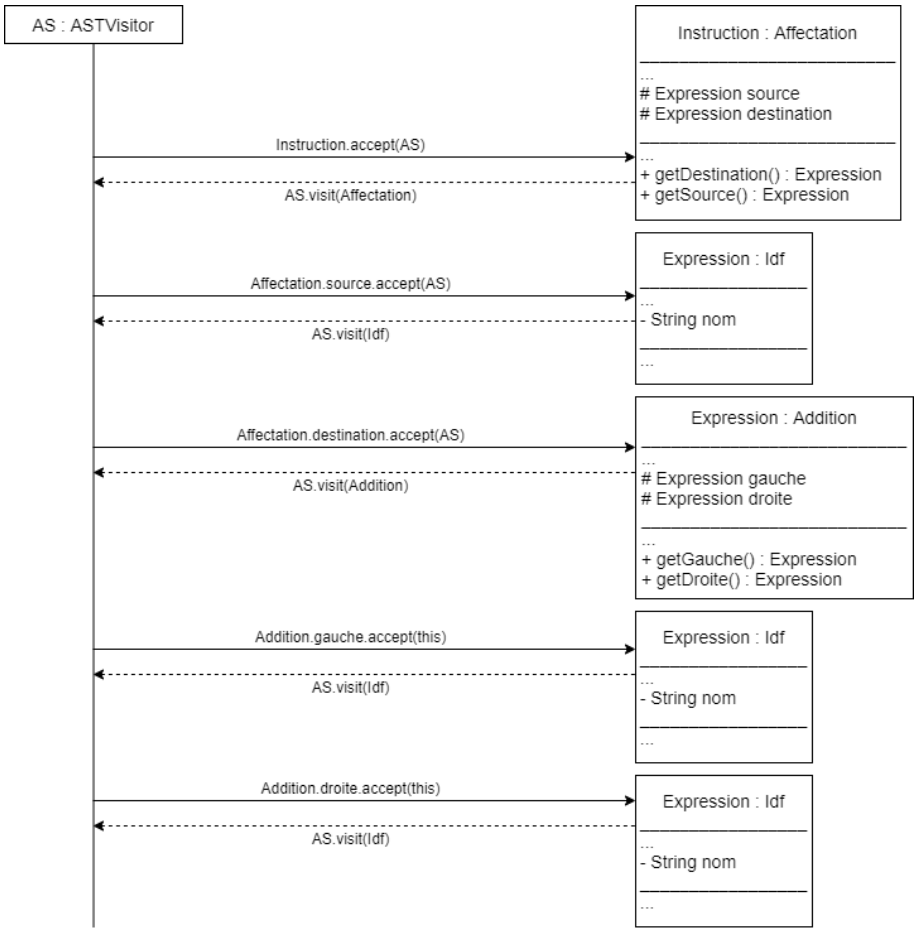


Figure 4 - Diagramme de ASTVisitor

Ce schéma représente le parcours du visiteur depuis l'analyseur sémantique. Tout d'abord il va visiter notre instruction qui est de type Affectation, puis en visitant la source, le visiteur va s'assurer que l'Idf est bien dans la table des symboles (initialisé), ensuite il va visiter la destination qui est une Addition, il va visiter les Expression gauche et droite qui sont des Idf, et ainsi nous allons visiter les noeuds de notre arbre jusqu'au feuilles (Idf) pour une addition.

