Remerciements

Nous tenons à remercier nos professeurs encadrants pour l'aide qu'ils nous ont apportée tout au long de notre projet ;

M. BOUQUET, pour les conseils qu'il nous a apportés tout au long du déroulement du développement et M. JULLIAND pour l'attention dont il a fait preuve à l'egard de notre groupe, notamment au cours des réunions sur l'état d'avancement du projet.

Table des matières

Introduction

Pourquoi écrire un compilateur ?

Il est à peu près certain que la majorité des informaticiens n'auront jamais à écrire de compilateur. Cependant, mener à bien l'écriture d'un " gros " programme, le tester, documenter... a déjà un intérêt en soi. De plus, les algorithmes employés dans un compilateur sont repris dans la résolution d'autres problèmes : manipulation de listes de recherche (gestion de la table des symboles), vérification de la correction d'enregistrements (analyse syntaxique), traitement de caractères, de mots (grammaire BNF). Enfin, il semble intéressant pour un programmeur de connaître les dessous des choses ; comprendre comment les programmes se compilent et s'exécutent aide à la compréhension des limitations, obligations... imposées par les langages.

Cette "transformation" du texte d'un programme vers une forme exécutable nécessite plusieurs étapes. Le regroupement de celles-ci constitue le processus dit de compilation. Dans ce rapport chacune de chapteries qu'on décrira amènera à la réalisation d'un compilateur. Afin de mener à bien ce développement, notre groupe a dû exploiter plusieurs modules, parallèlement les uns aux autres. Le résultat obtenu est, donc, la synergie de plusieurs modules dont il est question dans la suite de ce rapport.

Chapitre 1

Gestion de projet

Certain changements organisationnels ont dû être opérés au cours de projet. En effet, la fiche de projet initiale montre quelques différences par rapport à la fiche finale. L'affectation des tâches de certains groupes (voir IHM), en fonction de l'avancée des travaux a été changée.

1.1 Groupe et affectation de tâches

- Chef de projet : Pequignot Arnaud
- Communication et rédaction : Mariano, Duffaud

1. Analyse syntaxique et lexicale - Mariano, Duffaud
2. Contrôle de type - Mariano, Duffaud
3. Compilation - Bonnet
4. Mémoire - Bidal
5. Interprétation MiniJaja - Pequignot
6. Interprétation JajaCode - Gervais
7. GUI - Genkai
8. Rapport - Duffaud, Mariano
9. Batterie de test - Gervais, Bidal, Bonnet

1.2 Organisation

- Diagramme de GANNT (voir fiche de projet)
- Groupe de TP 2
- Estimation du temps de developpement

1.3 Organisation du travail

- Documentation JavaDoc - il s'agit d'un outil comparable à la Doxigen . A chapterir de commentaires correctement structurés dans le code source des fichiers d'entête (.h) de chaque classe, doxygen va générer un index de toutes les classes, méthodes et variables définies dans ces fichiers. Le d ?eveloppeur doit au pr ?ealable avoir bien document ?e ses classes. Les formats de sortie des fichiers peuvent être HTML, LATEX, RTF ou XML.
- SVN - Il a été possible d'utiliser correctement ce puissant outil grâce à la mise à disposition d'un serveur dédié par l'UFR en conjonction avec le compilateur Eclipse.
- Eclipse - L'éditeur d'Eclipse a été conçu pour développer en Java dès le déchapter, aussi est-il extrêmement complet en la matière : reconnaissance de syntaxe, repli de blocs de code, visualisation des hiérarchies de type et des héritages, formateur et correcteur de code, assistant de création de méthode constructeur... Dans notre cas il a été aussi utilisé pour la génération du parseur (avec un plug-in JavaCC).

