Rapport Compilation

AUGEARD Wilfried
CHAUVEAU Amandine
GERARDIN Xavier
TARMIL Chaima

May 2019

Table des matières

1 Introduction **3**

2 Plan du programme **3**
2.1 Gestion de l'environnement 3
2.1.1 Utilisation dans le Parser 3
2.1.2 Environment 3
2.1.3 StackEnvironment 3
2.2 Code intermédiaire 4
2.3 Gestion des erreurs 6

3 Difficultés rencontrées/Solutions apportées **6**

4 Conclusion **6**

1 Introduction

Dans le cadre de l'UE Compilation, il nous a été demandé de réaliser un projet. L'objectif est de pouvoir analyser un langage donné, le LEA (Langage Elémentaire Algorithmique), issu des langages classiques de programmation impérative tel que le Pascal, le C ... afin de générer son code intermédiaire. Par la suite nous aurions dû générer le code Y86 mais faute de temps, il nous a été demandé de ne faire que la première partie de ce projet.

Le projet peut se décomposer en 5 parties :
- Génération de la grammaire et du jflex pour lire et reconnaître le langage (Analyse Lexical et Syntaxique).
- Implémentation des classes *Node*, pour vérifier le type et construit le typage des expressions (Analyse Sémantique).
- Environnement
- Implémentation des classes *IntermediateCode*, pour générer le code intermédiaire.
- Gestion des erreurs.

2 Plan du programme

2.1 Gestion de l'environnement

Le package "environment" permet de gérer le stockage des procédures, des types des variables et de la pile.

2.1.1 Utilisation dans le Parser

La variable *typeEnvironment* stocke les variables au niveau de la déclaration des types. Les fonctions et procédures sont stockées dans la variable *procedureEnvironment* au niveau de la déclaration des procédures.

Pour les variables locales, nous utilisons la variable *stackEnvironment* qui représente une pile. Le principe est que dès que le programme entre dans un nouveau bloc, nous empilons une HashMap dans la pile. La tête de pile contient toujours les variables locales de l'environnement du bloc actuel. A la sortie du bloc, nous dépilons la tête de pile pour revenir au contexte précédent. Ainsi, les variables empilées dans le bloc ne seront plus accessibles par la suite.

2.1.2 Environment

La classe *Environment* stocke un nom et une HashMap. Cette HashMap associe un Type à un String. Cela permet de retrouver facilement le type d'une variable. La classe implémente les méthodes *putVariable(String var, Type value)* et *getVariableValue(String variable)* qui permettent respectivement d'enregistrer une variable et de récupérer le type d'une variable.

2.1.3 StackEnvironment

La classe *StackEnvironment*, qui étend la classe *Environment*, implémente une pile pour enregistrer les variables locales. Pour représenter cette pile, nous utilisons une LinkedList<HashMap<String,Type». Elle contient les méthodes *push(HashMap<String,Type> hm)* et *pop()* qui permettent respectivement d'ajouter une HashMap à la pile et de retirer la tête de pile. La méthode *getVariableValue(String variable)* effectue un parcours sur toute la pile pour trouver la variable demandée car il est possible qu'elle ne soit pas dans la tête de pile si ce n'est pas une variable locale du bloc actuel, elle doit quand même être accessible. La méthode *putVariable(String var, Type value)* place la variable dans la HashMap de la tête de pile.

2.2 Code intermédiaire

Lors de notre implémentation des *Nodes* nous avons changé le constructeur de *NodeLiteral* de sorte à ce qu'il prenne un *Object*. Nous en avons fait de même pour le constructeur de *Const*.

Nous avons créé une variable *exp* dans *Node* de sorte à pouvoir récupérer l'expression d'un noeud grâce à la fonction *getIntExp()*.

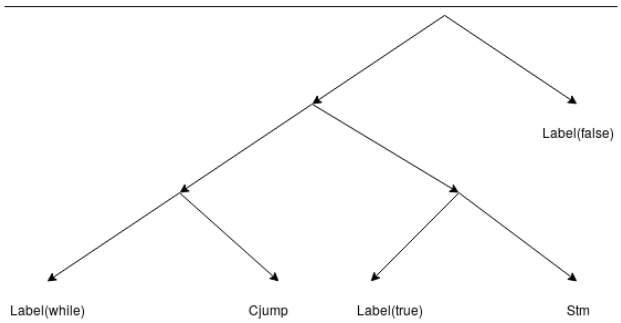
Voici un résumé des *IntermediateCode* renvoyé pour chaque *Node*.