Contrôle des instructions

Affectation

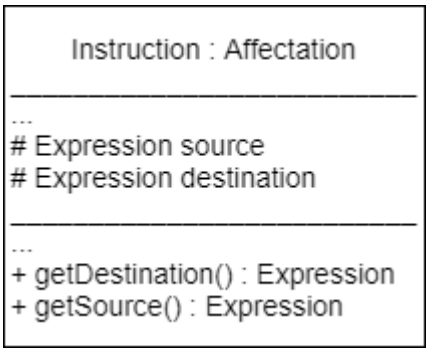


Figure 5 - Diagramme UML d'une affectation

Lors d'une affectation, la première chose que l'on contrôle est si la destination et la source sont correctes grâce au pattern visiteurs qui va visiter la source et la destination. Ensuite nous nous assurons que la destination ne soit pas une constante. Car effectivement nous ne pouvons pas affecter une constante. Puis pour finir, nous vérifions que la source et la destination soient du même type (TypeInteger ou TypeBooleen).

```
@Override
public Object visit(Affectation node) {
    Object gauche = node.getDestination().accept(this);
    Object droite = node.getSource().accept(this);
    if (this.constant.containsKey(((Idf)node.getDestination()).getNom())) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Affectation : Impossible d'affecter une constante à la ligne "
            + node.getLine()
        );
    }
    if (gauche.getClass() != droite.getClass()) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Affectation : Impossible d'affecter le type"
            + gauche.getClass() + " = " + droite.getClass() + " à la ligne " + node.getLine()
        );
    }
    return null;
}
```

Figure 6 - Analyse sémentique d'une affectation

Instruction conditionnelle

Une instruction conditionnelle est composée de 4 attributs :

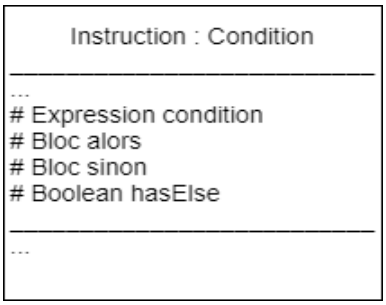


Figure 7 - Diagramme UML d'une instruction conditionnelle

Pour ces 4 attributs il suffit tout simplement de les visiter afin de savoir si l'expression et le(s) bloc(s) d'instructions est/sont correcte(s).

```java
@Override
public Object visit(Condition node) {
    node.getCondition().accept(this);
    node.getThenInstruction().accept(this);
    if (node.hasElse()) {
        node.getElseInstruction().accept(this);
    }
    return null;
}
```

Figure 8 - Méthode "visit" d'une condition

Contrôle des opérations binaires

Addition

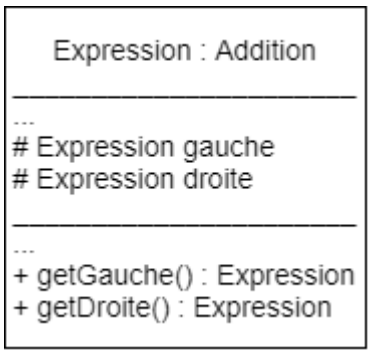


Figure 9 - Diagramme de l'addition

Une addition prend 2 Expression de TypeInteger et retourne un TypeInteger. Il faut donc d'abord vérifier que les Idf des 2 expressions soient bien dans la table des symboles (Initialisées) et que leurs type sont bien TypeInteger.

```java
@Override
public Object visit(Addition node) {
    Object gauche = node.getGauche().accept(this);
    Object droite = node.getDroite().accept(this);
    if (gauche.getClass() != TypeInteger.class || droite.getClass() != TypeInteger.class) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Addition : Impossible d'aditionner "
            + gauche.getClass() + " + " + droite.getClass() + " à la ligne " + node.getLine()
        );
    }
    return new TypeInteger("entier", "", 0, 0);
}
```

Figure 10 - Méthode "visit" d'une addition

Les autres opérateurs arithmétiques (-, *, /) sont contrôlés de la même manière.