Chapitre 2

Analyse lexicale et syntaxique

2.1 JavaCC

Java Compiler Compiler (JavaCC) est le plus utilisé des générateur de parser pour Java. Un générateur de parseur est un outil qui lit les spécifications d'une grammaire et qui la convertit en program Java. En plus d'être un générateur de parseur, JavaCC fournit d'autre possibilité relative à la génération de parser comme la construction d'arbre, le debugage, etc... JavaCC prend comme entrée un fichier MaGrammaire.jj qui contient entre autre les descriptions des règles de la grammaire et produit un parser descendant (dans le fichier MaGrammaire.java). Une classe MaGrammaire est définie dans le fichier java. Elle implémente l'interface MaGrammaireConstants, définie dans MaGrammaireConstants.java et qui contient les définitions des mots clés de la grammaire.

D'une façon plus générale l'analyse consistera à prendre un programme écrit en MiniJaja (écrit dans un fichier .mjj) et le découper en une séquence d'unités lexicales. L'analyse syntaxique va traiter ces unités lexicales afin de donner un arbre de syntaxe abstraite (ASA) qui représente la structure du programme. (fichier .java) Cet arbre est la base du compilateur, puisque toutes les autres tâches comme le contrôle de type, le compilateur et l'interpréteur dépendent directement de cet arbre.

2.2 Analyse lexicale

Le but de l'analyse lexicale est de transformer une suite de symboles en terminaux (un terminal peut être par exemple un nombre, un signe '+', un identifiant...). Une fois cette transformation effectuée, la main est repassée à l'analyseur syntaxique. Le but de l'analyseur lexical est de découper un fichier source en symboles terminaux et de les fournir à l'analyseur syntaxique. L'analyseur lexical peut séparer logiquement une sequence de symboles dans de sous-sequences appelés *tokens* en le classifiant par un ordre spécifique. Un possible exemple avec du langage C pourra être :

```
int main()
{
return 0 ;
}
```

L'analyseur pourra donc aboutir a cette sequence :

```
\int", \ ", \main", \(", \)",
\ ", \
{", \\n", \\t", \return"
\ ", \0", \ ", \;", \
\n",
\
}", \\n", \" .
```

en affectant à chaque élément aussi un "type" :

```
KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF
```

2.2.1 Créer un ficher *.jj*

La syntaxe du fichier passé en paramètre à javacc pour l'analyse lexicale a une "forme" type de ce genre :

- `javac_options`
- `PARSER_BEGIN(MaGrammaire)`
- `java_grammaire_declaration`
- `PARSER_END(MaGrammaire)`
- `(production)*`

Dans notre cas spécifique le fichier contiendra :

```
options
{
STATIC = false ;
}
PARSER BEGIN(...)
class parser
{
```

```
static void main( String[] args )
throws ParseException, TokenMgrError
{
parser parser = new ...( System.in ) ;
parser.Start() ;
}
}
PARSER END(...)
```

Le fichier .jj doit contenir la définition des tokens pour ainsi pouvoir les identifier dans le code source que l'on passera au parser. En spécifiant tous les termnes que l'on peut rencontrer dans un langage précis on parviendra au but de l'identification dans le langage.

En premier lieu il est nécessaire de définir tous les caractères « invisibles » du langage ceux qui vont être lus par le parser mais qui n'ont pas d'influence sur la structure du programme et qui doivent donc être passés. On spécifiera ainsi notre analyseur lexical en utilisant les mots clés *skip* et *tokens* :

```
SKIP :
{
<( " " | "\t" | "\n" | "\r" )+ >
}
TOKEN :
{
< BOOLEAN: "boolean">
|  < CLASS: "class" >
...
```

De cette façon l'analyseur pourra "comprendre" les chaînes de caractères à décomposer et à passer au parseur.

2.3 Analyse syntaxique

Si la définition des tokens permet d'identifier les symboles d'un langage, la spécification du parser permet de définir la structure du langage et donc de vérifier la validité d'un programme par rapport à la grammaire requise. Il faut donc définir une séquence de token qui sera déterminée comme valide par le parser pour ensuite passer à la phase d'analyse syntaxique.

