Rapport du TP de PDS M1 IL 2020/2021

Caruana Romain et Guibert Thomas

Sommaire :

Dans le cadre de nos études, nous avions à réaliser un compilateur VSL+ à l'aide de Java et de Llvm. Ce document explique nos choix de développement et explique les difficultés que nous avons rencontrées. Nous allons, dans un premier temps, parler des différents éléments que nous avons eu à créer, ainsi qu'expliquer pourquoi nous les avons développés de cette manière. Puis nous ferons une conclusion sur le travail que nous avons fait.
Vous pouvez retrouver notre code à cette adresse git.

I - Les expressions

Les expressions sont les opérations de calcule de base, elles sont au nombre de quatre : Additions, Multiplications, Soustractions et Divisions. En plus de permet de faire des opérations , les expression doivent gérer les priorité de calcul.

L'expression charger de l'addition était déjà donnée au début du TP, nous avons peut facilement produire les trois autres expressions qui extends aussi de Expression.

Grace a factor et expression dans le Parser, la priorité des multiplication et des divisions par rapport au soustraction et au addition est respectée. Le parser gère

aussi les priorité des addition et des soustraction avec parenthèse par rapport au multiplication et au division.

Pour cette partie du projet nous n'avons pas eu de difficultés. vu la simplicité des opérations, d'autant plus que l'opération add était déjà présente au début.

```
expression returns [TP2.ASD.Expression out]
    : r=expression PLUS l=factor  { $out = new TP2.ASD.AddExpression($l.out, $r.out); }
    | r=expression MOINS l=factor  { $out = new TP2.ASD.SubExpression($l.out, $r.out); }
    | f=factor { $out = $f.out; }
    ;


factor returns [TP2.ASD.Expression out]
    : p=primary { $out = $p.out; }
    | LP l=factor MULTI r=expression RP { $out = new TP2.ASD.MulExpression($l.out, $r.out); }
    | LP l=factor DIVI r=expression RP { $out = new TP2.ASD.DivExpression($l.out, $r.out); }
    | l=factor MULTI r=expression  { $out = new TP2.ASD.MulExpression($l.out, $r.out); }
    | l=factor DIVI r=expression  { $out = new TP2.ASD.DivExpression($l.out, $r.out); }
    | LP l=factor PLUS r=expression RP { $out = new TP2.ASD.AddExpression($l.out, $r.out); }
    | LP l=factor MOINS r=expression RP { $out = new TP2.ASD.SubExpression($l.out, $r.out); }
    ;
```

II - L'instruction d'affectation et déclaration des variables

L'affectation d'une variable va permettre d'assigner une valeur à une variable. Dans cette valeur on peut retrouver une autre variable, une constante ou encore une expression, qui contient des variables ou non. On y fera référence ici à la table des symboles afin de stocker les valeurs, ou bien de consulter la valeur des variables utilisées pour l'affectation.

Cette partie ne fut pas compliquée, car nous avions déjà à disposition la classe SymbolTable avec les méthodes nécessaires pour l'allocation ou la consultation.

Avec cette instruction de l'affectation, nous avons également mis en place la notion d'instruction avec la classe Instruction, qui contiendra par la suite les instructions If, While et Return plus tard. À noter que nous avons fait le choix de ne pas inclure les blocs (voir partie trois) dans la notion d'instruction car nous ne reconnaissons pas ici un bloc comme étant une instruction à part, mais simplement comme un regroupement d'instructions.

III - Les blocs

Comme nous l'avons vu dans la partie précédente, un bloc permet de regrouper plusieurs instructions en un ensemble. Un bloc sera alors utilisé dans le cas du programme, d'une fonction, d'un If ou d'un While, où il y a un besoin de contenir plusieurs instructions. Voici comment nous avons organisé nos blocs et nos instructions:

:

```
bloc returns [TP2.ASD.Bloc out]
    : {List<TP2.ASD.Instruction> l = new ArrayList(); List<TP2.ASD.Variable> r = new ArrayList(); }
    (INT v=variable {r.add($v.out);}  ( VIRG v=variable {r.add($v.out);} )* )?
    (ins=instruction {l.add($ins.out);} )+ { $out = new TP2.ASD.Bloc(r,l);}
    ;

instruction returns [TP2.ASD.Instruction out]
    : a=assignment { $out = $a.out; }
    | i=sialors { $out = $i.out; }
    | w = tantque { $out = $w.out; }
    | ret = retourne { $out = $ret.out; }
    ;
```