Supérieur-égal

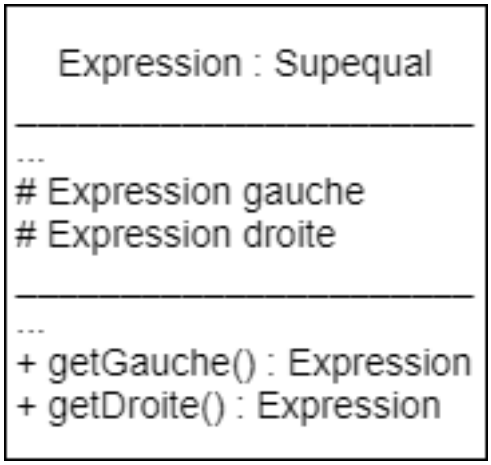


Figure 11 - Diagramme de supérieur égal

Un supérieur-égal est une comparaison qui compare 2 Expression et retourne un TypeBooleen. Il faut donc d'abord vérifier que les Idf des 2 expressions soient bien dans la table des symboles (Initialisées) et que leurs types sont bien TypeInteger.

```
@Override
public Object visit(Supequal node) {
    Object gauche = node.getGauche().accept(this);
    Object droite = node.getDroite().accept(this);
    if (gauche.getClass() != TypeInteger.class || droite.getClass() != TypeInteger.class) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Supequal : Impossible de comparer le type "
            + gauche.getClass() + " >= " + droite.getClass() + " à la ligne " + node.getLine()
        );
    }
    return new TypeBooleen("booleen", "", 0, 0);
}
```

Figure 12 - Méthode "visit" de supérieur égal

Les autres opérateurs de comparaison (>, <, <=) sont contrôlés de la même manière.

Double égal

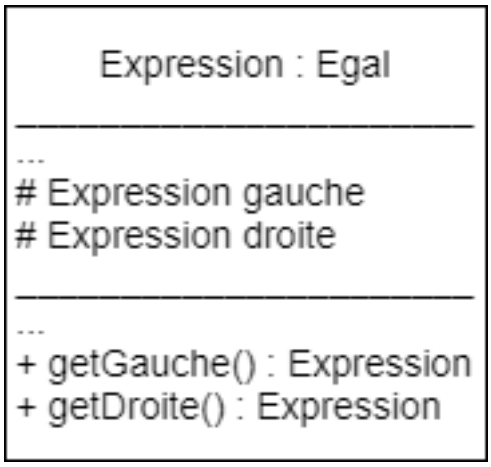


Figure 13 - Diagramme de égal

Un double égal est une comparaison qui compare 2 expressions et retourne un TypeBooleen. Il faut donc d'abord vérifier que les Idf des 2 Expression soient bien dans la table des symboles (Initialisées) et que leurs types sont bien correspondant (ont le même type).

```
@Override
public Object visit(Egal node) {
    Object gauche = node.getGauche().accept(this);
    Object droite = node.getDroite().accept(this);
    if (gauche.getClass() != droite.getClass()) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Egal : Impossible de comparer "
            + gauche.getClass() + " == " + droite.getClass() + " à la ligne " + node.getLine()
        );
    }
    return new TypeBooleen("booleen", "", 0, 0);
}
```

Figure 14 - Méthode "visit" de l'égalité

L'opérateur différent (<>) est contrôlé de la même manière.

Contrôle des opérations unaire

Tilda

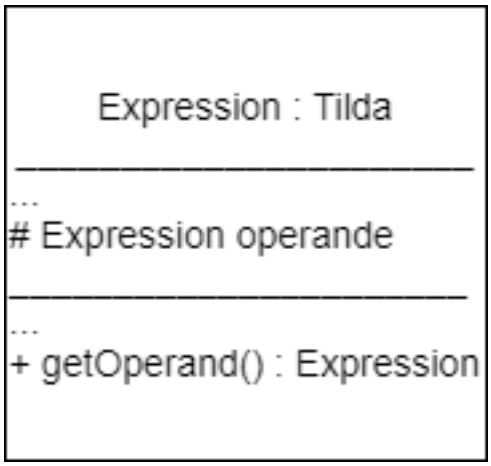


Figure 15 - Diagramme de tilda

L'opérateur tilda (~) inverse les bits d'un TypeInteger et retourne un nouveau TypeInteger. Il faut donc vérifier que l'Idf est bien initialisé et bien TypeInteger.

```
@Override
public Object visit(Tilda node) {
    Object op = node.getOperand().accept(this);
    if (op.getClass() != TypeInteger.class) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Tilda : L'opérateur ~ ne s'applique pas sur le type"
            + op.getClass() + " à la ligne " + node.getLine()
        );
    }
    return new TypeBooleen("booleen", "", 0, 0);
}
```