- **Node, NodeCaseList , NodeList :** Appel récursif, grâce à la fonction *seqRec()* sur les éléments de *elts* créant des *Seq* reliant chacun des codes intermédiaires générés entre eux.
- **NodeArrayAccess :** *Mem* de l'expression générée par le *NodeExp t*.
- **NodeAssign :** *Move* des codes intermédiaires du noeud droit et du noeud gauche.
- **NodeCallFct :** Ici le *exp* et le code intermédiaire généré sont différents. *exp* est un *Call* contenant un *Name* de la fonction et une *ExpList* contenant le code intermédiare de chacun des arguments. Le code intermédiare renvoyé est un *Jump* sur le *LabelLocation* du *Name* créé précédemment.
- **NodeCase :** *Seq* d'un *Label* contenant un *LabelLocation* au nom de la case et du code intermédiaire généré par le *Stm*.
- **NodeDispose :** *Move* de l'expression du *Node* et de *null*.
- **NodeId :** *Name* avec un *LabelLocation* au nom du noeud.
- **NodeIf :** Deux cas sont possibles, soit il existe un *else*, soit il n'y en a pas. Il faut récupérer le *NodeExp* correspondant au noeud contenant l'expression booléenne afin de déterminer le relop (opérateur). On crée deux *LabelLocation* pour le then et pour le else. On crée ensuite un *Cjump* qui correspondra au code intermédiaire renvoyé.
- **NodeLiteral :** *Const* contenant la valeur du litéral.
- **NodeNew :** *Label* contenant le *LabelLocation* du *Name* généré par le code intermédiaire de son noeud *stm*.
- **NodeOp :** Deux sont possibles, soit c'est une opération binaire, soit une opération unaire. Nous faisons générer le code intermédiare des opérandes existants. Dans le cas de l'opération unaire le code intermédiaire du second opérande est remplacé par *null*. Puis nous réalisons un *Switch* sur l'opérateur. Nous pouvons ensuite créer le code intermédiare du noeud : *Binop*.
- **NodePtrAccess :** *Mem* de l'expression générée par le code intermédiaire de son noeud.
- **NodeRel :** Un *Switch* est réalisé sur l'opérateur afin de déterminer les *Binop* correspondant à celui-ci. Le code intermédiaire final renvoyé est un *Binop*. Une séquence de *Binop* est utilisée pour permettre de générer les opérateurs en utilisant seulement ceux implémentés dans la classe *Binop* c'est-à-dire *PLUS, MINUS, MUL, DIV, AND, OR, LSHIFT, RSHIFT, ARSHIFT, XOR*
  VOIR SCHÉMA POUR LE CHOIX DES BINOP
- **NodeReturn :** *Temp* conteant une nouvelle *TempValue*. On lance tout de même la génération du code intermédiaire de son noeud.

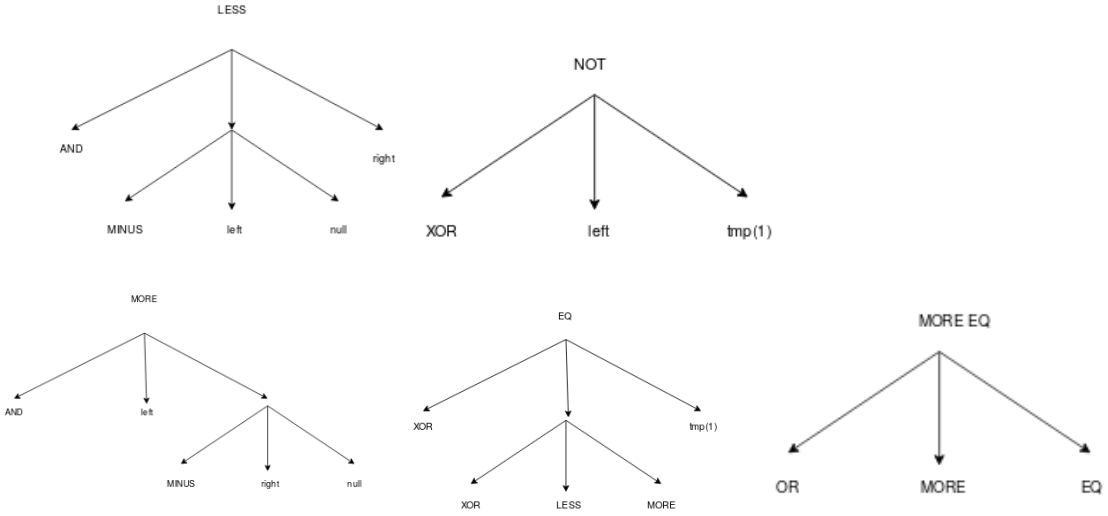
— **NodeSwitch :** *Seq* d'un *Label* contenant le *LabelLocation* du *Name* généré par le code intermédiaire de son noeud *e* et du code intermédiaire généré par son noeud *stm*.