La phase successive à l'analyse lexicale est la vèrification du bon enchainement des tokens (terminaux) reconnus par JavaCC. Cette tâche est accomplie en gérant l'analyseur syntaxique qui reconnaît une grammaire du type BNF. Il est aussi possible, en plus de la vérification de la syntaxe de la grammaire, de lui faire exécuter des actions lorsqu'une régle est reconnue.

La spécification du parseur consiste à ce qu'on appelle une *production BMF* et apparaîtra comme une définition de méthode java :

```
void Start() :
{}
{
<NUMBER>
(
<PLUS>
<NUMBER>
)*
<EOF>
```

2.3.1 Génération du parseur

Ayant conçu le fichier parse.jj, on appellera JavaCC sur ce document.

```
D:
\home\\parser>javacc parser.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file parser.jj . . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.
File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
```

Cette opération va générer 7 classes java, chacune avec son propre fichier

- TokenMgrError - c'est une simple classe d'erreur. Elle est utilisée pour les erreurs détectés par l'analyseur lexicale et c'est aussi une sous-classe de Throwable.
- ParseException - c'est une autre classe d'erreur. Sous-classe de Exception et Throwable.
- Token - c'est une classe qui représente les tokens.
- SimpleCharStream
- ParserConstants - nombre de classes utilisés par l'analyseur lexical et parseur.

– ParserTokenManager - c'est l'analyseur lexical
– Parser - est le parseur en soi même.

2.4 Problèmes : elimination de la récursivité gauche

La grammaire proposé pour le langage MiniJaja est, dans certains cas, ambigüe parce qu'elle produit deux arbres d'analyse différentes pour la même entrée. Donc on pourrait avoir deux dérivations les plus à gauche différentes pour la même entrée.

La pluchapter des générateurs d'analyseurs syntaxiques (voir Flex/Bison) permettent la spécification de règles de "suppression de l'ambiguïté" qui suppriment les dérivations non désirées afin de s'assurer que la chaîne d'entrée a une seule interprétation possible.

Toutefois, dans notre cas, en utilisant JavaCC , on n'a pas d'autres choix que de réécrire la grammaire en une grammaire équivalente acceptable pour une analyse LL(1) (ou, pour plus tard, LR(1))

Pour résoudre ce problème, nous avons appliqué simplement la règle d'élimination de la récursivité du cours : Production « A -¿ A alpha — beta » peut-être remplacée par :

```
A -> {beta}A'
A -> {alpha}A' | {epsilon}
```

Chapitre 3

Le contrôle de type

Comme son nom l'indique, le contrôle de type est censé vérifier que le typage des opérandes est respecté. Pour cela, il doit réaliser un certain nombre de tâches sur l'arbre de syntaxe abstraite du code source et les vérifier. Le contrôle de type doit donc vérifier que chaque variable, tableau, constante ou méthode est bien déclaré(e) avant toute utilisation. Il doit vérifier egalement leur type (entier, booléen, void), ainsi que leur constructeur de type (variable, tableau, constante, méthode). Une méthode ne doit pas être appelée comme une variable, ou une variable comme un tableau. Les types des paramètres des méthodes doivent aussi être vérifiés à chaque appel. Le principal travail du contrôle de type consiste à vérifier que chaque noeud de l'arbre de syntaxe abstraite est bien du type dont il devrait être. Par exemple, il faut vérifier que lors d'une addition, on additionne bien deux entiers, et non des booléens ou des méthodes. Afin de faciliter le travail du compilateur et de l'interpréteur, le contrôle de type doit s'assurer que toutes les variables sont initialisées. Si ce n'est pas le cas, il s'occupe de les initialiser à True pour les entiers et à False pour les booleens.