Figure 16 - Méthode "visit" de tilda

Non

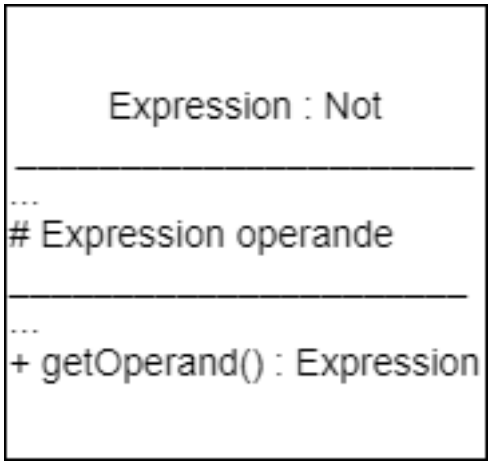


Figure 17 - Diagramme du Not

L'opérateur non prends un TypeBooleen et retourne l'inverse de celui-ci (ex : non vrai => return false). Il faut donc vérifier que l'Idf est bien initialisé et bien TypeBooléen.

```java
@Override
public Object visit(Not node) {
    Object op = node.getOperand().accept(this);
    if (op.getClass() != TypeBooleen.class) {
        throw new RuntimeException(
            "Erreur dans l'analyseur sémantique Not : L'opérateur non ne s'applique pas sur le type"
            + op.getClass() + " à la ligne " + node.getLine()
        );
    }
    return new TypeBooleen("booleen", "", 0, 0);
}
```

Figure 18 - Méthode "visit" de not

Contrôle des feuilles de l'arbre abstrait

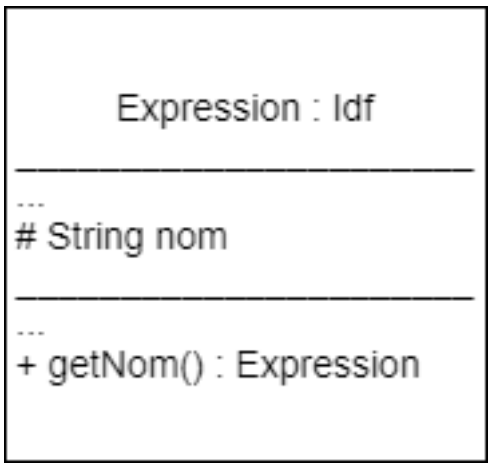


Figure 19 - Diagramme de l'Idf

Les feuilles de l'arbre se trouvent être de type Idf (Identificateur), donc pour contrôler les feuilles de l'arbre, il suffit juste de contrôler si l'Idf en question se trouve dans la table des symboles ou dans la table des constantes.
Aussi, le nom du programme est un Idf qui se trouve dans la table des symboles mais sans type. Il faut donc avoir une exception pour ce cas-là lors de la vérification de l'Idf (la vérification se fait sur le type de l'Idf).

```java
@Override
public Object visit(Idf node) {
    Typebase monType = this.tds.get(node.getNom());
    if (monType == null) {
        boolean flag = this.tds.containsKey(node.getNom());
        if (!flag) {
            monType = this.constant.get(node.getNom());
            if (monType == null) {
                throw new RuntimeException(
                    "Erreur dans l'analyseur sémantique Idf : Identificateur introuvable à la ligne "
                    + node.getLine()
                );
            }
        }
    }

    if (monType instanceof TypeInteger) {
        return ((TypeInteger) monType);
    }
    if (monType instanceof TypeBooleen) {
        return ((TypeBooleen) monType);
    }
    return null;
}
```

Figure 20 - Méthode "visit" d'un ifd

- Tout d'abord nous regardons si l'Idf est présent dans la table des symboles.
- Si oui, la méthode va retourner le type de l'Idf.
- Si non, nous regardons si le nom de l'idf est présent dans la table des symboles.
    - Si oui, c'est le nom du programme et le visiteur retourne null.
    - Si non, on regarde si l'Idf est dans la table des constantes et s'il il l'est on retourne son type. Si non, une exception est levée.

