1. Analyseur MiniJaja et JajaCode

Les analyseurs ont été générés à partir de Javacc et des règles de grammaire MiniJaja et Jajacode, traduites dans un fichier JJtree. Le compilateur Javacc va générer à partir de ce fichier les classes java pour pouvoir parser et réaliser l’arbre de syntaxe abstrait. L’arbre est composé de nœuds représentés par des classes java. Javacc permet aussi de faire une interface visiteur MiniJajaVisitor et JajaCodeVisitor qui sera utile pour parcourir l’arbre dans les modules, au lieu de modifier les classes java simulant les nœuds de l’arbre. Cet arbre sera ensuite réutiliser par le Contrôleur de type, les Interpréteurs et le Compilateur grâce au visiteur.

Peu de choix sur la conception ont été fait sur ces modules, mais il a été nécessaire de redéfinir un nouveau constructeur. Par défaut, le constructeur de base ne prenait pas en compte les String comme paramètre, mais pour les besoins des tests il nous en fallait un. Un autre choix a été de pouvoir générer un nœud de l’arbre pour les tests à partir d’un bout partiel de code. Nous avons donc dû apprendre comment un nœud pouvait être fait à partir de la bonne fonction appliquée au parser.

Les interfaces des visiteurs sont importantes car elles constituent le cœur des modules d’interprétation, de compilation et de contrôle. Les visiteurs permettent les opérations sur les différents nœuds de l’AST MiniJaja et JajaCode. Ceci implique que toutes les classes de l’AST MiniJaja possèdent une méthode jjtAccept(MiniJajaVisitor visitor, Object data) ou jjtAccept(JajaCodeVisitor visitor, Object data). Tandis que les interfaces ont une méthode jjtVisit pour chacun des nœuds des AST tel que : jjtVisit(ASTclasse n, Object data) ou jjtVisit(ASTpop n, Object data). Grâce à aux méthodes jjtVisit, il est possible de personnaliser l’action que l’on souhaite effectuer lorsqu’on parcours l’AST en fonction du visiteur utiliser.

Ces fonctions peuvent retourner une erreur, qui se déclenchée lors d’erreur de syntaxe dans le code passé en paramètre du parser. Cette erreur est récupérée et affichée par le GUI.

1. Interpréteur MiniJaja

L’interpréteur MiniJaja possède deux classes java MiniJajaInterpreteurVisitor et MiniJajaRetraitVisitor, implémentant l’interface visiteur MiniJajaVisitor. La première gère l’interprétation des différentes instructions MiniJaja comme la déclaration de constante, les appels de fonctions, les opérations tandis que l’autre s’occupe uniquement des retraits dans la mémoire. La séparation des tâches simplifie l’implémentation des méthodes dans le visiteur. En effet l’utilisation un seul visiteur gérant l’interprétation et le retrait en mémoire, nous oblige à jonglé avec les méthodes en indiquant à chaque fois la situation dans laquelle se trouve l’interpréteur.

Par exemple, la méthode visitant un nœud ASTvar est utilisée aussi bien pour la déclaration d’une variable dans l’interpréteur que pour le retrait de celle-ci dans le visiteur s’occupant du retrait. Devoir spécifier avant chaque appel si l’on se trouve dans une phase de retrait ou de déclaration complexifie le code et sa maintenance.

Le parcours de l’arbre est effectué automatiquement en visitant chaque nœud fils du nœud en cours de traitement. On descend dans les branches de l’arbre jusqu’à atteindre une des feuilles. La fonction visit prend aussi en paramètre la MemoireMiniJaja qui sera modifiée durant l’interprétation

Pour les erreurs, elle ne sont envoyé au GUI qui s’occupe de les afficher.

1. Interpréteur JajaCode

L’interpréteur JajaCode se comporte différemment de l’interpréteur MiniJaja à cause de la structure de l’arbre abstrait en forme de peigne, causée par la forme même du JajaCode car une ligne est équivalente à une instruction et que la lecture est séquentielle. Ainsi chaque nœud possède une instruction et le nœud de l’instruction suivante. Ces nœuds sont simulés par la classe ASTJajaCode, ayant 3 fils :

* Un pour l’adresse de l’instruction.
* Un pour l’instruction.
* Un fils de type JajaCode qui est l’opération à l’adresse suivante .