Le contrôle de type peut être divisé en deux grandes chapteries :

1. Les méthodes pour le contrôle de type
2. Le paradigme visiteur

Le contrôle de type s'effectue par l'appel de la méthode controleDeType sur l'arbre de syntaxe abstraite. Cette méthode fait appel à la méthode controleDeType du noeud racine. A ce point on "controlera" le type en appellant cette méthode sur chacun de noeuds de l'ASA. Dans la majorité des cas, cette méthode vérifie le type de chaque noeud fils puis retourne le type du noeud courant. Le type d'un noeud fils est le résultat de la méthode controleDeType sur celui-ci.

```
public ControleDeType(SimpleNode noeud)
```

3.0.1 Liste de méthodes et de symboles : structures

La table des symboles est en réalité un "vector" contenant la liste des symboles. Ce type de structure s'avère trés performante car son utilisation est assez similaire à celle d'un tableau tout en restant de taille variable. Chaque symbole sera, alors, identifié par un nom. Dans la table des symboles, il sera possible d'ajouter un symbole, de le retirer ou de rechercher un symbole à chapterir de son identifiant.

Dans la chapterie méthode on créera les "vector" nécessaires :

```
private Vector lsVar = new Vector();
private Vector lsTab = new Vector();
private Vector lsCst = new Vector();
private Vector lsMethode = new Vector();
private Vector lsErreur = new Vector();
```

Representant chaqun un "symbole" : une variable, une constante, une méthode ou un tableau, déclaré dans le programme.

Comme on a pu voir on déclare aussi un "vector" lsMethodes. Cela parce-que là aussi il sera utilisé ce paradigme pour enregistrer les différentes fonctions déclarées. Le type de retour, le nom, puis les types des paramètres de la fonction sont enregistrés dans le vector lsMethode

3.0.2 Le visiteur

Le paradigme "visitor" se charge de visiter tous les noeuds de l'ASA. Une chaine de caractères sera retourné par le visiteur lorsque celui-ci visite un noeud. Dans le cas d'un noeud le visiteur retournera la chaîne de caractères ok, entier, booleen, erreur selon les circonstances.

Dans le cas ou le noued sont des feuilles on consultera le vector des symboles pour une vérification de la déclaration des symboles et une vérification de leur type (correcte ou incorrecte selon le noeud en question).

On visitera les fils de cette façon :

```
for (int i=0;i<node.jjtGetNumChildren();i++){
      node.jjtGetChild(i).jjtAccept(this,data);
}
```

Chapitre 4

La mémoire

4.1 La pile

4.1.1 Ses composants de base

Ce sont des quadruplets contenant des chaînes de caractères :

1. l'identifiant ;
2. la valeur (ou l'arbre d'interprétation pour une fonction ou la taille du tableau pour un tableau) ;
3. la sorte (ou nature) ;
4. le type de l'objet mémoire.

La pile a 3 variables globales :

1. sa représentation en HashMap ;
2. l'identifiant du sommet de pile ;
3. la position de l'identifiant dans sa liste.

Le fait que le dernier quadruplet soit connu permet d'avoir des accès rapides à la pile.

4.1.2 chaptericularités

Lors de la création de la mémoire, on crée une pile vide et un tas de taille donnée.
des fonctions "outil" ont été implantées pour éviter des redondances de lignes de code ou les tests :

1. effacer la pile (pour test interprétation jajacode) ;
2. récupérer le dernier élément ajouté à une liste (qui n'est pas forcément le dernier de la liste) ;
3. récupérer le quadruplet au sommet de la pile ;
4. récupérer un élément de la pile à une profondeur donnée ;

5. décaler les pointeurs de la pile : cela correspond à reconstituer la chaîne liant les éléments de la pile lors de la suppression d'un élément en milieu de pile ;
6. changer les pointeurs : lorsqu'on change l'identifiant d'un élément, on change aussi les pointeurs sur cet élément ;
7. Le maximum a été fait pour prévenir les erreurs et permettre au programme de es traverser quoiqu'il arrive.

4.2 Le tas

4.2.1 Description et aperçu général

Une fonction "defFreeSpace(int size, int beginAddress, boolean busy)"qui divise un espace donné en slots de puissance de 2 : Le tas est découpé en cellules ayant la plus grande taille (en puissance de 2) possible.