Production du byte code

Fonctionnement de Jasmin

Lors de ce travail, nous avons utilisé un outil appelé "Jasmin" permettant de transformer le code Java en code assembleur (Jasmin) qui peut être converti ensuite en byte-code exécutable par la JVM (Java Virtual Machine). Le fonctionnement de Jasmin se fait sur le modèle de pile (stack). Les instructions sont donc stockées sur la pile puis ensuite exécutées. Ce qu'il faut savoir est que la JVM offre l'accès aux librairies Java nous permettant donc d'afficher sur la console ou de lire des entrées utilisateur. Lors de la production du code assembleur, nous avons utilisé la classe "PrintWriter" qui nous permet d'écrire dans un fichier depuis Java.

Déclarations

Programme

Pour la création d'un programme HEPIAL, la déclaration du programme est nécessaire (un peu comme une classe en Java). Elle sera composée du mot clé "programme" suivi de son nom ainsi que les mots

“debutprg” et “finprg” qui déterminent respectivement le début et la fin du programme. Entre les mots “debutprg” et “programme” se fera la déclaration de variables qui sera expliquée dans la section suivante.
Une fois les structures globales complétée, il est nécessaire d’ajouter d’autres informations pour créer une classe et une méthode “main” qui sera le point d’entrée du programme. Les lignes à introduire sont les suivantes :

```
.class public <NOM_DU_PROGRAMME>
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 20000
.limit locals 100
```

Figure 21 - Code Jasmin pour l'entrée du programme

Puis la fin du programme s’écrira avec les deux lignes suivantes. Entre ces deux parties se trouveront les instruction assembleur.

```
return
.end method
```

Figure 22 - Code Jasmin pour la fin d'un programme

Variables

Comme dans tout langage de programmation, la déclaration du programme ou de variables est nécessaire et HEPIAL n’est pas une exception. Les variables que nous allons déclarer dans notre langage seront uniquement de deux types : entier et booléen. Pour écrire de l’assembleur Jasmin qui déclare des variables nous avons utilisé l’instruction “.var” qui doit être suivie de son numéro (pour l’identifier) du mot clé “is”, son nom (donné dans le programme en Java) et de son type. Ce dernier s’écrit avec un “I” si la variable est entière ou avec un “Z” si la variable est booléenne.
Voici comment s’écrit la déclaration de variables en Jasmin :

```
.var 0 is i I
.var 1 is d Z
```

Figure 23 - Code Jasmin pour la déclaration de variable

Instructions

Affectation

Une fois la variable déclarée, il faut lui affecter une valeur. Cela se fait avec l’instruction Jasmin “istore” qui va prendre la valeur qui identifie la variable (donnée au moment de la déclaration). La commande va la prendre en considération la valeur qui est au-dessus dans la pile pour l’affecter à la variable définie.

Chargement

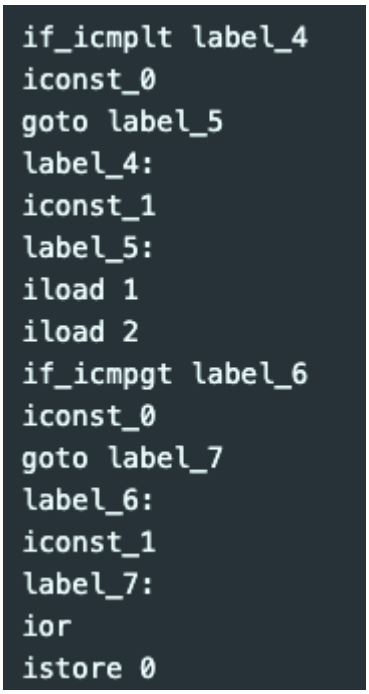
Si l'on souhaite accéder à une variable que nous avons dans notre programme, il suffit d'utiliser l'instruction Jasmin "iload" suivie de l'identifiant de la variable que nous voulons lire.

Arithmétique et opérations unaires

Pour nous permettre de compiler et d'utiliser le langage HEPIAL, nous avons dû implémenter les instructions au sein du langage. Ces dernières se décomposent en deux catégories : unaire (un seul opérande) et binaire (deux opérandes). Nous devons donc convertir les instructions Java en code assembleur Jasmin. Pour se faire nous avons utilisé cette table de conversion.