Nous pouvons voir ici que l'on commence par la déclaration de variables (temporaires ou non), puis on retrouve toute une série d'instructions.
Ici non plus nous n'avons pas rencontré de difficultés. Nous avons néanmoins débattu pour savoir si nous traitions les blocs comme des instructions (ce que l'on nous avait conseillé), ou si cet élément serait à part. Il aura été retenu que le bloc serait traité séparément du reste, car nous ne voyons pas l'utilité d'inclure les blocs dans les instructions. Il semblait plus logique pour nous de toujours passer par un bloc, même si celui-ci ne doit contenir qu'une seule instruction.
Il est important de noter également que nous n'avons pas gérer le niveau de profondeur des blocs.

IV - Les variable dans les expression

Cette partie a pour but la prise en compte des variables dans les expressions. La partie était plutôt simple à faire étant donné que les variables et les expressions avaient déjà été faites auparavant. Il nous fallait juste créer la nouvelle classe Variable Expression pour pouvoir mettre en place cette nouvelle fonctionnalité.

La chose la plus complexe à mettre en place était la vérification des variables. En effet, une variable doit être déclarée avant d'être utilisée dans une expression. Pour ce faire, il suffit de vérifier sa présence dans la table des symboles créée un peu plus tôt.

```
primary returns [TP2.ASD.Expression out]
    : INTEGER { $out = new TP2.ASD.IntegerExpression($INTEGER.int); }
    | IDENT { $out = new TP2.ASD.VariableExpression($IDENT.text); }
```

La gestion des variables dans les expression est gérée dans primary qui permet d'avoir soit des entiers comme pour les expression classique, soit des variables.

V - Les instructions If et While

- Nouvelle instruction If

  Pour l'instruction IF la création de Label était essentielle pour mettre en place sa structure. Il était aussi nécessaire de créer instruction pour la condition ainsi

qu'une autre pour les saut d'instruction. C'est trois éléments sont crée et gérer par le LLVM.

Après avoir créé tout ce qui était essentiel à la structure d'un if, il fallait juste donner les deux possibilités de if, avec ou sans Else. Puis dans la partie RetInstruction de la nouvelle classe IfElseInstruction, il manquait plus qu'à suivre la structure que l'on avait peut voir en cours avec le code 3 adresses.

Le Parser donne accès aux différentes possibilité du IF, il n'y a eu aucune difficulté pour cette partie.

```
sialors returns [TP2.ASD.IfElseInstruction out]
    :IF e=expression THEN b1=bloc ELSE b2=bloc FI { $out = new TP2.ASD.IfElseInstruction($e.out, $b1.out, $b2.out); }
    |IF e=expression THEN b1=bloc FI { $out = new TP2.ASD.IfElseInstruction($e.out, $b1.out,null); }
    ;
```

- Nouvelle instruction While

  L'instruction While est exactement comme l'instruction IF, les seules différences sont que While n'a qu'une seule forme possible et que les structures soient légèrement différentes. Il n'y a donc eu aucun problème pour cette partie relativement simple au vu de sa ressemblance avec If.

```
tantque returns [TP2.ASD.WhileInstruction out]
    : WHILE e=expression DO b=bloc DONE { $out = new TP2.ASD.WhileInstruction($e.out, $b.out); }
    ;
```