Ces portions (qui sont des objets "descriptionTas") du tas sont étiquetées en tant qu'espace, comme l'est une portion utilisée, mais marquées comme libres : la table d'indexage du tas contient la suite d'objets "descriptionTas".

Ces objets décrivant les morceaux de tas ont pour attributs :

1. leur adresse de début ;
2. la longueur de données qu'ils occupent ;
3. la puissance de deux supérieure (si l'espace est occupé uniquement) ou égale (surtout si l'espace est libre) la plus proche ;
4. le nom (l'identifiant) du tableau stocké associé ;
5. un booléen : "espace occupé" ou "espace libre".

On retrouve ainsi facilement les adresses de début et de fin des morceaux de tas. L'identifiant est mémorisé, car si l'index du tas le plus utilisé est celui avec pour clé les puissances de 2, il est très pratique de doubler cet index par une autre table de hashage indexée par identifiant : cela évite un parcours qui peut être complet du tas.

La fonction "defFreeSpace" définit les paramètres pour en appeler une autre servant à créer concrètement les espaces mémoires. cette fonction est également utilisée lors de la création ou ajout d'une occurence d'un tableau. C'est cette dernière qui remplit les deux tables de hashage référencant l'objet "descriptionTas".

L'ajout d'un élément dans le tas est fait en faisant appel à ces deux dernières fonctions : On parcourt le tas. Si aucun slot ne convient, on défragmente. Après défragmentation, si aucun slot ne convient, un message d'erreur remonte. Sinon, on ajoute le tableau donné dans le tas et l'espace libre du slot dans lequel il se trouve est redéfinit.

Le code étant très pur (- de 300 lignes), il semble que rien d'autre ne soit original et ne mérite plus de développement. Un mot quand meme :

1. Différentes occurences d'un tableau ont différents emplacements dans le tas ;
2. les indexes contiennent des listes chaînées d'objets "descriptionTas" ;
3. les "descriptionTas" d'espaces libres sont créés sans identifiant (constructeur à peine différent).
4. La fonction d'affichage affiche les attributs des objets "descriptionTas".

4.2.2 Avantages de l'architecture de ce tas

1. Accès rapides aux éléments (dû à la technique d'implantation conseillée) ;
2. Code robuste car il colle à la réalité (par exemple les données étant stockées dans un tableau d'octets, on ne pourra pas ajouter de types fantaisistes plus longs qu'entiers et booléens) ;
3. Algorithme fainéant : le désordre géométrique reste le plus total tant qu'il reste de la place en mémoire. On libère ainsi le processeur pour les autres tâches du compilateur.

4.2.3 Inconvénients de l'architecture

Un point est qu'il y a deux tables d'indexage à référencer. Néanmoins, étant donné le temps gagné, cela ne semble pas être un réel inconvénient.

4.2.4 Optimisations envisagées et envisageables

1. La fonction "defFreeSpace" est utilisable même lors du premier découpage et permet de créer un tas de taille différente d'une puissance de 2.
2. Pour optimiser, on pourrait envisager de défragmenter le tas chaque fois que 30

Chapitre 5

L'interprétation MiniJaja / JajaCode

5.1 L'interprétation Jajacode

L'objectif de l'interpréteur Jajacode est de parcourir une liste d'instruction Jajacode et d'effectuer les opérations correspondantes dans la pile. Cette liste d'instruction provient de la compilation préalable du code Minijaja. Ici l'interpréteur n'agit pas directement sur la pile mais fait appel à des fonctions de la chapterie Mémoire du projet, cela afin de faciliter la lisibilité et la simplicité du code.