Pour simplifier son parcours, l’arbre est transformé en un tableau de nœud où les indices des instructions correspondent à leur adresse dans le code JajaCode. Comme ça, effectuer l’instruction d’adresse A consiste à récupérer le nœud à l’indice A.

Un autre changement comparé au MiniJaja est dû au fait que cet arbre ne possède qu’une seul feuille correspondant à l’instruction jcstop, indiquant que le programme se termine. Dans un premier temps, nous avions fait comme pour le MiniJaja pour le traitement des instructions, c’est-à-dire :

-Récupère le nœud ASTJajaCode à l’indice A

-Si l’instruction est jcstop, arrêt du programme. Sinon on effectue l’instruction, on calcul l’adresse suivante A’ et on visite le nœud d’indice A’.

Le problème est méthode utilisée pour visitée le nœud A ne se termine que lorsque la méthode du nœud A’ se termine. C’est-à-dire uniquement si on tombe sur jcstop. Pour des programmes courts, cette méthode ne pose pas de problème, mais lorsque les programmes sont longs et que l’on doit visiter une centaine de nœud, alors la pile d’appel peut être saturée par la méthode visitant le nœud ASTJajaCode. En plus de cela, la structure de visiteur ajoute deux appels dans la pile, un pour la fonction visite et un pour la fonction accept.

La solution que nous avons adopté pour palier à ce problème est l’utilisation d’une boucle while dans le visiteur de ASTJajacode. Pour ce faire, on considère A comme adresse de l’instruction courante. Tant que A !=adresse de fin, on récupère le nœud ASTJajaCode à l’indice A et on effectue son instruction et on obtient une nouvelle adresse pour A. Dans notre programme, l’adresse de fin est -1 car aucune instruction ne peut avoir cette adresse. L’exécution de l’instruction jcstop a pour effet d’affecter -1 à A. Avec cette solution, la pile d’appel ne contient qu’une seul fois la méthode pour le nœud ASTJajaCode. La fonction visit prend aussi en paramètre la Memoire qui sera modifiée durant l’interprétation. L’adresse courante est une variable de l’interpreteur.

Comme pour l’interpreteur MiniJaja, les erreurs sont renvoyées au GUI.

1. Compilateur

Le compilateur se base aussi sur le visiteur MiniJajaVisitor et se base sur le même principe l’interpréteur MiniJaja avec deux classes, une pour ajouter les méthodes lié aux opérations réalisant l’interprétation et une pour les méthodes lié aux retraits. La fonction visite prend en paramètre l’adresse n à laquelle commence les opérations JajaCode simulant l’opération visitée. Pour le nœud classe, la fonction visite prend en paramètre 1, car la compilation de classe commence à l’adresse 1 avec « 1 init ; » en JajaCode. Ils retournent un couple Paire (mem,nbInstr) où :

* mem : String qui contient le JajaCode
* nbInstr : int qui correspond au nombre de ligne JajaCode de mem.

Le compilateur reprend ainsi le principe vu en cours pour les règles de compilation ce qui implique que nous n’avons pas eu de réel choix à faire. De plus, le compilateur ne génère aucune erreur. Si des instructions sont mal conçues (ordre des instructions, instructions inexistantes…), elles seront détectées par l’analyseur JajaCode si le code est mal formée, ou elles génèreront des erreurs lors de leur interprétation. Nous avons fait ce choix car le contrôleur de type et l’analyseur MiniJaja nous assure déjà que le code est correct syntaxiquement et sémantiquement. Ainsi, les seules autre erreurs qui peuvent se produire se feront lors de l’interprétation du code, ce qui n’est pas du ressort du compilateur.

1. Mémoire MiniJaja

Le module MemoireMiniJaja s’occupe de simuler et des opérations sur le mémoire utilisée par l’interpréteur MiniJaja. Elle est composée d’un Tas et d’une TableSymbole. La table des symboles simule la pile de la mémoire pour l’enregistrement des variables, constantes et des méthodes, en y ajoutant le maniement de la portée. Ce qui n’est pas faisable en utilisant uniquement une pile.

Les appels des méthodes de gestion de la table des symboles et du tas se font en par l’emploi de fonctions dans la classe MemoireMiniJaja. C’est-à-dire que lors de l’interprétation du MiniJaja, toutes les opérations qui nécessite de manipuler la table des symboles ou le tas s’effectue en utilisant ces méthodes.