VI - Les prototypes et les fonctions

Pour la définition de fonctions, on peut utiliser un prototype qui déclare son entête (sauf pour la fonction "main"). Grâce à ce prototype, on peut vérifier si sa fonction correspondante va bien de paire avec. On vérifie d'une part le type de retour, mais aussi son nom, ainsi que ses paramètres. Pour la vérification des paramètres, nous sommes supposés ne regarder que le type de chaque paramètre et voir s'il y a une correspondance (dans l'ordre) avec les paramètres déclarés dans le prototype. Voici notre structure :

```
prototype returns [List<TP2.ASD.Prototype> out] locals [String ident, List<String> declarations]
    : { $out = new ArrayList<TP2.ASD.Prototype>(); }
    ( { $declarations = new ArrayList<String>(); }
    ( PROTO INT IDENT {$ident = $IDENT.text; }
    LP (( IDENT {$declarations.add($IDENT.text); }) (VIRG IDENT {$declarations.add($IDENT.text);})*)? RP
    { $out.add(new TP2.ASD.Prototype($ident, $declarations, new TP2.ASD.Int())); }

    | PROTO INT IDENT {$ident = $IDENT.text; }
    LP (( IDENT {$declarations.add($IDENT.text); }) (VIRG IDENT {$declarations.add($IDENT.text);})*)? RP
    { $out.add(new TP2.ASD.Prototype($ident, $declarations, new TP2.ASD.Int())); }))*
    ;


function returns [List<TP2.ASD.Function> out] locals [String ident, TP2.ASD.Type type, List<String> declarations]
    : { $out = new ArrayList<TP2.ASD.Function>(); }
    ( { $declarations = new ArrayList<String>(); }
    ( FUNC INT IDENT {$ident = $IDENT.text; }
    LP (( IDENT {$declarations.add($IDENT.text); }) (VIRG IDENT {$declarations.add($IDENT.text);})*)? RP
     b=bloc { $out.add(new TP2.ASD.Function($ident, new TP2.ASD.Int(), $declarations, $b.out)); }

    | FUNC VOID IDENT {$ident = $IDENT.text; }
    LP (( IDENT {$declarations.add($IDENT.text); }) (VIRG IDENT {$declarations.add($IDENT.text);})*)? RP
    b=bloc { $out.add(new TP2.ASD.Function($ident, new TP2.ASD.VoidType(), $declarations, $b.out)); }))+
    ;
```

Dans cette partie nous rencontrons un problème lors de la vérification des paramètres. En effet, nous n'arrivons pas à accéder uniquement aux différents types des paramètres du prototype. Cela rend la comparaison difficile car nous devons forcément comparer des paramètres identiques (même type et même nom) sinon une exception est lancée. Mais si le nom est identique, nous réussissons tout de même à comparer les entêtes et arrivons à procéder à la création du code Llvm.

VII - Les instructions Print et Scan

Les deux instructions Print et Scan servent respectivement à afficher ou à obtenir une valeur pour une variable. Dans le cas du langage Llvm, il faut, avant le début du code, déclarer l'appel à ces deux instructions (qui viennent de printf et scanf), et au début du code, déclarer tous les appels avec la taille qu'ils occupent ou bien ce qu'il y aura d'affiché.

Dans notre code, nous n'avons pas compris comment implémenter les déclarations d'appel à ces instructions. En effet nous ne savons pas à l'avance ce qu'il y aura à afficher ou bien quelle variable recevra une valeur en entrée. C'est pour ces raisons que nous n'avons pas réussi à implémenter ces deux instructions.

VIII - Les tableaux d'entiers

Les tableaux d'entier est une fonctionnalité du compilateur que nous n'avons pas pu implémenter dans notre projet. Il devait permettre l'utilisation du type tableau dans les fonctions et les instructions.

Cependant, nous avons rencontré un problème, nous ne savons pas si le type tableau doit être un sous-type du type int déjà présent ou un nouveau type à part entière. De plus, les modifications à faire pour ajouter cette fonctionnalité sont trop nombreuses. Nous avons eu beaucoup de mal à faire ces changements.

IX - Conclusion

Pour conclure, nous avons réussi à faire un compilateur fonctionnel mais qui ne possède pas toutes les options demandées du fait de notre difficulté à réaliser les dernières parties. Les parties basiques du projet fonctionnent, ce qui donne déjà une certaine marge de ce qu'il est possible de faire. Il resterait néanmoins important de comprendre comment comparer les types des variables afin de finaliser la partie fonctions / prototypes et également de pouvoir réaliser des print et des read.

Tout au long du projet, nous avons avancé à un rythme régulier, ce qui nous a permis de ne pas prendre de retard et ainsi de passer plus de temps sur des parties plus compliquées, comme les fonctions.