Le fonctionnement

L'interpréteur reçoit une liste d'instruction Jajacode en provenance du compilateur, puis va parcourir un à un les éléments de cette liste en appliquant les règles d'interprétation propres à chacun de ces éléments. Par exemple, l'élément parcouru correspond à l'instruction ''pop", l'interprétation de cet élément aura pour résultat d'appeler la fonction *dépiler()* de la classe *Memory* et ainsi dépiler le premier élément de la pile.

L'interpréteur possède deux modes de fonctionnement : le mode ''Points d'arrêt" et le mode ''Pas à pas". Le premier consiste à interpréter les instructions les une à la suite des autres jusqu'à une ligne où a été définit un point d'arrêt et de stopper l'interprétation. Il suffit ensuite de relancer l'interprétation pour continuer jusqu'à la fin ou jusqu'au prochain point d'arrêt. On peu aussi à ce moment là faire appel au mode ''Pas à pas" (qui peut tout aussi être appelé au début ou à n'importe quel autre moment), cela a pour effet d'interpréter l'instruction courante, de se positionner pour interpréter la suivante, et de stopper l'interprétation avant un nouvel appel au ''Pas à pas" ou au ''Point d'arrêt". On peut donc ainsi lancer l'interprétation jusqu'à un premier point d'arrêt, puis faire du pas à pas pour observé la chapterie intéressante, puis relancer l'interprétation jusqu'au second point d'arrêt, etc.. . . .

Le choix de la programmation

La programmation de la chapterie Interprétation Jajacode a été composé de deux chapterie plus ou moins distinctes : l'interprétation proprement dites, et les objets/instructions Jajacode.

Pour ce dernier, la solution a été choisie en accord avec le groupe gérant la chapterie Compilation. Elle consiste à créer pour chaque instruction Jajacode un objet correspondant et de les stocker dans un vecteur lors de la compilation. Le résultat fut une liste importante de classes à implémenter. Heureusement beaucoup de ces objets sont semblables ce qui a allégé le code nécessaire. Chacune de ces classe possède donc des variables propres, un créateur, une fonction *toString()* qui retourne l'instruction sous forme textuel pour l'affichage de la chapterie Compilation, et enfin une fonction *interpretation()* qui sera développée ci-dessous.

L'interprétation a qu'en à elle été implémentée afin d'être exécuter de deux manières citées précédemment : pas à pas et avec point d'arrêt. La première consiste simplement à regarder l'indice actuel stocké, de lancer l'interprétation de l'instruction à cet indice dans le vecteur Jajacode, de stocker le nouvel indice obtenu, et de laisser la main. La seconde manière lance une boucle de type *dowhile* avec comme condition que l'interprétation n'est pas terminée et que l'indice parcouru n'existe pas parmi les points d'arrêt. L'interprétation d'une instruction est réalisée quand à elle dans cette boucle par l'appel de la fonction *interpretation()* de chacun des objets avec en paramètre la pile courante ainsi que l'indice parcouru. Cette fonction contient la traduction des règles d'interprétations Jajacode correspondante à l'objet. Cette interprétation est totalement différente suivant les objets regardés. En effet, les instructions comme *pop* ou *swap* auront pour simple effet d'appeler une instruction de la classe *Memory* . Tandis qu'une instruction comme *invoke()* créera tout d'abord un nouveau *Quadruplet* de pile puis l'empilera. Puis la fonction retourne finalement l'indice de l'instruction suivante de l'interprétation qui peut soit être l'instruction suivante de le vecteur d'instruction, soit une instruction situé ailleurs dans le dit vecteur (dans le cas d'une instruction *goto()* par exemple).

Problèmes rencontrés

La pluchapter des problèmes rencontrés se situent au niveau de l'interaction Interprétation / Mémoire. En effet, les appels à la mémoire et plus précisément à ses fonctions est finalement limité à une dizaine de celles-ci. Mais ces fonctions pouvant être appelés par l'interpréteur Jajacode aussi bien que l'interpréteur Minijaja, leurs paramètres ont nécessités diverses modifications ce qui a eu pour finalité de modifier une grande chapter des appels de fonctions qui posaient problèmes.