HEPIAL	**Jasmin**
Instructions Binaires	
+ (Addition)	iadd
- (Soustraction)	isub
* (Multiplication)	imul
/ (Division)	idiv
ou	ior
et	iand
Instructions Unaires	
non	ifeq
~ (Tilda)	ixor

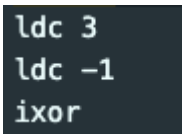
Tout d'abord, pour les opérations arithmétiques, nous avons commencé par effectuer un parcours de l'expression de gauche et un parcours de l'expression de droite. Cela aura pour but de mettre les valeurs sur la pile de Jasmin. S'il s'agit d'un nombre l'instruction "ldc" sera utilisée ou l'instruction "iload" permettra de charger la valeur d'une variable.
Voici un exemple de la conversion d'une addition en Jasmin :
Ensuite, nous avons continué avec les opérateurs "et" et "ou". Ces derniers fonctionnent uniquement si les expressions de gauche et droite sont parenthésées et si dans les parenthèses se trouvent des opérations qui retournent un booléen. On va donc caster l'opération de gauche et de droite en classe "Parenthese" et vérifier s'il s'agit d'une relation avec un résultat binaire. Si c'est le cas, alors on parcourt les opérations et on ajoute à la pile les résultats qui seront de type booléen. Puis nous ajoutons sur la pile l'opérateur que nous souhaitons.
Voici un exemple de code Jasmin permettant d'effectuer un "ou" :



```
if_icmplt label_4
iconst_0
goto label_5
label_4:
iconst_1
label_5:
iload 1
iload 2
if_icmpgt label_6
iconst_0
goto label_7
label_6:
iconst_1
label_7:
ior
istore 0
```

Figure 24 - Code Jasmin effectuant un "ou"

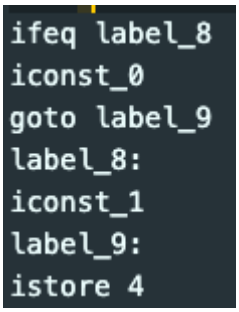
Pour terminer avec cette partie, nous avons converti les opérations unaires en Jasmin. Par rapport aux instructions précédentes, les deux instructions unaires se codent de façon différente l'une de l'autre.
Pour l'opération "tilda", il suffit de parcourir l'opérande et d'ajouter sur la pile la valeur "-1" ainsi de pouvoir faire un xor entre la valeur de l'opérande et celle-ci. On finira par ajouter la ligne "ixor" de Jasmin pour compléter notre opération.



```
ldc 3
ldc -1
ixor
```

Figure 25 - Code Jasmin pour l'opération "tilda"

Pour l'opération "non", nous avons parcouru l'opérande et utilisé le commande "ifeq" de Jasmin qui va mettre une valeur booléenne sur la pile en fonction du résultat.



```
ifeq label_8
iconst_0
goto label_9
label_8:
iconst_1
label_9:
istore 4
```

Figure 26 - Code Jasmin effectuant un "non"

Conditions

Après les instructions, nous avons implémenté les conditions. Cela se fait toujours avec des commandes en assembleur Jasmin qui vont retourner, cette fois, des booléens (0 ou 1) en fonction du résultat d'une comparaison. De même que les instructions, les conditions se font à l'aide de la pile. On va donc récupérer les dernières valeurs sur la pile pour les comparer entre elles. Les instructions pour les types de comparaison sont converties à l'aide du tableau suivant.

HEPIAL	Jasmin
== (Egal)	if_cmpeq
> (Supérieur)	if_cmpgt
< (Inférieur)	if_cmplt
<= (Inférieur égal)	if_cmple
>= (Supérieur égal)	if_cmpge
<> (Différent)	if_cmpne

Pour ensuite appliquer les conditions et les instructions qui doivent être exécutées, nous avons utilisée l'opérateur "ifeq". En effet, les instructions du tableau ci-dessus mettent sur la pile une valeur booléenne comme dit précédemment, on peut donc utiliser ce dernier opérateur pour écrire notre condition.

```
iload 1
ldc 3
if_icmpeq label_8
iconst_0
goto label_9
label_8:
iconst_1
label_9:
ifeq label_10
iload 1
ldc 1
iadd
istore 1
goto label_11
label_10:
label_11:
```