Les erreurs qui peuvent survenir lors des opérations sont remontées à l’interpréteur, qui les remonte au GUI.

1. Mémoire JajaCode

La classe Mémoire possède comme attribut une Pile et un Tas. Comme pour la MemoireMiniJaja, il est nécessaire de passer par les fonctions intermédiaires de la Mémoire pour pouvoir modifier le Tas ou la Pile. Comme pour le JajaCode, il n’y a pas d’ambiguïté avec la portée des variables une Pile suffit pour les enregistrer. La gestion des erreurs se passe comme pour MemoireMiniJaja.

1. Pile

La pile sert pour la Mémoire, c’est dans cette structure de donnée que sont stockées les variables pour l’exécution du JajaCode. Elle ne comporte une méthode push, pop et swap. Les « cases » de la pile possèdent le type Element. Un Element a un attribut valeur et un attribut Element pour connaitre l’Element suivant dans la Pile.

Les fonctions de gestion de la pile présentent dans le cours tel que empiler, declVar ou encore retirerDecl étant déjà Mémoire, la pile ne possède que les 3 méthodes : pop, swap et push. Ce qui nous a permis de la réutiliser pour d’autre module comme la MemoireMiniJaja. Un autre intérêt est le parallèle entre la mémoire MiniJaja et JajaCode, car les deux possèdent une pile (TableSymbole/pile) et un tas (Tas pour les deux) et leurs modification se font par des méthodes au sein des classes MemoireMiniJaja et Mémoire.

Les deux piles des mémoires utilise des Quadruplets pour enregistrer les informations.

1. Tas

Le tas est la structure qui enregistre les valeurs des cases d’un tableau. Il a comme attribut :

* Un tableau de Valeur comme Mémoire de taille N.
* Une table de hachage qui contient les différentes adresses des blocs vide, de taille log2(N). De la forme suivante :

|  |  |
| --- | --- |
| Key | Valeur |
| 0 | 0 -> 1 |
| 5 | 12 |
| 1 | 4->6 |
| … |  |

Elle se lit :

Les éléments possédants la clé Key ont comme taille 2Key et comme adresse val1, val2,…

Ici : Key = 0 -> Un bloc de taille 1 à l’indice 0 dans la mémoire et un autre à l’indice 1

Key = 5 -> Un bloc de taille 32 à l’indice 12.

* Une table de hachage servant de table d’indirection, elle fait le lien entre un tableau ayant l’identifiant t1 en mémoire et les valeurs de ces cases dans la mémoire du tas. Dans cette table on enregistre comme valeur : la taille, l’indice de départ, le nombre de tableau utilisant le tableau utilisant cette valeur (Lors de l’affectation de tableau par adresse).

Déroulement de la déclaration d’un tableau :

Lors de la déclaration d’un tableau de taille M dans la mémoire et que l’espace en mémoire restant est suffisant, on calcul t = log2(M) pour savoir la taille, en puissance de 2 du bloc petit bloc pouvant le contenir. On supprime ce bloc libre de taille 2x et on ajoute 2x-M blocs libres. On ajoute le tableau dans la table d’indirection et on le décale vers la « gauche » pour que son indice de départ soit égale à l’indice du premier bloc libre dans la mémoire et on réorganise les bloc libres (fusions des blocs de tailles égales, changement des indice)

Déroulement de la suppression d’un tableau :

Il y a deux cas en fonction du nombre d’occurrence de tableau utilisant la même clé dans la table d’indirection :

Si nombreOccurence > 1 : On décrémente uniquement le nombre d’occurrence de 1.

Sinon nombreOccurence == 1 : On ajoute dans la table des blocs libre les M bloc du tableau et on réorganise la mémoire, en décalant vers la « gauche » les tableaux qui se trouvaient après le tableau supprimer et en recalculant les adresses des bloc libres. Enfin, on supprime dans le supprime dans la table d’indirection.

1. Quadruplet

C’est la structure de format des variables, constantes, tableau et méthode dans les piles. Un Quadruplet possède un nom (String), un type(entier, booleen, vide) , une valeur(vrai, faux,1, omega) et le type de l’objet représenté(var,cst,meth, tab). Avec les quadruplets, on peut donc faire toutes les variables possibles dans le code MiniJaja.