Un problème s'est aussi posé lors de la gestion de l'interprétation : Comment savoir de quelle type est l'instruction de l'indice parcouru ? et cela afin de pouvoir par exemple déclarer une variable locale correspondante. Finalement la solution trouvée a consisté en la création d'une classe *JJCObject* qui sera la classe mère de tous les autres objets Jajacode et permettra de définir n'importe lequel de ces objets.

Tests

La phase de tests n'a put être qu'appliqué à un nombre restreint d'instructions du fait que l'interprétation de certaines autres instructions nécessitait certaines fonctions mémoire qui malheureusement se terminaient par une erreur. Cependant, une série de tests a été effectuée à l'aide d'une classe Mémoire fictive qui se contentait d'afficher l'action effectuée dans la console, et cela afin de voir si les instructions Jajacode étaient bien toutes parcourus et interprétées.

Chapitre 6

Le compilateur

6.0.1 Principe

Un compilateur est un programme informatique qui traduit un langage ("source"), en un autre langage ("cible"), en préservant la signification du texte source. En pratique, un compilateur sert le plus souvent à traduire un code source écrit dans un langage de programmation en un autre langage, habituellement un langage d'assemblage ou un langage machine. Dans notre cas, le langage source est le Minijaja et le langage cible est la Jajacode, le compilateur a pour but de traduire les instructions MiniJaJa contenues dans l'arbre de syntaxe abstraite en instructions JaJaCode, suivant les règles définies dans le cours.

6.0.2 Compilateur

Stockage des données jajacode

Le jajacode est caractèrisé par une liste d'instructions possedant une adresse, de ce fait on n'a pas utiliser une liste en java parce qu'on ne peut pas associer à chaque instruction son adresse. L'utilisation d'un tableau n'est pas pratique, puisqu'on ne peut pas savoir à l'avance le nombre d'instructions qu'il va y avoir. On a donc utiliser un "Vector" qui achapterient au package java.util et qui permet de stocker des objets dans un tableau dont la taille évolue avec les besoins. Avec un "Vector", on peut donc ajouter une instruction comme dans une liste et utiliser son indice dans le vector comme adresse.

Parcours de l'arbre

Pour parcourir l'arbre, on utilise une interface "ParserVisitor" du package asa. Cette interface permet d'utiliser les données de la classe asa depuis la classe "Compilateur" en la faisant hériter de "ParserVisitor". A chaque noeud est appelé la fonction d'interface correspondant au type de noeud, on peut demander l'un des fils du noeud courant avec "jjtGetChild()" et son analyse avec "jjtAccept()". Ceci nous permet donc de faire un parcours complet de l'arbre d'analyse synthaxique.

fonctions de la classe *Compilateur.java*

Cette classe possede tout d'abord un constructeur, qui va créer un nouveau vector et initialiser la valeur de k à 1 (k permet de retenir la valeur de k du new(a,entier,var,k) pour les entêtes. On retrouve une fonction "getJajacode()" qui va retouner le vector contnant le jajacode. Les fonctions "print()" et "toString()" qui vont permettre d'afficher le jajacode contenu dans le vecteur. La premiere affiche directement dans la console et la seconde retourne un String contenant le jajacode. La fonction compilation qui prend en parametre un SimpleNode va lancer la compilation, c'est à dire qu'elle va parcourir l'arbre d'analyse syntaxique grâce aux "visiteurs" et remplir le vecteur jajacode.