Figure 27 – Code Jasmin effectuant une condition "égale"

Boucles

Le langage HEPIAL contient aussi des boules qui permettent de répéter des instructions, comme par exemple la boucle "pour" (for) ou "tantque" (while).
La boucle "pour" utilise une variable qui va compter le nombre de tours que la boucle va faire et ainsi nous permettre de vérifier la condition (cette variable pourrait aussi servir comme indice d'un tableau mais cette fonctionnalité n'a pas été implémenté dans cette version). La boucle se définit en langage

HEPIAL comme “pour i allantde 0 à 5 faire … finpour”. Cette instruction va donc devoir être traduite en Jasmin, ce qui donnera quelque chose de ce style :

```
label_8:
ldc 5
iload 0
if_icmplt label_9
iload 2
ldc 1
iadd
istore 2
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 2
invokevirtual java/io/PrintStream/println(I)V
iload 0
ldc 1
iadd
istore 0
goto label_8
label_9:
```

Figure 28 - Code Jasmin effectuant une boucle "pour"

La boucle “tantque” va uniquement répéter une instruction tant que la condition ne sera plus vérifiée. Elle se décrit en HEPIAL comme “tantque … faire … fintantque” qui se traduit en Jasmin par :

```
label_10:
iload 4
ldc 1
if_icmpeq label_12
iconst_0
goto label_13
label_12:
iconst_1
label_13:
ifeq label_14
ldc 0
istore 4
goto label_10
label_14:
```

Figure 29 - Code Jasmin effectuant une boucle "tant que"

Écrire dans la console

Du fait que nous produisons du byte-code directement interprétable par la JVM, il est possible d’utiliser les librairies Java pour afficher du texte dans la console, que ce soit une phrase ou le résultat d’une opération arithmétique. Cependant, il faut faire attention aux types des variables, c’est pourquoi nous avons testé le type avant d’écrire l’instruction Jasmin pour l’écriture qui se présente comme ceci :

```
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 1
invokevirtual java/io/PrintStream/println(I)V
```

Figure 30 - Code Jasmin permettant d'écrire dans la console

En effet, il suffit de changer la valeur dans les parenthèses de “println” et mettre “Z” si on souhaite afficher un booléen ou “Ljava/lang/String;” pour afficher une chaîne de caractères.
Nous pouvons donc préciser que le “getstatic” permet de charger sur la pile la librairie système permettant d’écrire sur la sortie standard.

Lire une entrée utilisateur

Pour lire une entrée de l’utilisateur sur l’entrée standard de la console, il faut utiliser plusieurs librairies de Java. Avec Jasmin cela se traduit par l’invocation de plusieurs librairies au sein de la JVM. Plus précisément, nous avons utilisé la librairie “Scanner” de Java. Une fois l’entrée saisie, la valeur sera stockée dans une variable (à noter que nous acceptons uniquement les valeurs entières).
Voici le code Jasmin permettant d’effectuer ces opérations :

```
new java/util/Scanner
dup
getstatic java/lang/System/in Ljava/io/InputStream;
invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
invokevirtual java/util/Scanner/nextInt()I
istore 0
```

Figure 31 - Code Jasmin permettant de lire dans la console

Conclusion

Nous arrivons donc à la fin de ce TP. Nous avons vu plusieurs parties de la compilation tout au long, en commençant par l’analyse lexicale, puis l’analyse syntaxique, ensuite l’analyse sémantique et enfin la génération du byte-code. Cela nous a permis de mettre en application les notions vues au cours de techniques de compilation. Il s’agissait d’une première pour nous car nous n’avions jamais créé un langage de programmation auparavant et cela a rendu le TP très intéressant. Cependant, il a été difficile pour nous de comprendre dès le début toutes les parties à implémenter et l’utilisation du design pattern “Visitor” ne nous a pas aidé. Malgré cela, après avoir posé quelques questions à l’assistant du cours, nous sommes parvenus à comprendre tout le fonctionnement et nous l’avons implémenté.