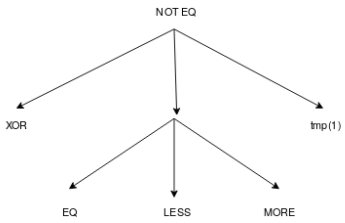
— **NodeWhile :** Deux cas sont possibles, soit le noeud *boolExpr* contient un noeud de type *Boolean* et donc correspond à une opération logique, soit ce n'est pas le cas est donc c'est une opération d'égalité implicite avec *null*. Pour tester dans quel cas nous nous trouvons, nous testons le type du *NodeExp* de *boolExpr* par comparaison de chaine de caractères. Si nous sommes dans le cas d'une opération explicite, nous réalisons un *Switch* sur l'operande, sinon nous considérons que l'opérande est un *EQ* et que le code intermédiaire du second opérande est remplacé par *null*. Nous créons aussi des *Label* correspondant aux différentes étapes d'un while (le test du *while*, quoi faire quand cela est *true*, où aller quand le test devient *false*).
Nous créons ensuite un *Cjump* comprenant le test à réaliser et les *LabelLocation* de *true* et de *false*. A partir de tout cela, nous pouvons créer une séquence de *Seq* reprenant toutes les étapes du while. (VOIR SCHÉMA REPRÉSENTATIF DE LA SÉQUENCE DE SEQ)

Précisions sur le code intermédiaire - NodeWhile :
Chaques pères de l'arbre correspondent à un *Seq*



Précisions sur le code intermédiaire - NodeRel :
Chaques père de l'arbre correspondent à un *Binop* et *left et right* correspondent aux opérandes.





2.3 Gestion des erreurs

Gestion des erreurs en utilisant la fonction *semanticError()* au niveau du fichier *parser.grammar*. Voici une liste des erreurs gérées :

— **Type nommé non défini :** gestion au niveau de *type_ declaration*

— **Range :** inversé, non défini. Gestion au niveau de *subrange_ type*

— **Function, procedure** double déclaration, appel sur fonctions non déclarées et avec le mauvais nombre d'arguments. Gestion au niveau de *procedure_ definition_ head*

— **Affectation** : Gestion au niveau de *assignment_ statement*

3 Difficultés rencontrées/Solutions apportées

Nous avons eu du mal à comprendre ce qui était attendu dans le parser au niveau des *return* des nodes, les tokens. Nous avons dû faire plusieurs tests avant de trouver les bons *%typeof*.

L'implémentation du code intermédiaire fût aussi difficile à réaliser. En effet, nous avions eu du mal à concevoir et comprendre la hiérarchie de celui-ci. Pour palier à ce problème, nous avons dû réaliser un schéma d'architecture des classes. De ce fait, il était plus clair de s'imaginer les dépendances entre chaque classe et de faire le lien entre le code intermédiaire et les classes *Node*. Un des problèmes persistant fût celui du typage entre les types *Stm*, *StmList*, *Exp* et *ExpList*. Les messages d'erreurs du compilateur nous ont grandement aidés.

De manière générale, la principale difficulté fût d'avancer sans étapes concrètes, avancer à l'aveugle.

4 Conclusion

Globalement, le projet fonctionne correctement. Des améliorations sont cependant possibles :

— Gestion des erreurs : certaines erreurs ne sont pas géré dû à un soucis de temps

— Génération du code intermédiaire qui est difficilement lisible, amélioration des fonctions *toString()* ou ajout d'une fonction *toDot()*.

— Code intermédiaire : NodeArrayAccess pourrait être géré de manière plus propre. Il faudrait faire un *Mem* d'un *Binop* entre l'adresse du tableau PLUS le *Binop* de *indice i* MUL *Taille d'un object du tableau*.