6.1 Difficultés

Retraits des déclarations

Chaque noeud des déclarations ont pour caracteristique d'avoir un retrait, generalement il doit enlever la declaration qui avait été empiler dans la pile. Les fonctions de visite utilisent deux parametres, le second est un objet nommé "data" pour toutes les fonctions de visite de cette classe. On a donc utilisé cet objet pour indiqué si on effectue un retrait ou pas, on lui affecte un String "retrait" pour lancer le retrait. Dans les fonctions de visite pour les déclarations on gere les 2 cas, à l'aide d'un simple if. Exemple pour le noeud methode :

```
public Object visit(ASTmethode node, Object data){
if((String)data !="retrait"){
//push(n+3)
jajacode.addElement(new JJCTpush(Integer.toString(jajacode.size()+3)));
//new(i,t,meth,0)
String i=node.jjtGetChild(1).toString();
...
}else{
//swap
jajacode.addElement(new JJCTswap());
//pop
jajacode.addElement(new JJCTpop());
}
return null;
}
```

Variables d'entêtes

Les variables déclarées en entêtes possede un indice k qui est incrémenté dans l'ordre inverse de l'incrementation du vector. On a donc créé une variable globale k que l'on initialise à 1 dans le constructeur. Ensuite on crée dans le noeud entete le "new(i,t,var,k)" avec la valeur -1 pour k qui signifie que c'est une valeur temporaire et on incrémente k. Lorsque toutes les entêtes ont été parcourue, on modifie les valeurs de k,grace à la méthode "setIndice()" créée par l'interpreteur. Enfin on réinitialise la valeur de k à 1. Exemple :

```
//pens
node.jjtGetChild(2).jjtAccept(this,data);
//modification ordre des entetes
for (int c=1; c<k; c++) {
((JJCnewV)jajacode.get(jajacode.size()-c)).setIndice(c);
}
//reinitialisation de k
k=1;
```

If et *goto*

Les "if" et les "goto" ont pour chapericularité d'utiliser les adresses des instructions jajacodes, or il est impossible de savoir sur le coup quelle adresse il faut. Pour remedier à ce probleme, on enregistre l'adresse courante pour la passer au goto plus tard. On créé une instruction "nop" à l'adresse du if, on parcour l'arbre jusqu'à la fin du if puis on change le nop par un if avec comme adresse l'adresse courante. Enfin on créé le goto avec comme l'adresse de dechapter qu'on a gardée. Exemple (Noeud Tantque) :

```
public Object visit(ASTtantque node, Object data){
int addressGoto = jajacode.size();
//pe
node.jjtGetChild(0).jjtAccept(this,data);
//not
jajacode.addElement(new JJCTnot());
//if(n + ne + niss + 3)
int fi = jajacode.size();
jajacode.addElement(new JJCnop());
//iss
node.jjtGetChild(1).jjtAccept(this,data);
//retour goto
jajacode.addElement(new JJCgoto(addressGoto));
```

```
//valeur du if
jajacode.set(fi, new JJCif(jajacode.size()));
return null;
}
```

Plusieurs rêgles par noeud

Pour certains noeuds, il y a deux rêgles à gerer. Par exemple le noeud "Somme" à pour but d'effectuer la somme soit d'entiers soit de tableaux (csomme et csommeT dans le cours). Or l'arbre de syntaxe abstraite ne gere pas ces deux noeuds. Puisque ces deux noeuds ont pour premier fils la variable ou le tableau, on effectue un test sur l'instance de celui-ci, puis on effectue le code correspondant. Exemple du noeud Somme :

```
  public Object visit(ASTsomme node, Object data){
if(node.jjtGetChild(0) instanceof ASTident) {
    // csomme : somme(ident(i), e) => {pe + inc(i)}

    String ident = (String) node.jjtGetChild(0).toString();
    // pe
    node.jjtGetChild(1).jjtAccept(this,data);
    //inc(i)
    jajacode.addElement(new JJCTinc(ident));
}
else {
    // csommeT : somme(tab(ident(i), e1), e) => {pe1 + pe + ainc(i)}

    String ident = node.jjtGetChild(0).jjtGetChild(0).toString();
    //pe1
    node.jjtGetChild(0).jjtGetChild(1).jjtAccept(this,data);
    //pe
    node.jjtGetChild(1).jjtAccept(this,data);
    //ainc(i)
    jajacode.addElement(new JJCainc(ident));
}
return null;
}
```