Rapport projet Compilation-LU3IN018

ADDAD Youva
3872388

Choun-Tong LIEU
Frédéric PESCHANSKI
Emmanuel CHAILLOUX

Pour les ajouts dans le lexer.flex et le parser.cup ,un pour l'analyse lexicale et l'autre pour l'analyse syntaxique, et je signale aussi que a chaque nouveau KAST crée il faut le rajouter dans l'interface KASTVisitor et du coup une définition dans compiler.java et KPrettyPrint.java

1.1) Ajout de nouvelles primitives d'entrées-sorties :

1.1.1) readInt () :

Dans un premier temps tout faut commencer par le fichier JCompiler.

J'ai ajouté dans le lexer.flex le mot clé readInt comme ci-dessus
"$readInt$" { $return\ symbol("READINT", sym.READINT);$ }

Dans le fichier parser.cup j'ai rajouté une nouvelle règle de syntaxe pour le readInt avec les parenthèses et terminal readInt

En travaillant toujours dans le frotend j'ai crée un ast pour le RESULT, dans le dossier frotend.ast j'ai crée une classe qui hérite de Expr, dont on va définir les méthodes expand, prettyPrint et buildDotGraphe, pour la méthode expand le readInt s'expanse avec le Kast call le KCall, j'ai choisis cette façon d'expansion parce que le readInt () est une primitive qui ne prend aucun argument et qui renvoie un entier, d'ou le fait de renvoyé un KCall pour le expand.

Sachant qu'on a utilisé la KCall pour la KAST on n'a pas à crée dans le middleend un nouveau Kast pour le readInt.

Ensuite dans le backend du JCompiler dans la classe PrimEnv j'ai rajouté dans la méthode statique defaultPrimEnv ()

```
primEnv.register(new Primitive("readInt", "Read Integer", "P_READINT"));
```

On ce qui concerne la compilation j'en est fini, passant maintenant à la NativeVM.

Dans le fichier prim.c j'ai rajouté une fonction $do\_readInt\_prim(stack)$ qui utilise la fonction scanf de la libraire du c ,la fonction scanf renvoie l'entier pour le nombre de lecture réussie ,donc si la lecture est réussite c.-à-d. lecture d'un entier on affecte l'entier on augmente varray de 1 et en stocke

l'entier au sommet de la pile, sinon on data=0 et on fait les mêmes opération.
J'ai fournis plusieurs exemples que vous trouverez dans testProjet
L'exemple de PlusMoins.js : qui lis un entier au clavier et le compare à 42 et regarde si les deux nombres sont égaux il arrête l'itération :

```
   var Nombre=readInt();
var NombreMystere=42;
while(Nombre!=NombreMystere){
  Nombre=readInt();
}
```

```
loading bytecode file: ../JCompiler/testProjet/PlusMoins.js.bc
--------------------
12
13
58
56
90
67
42
--------------------
VM stopping
```

L'exemple SommeChiffre.js : fait la somme de 10 chiffres rentrer au clavier :

```
var i=0;
var somme=0;
var f=0;
while(i!=10){
   f=readInt();
   somme=somme+f;
   i=i+1;
}
somme;
```

```
loading bytecode file: ../JCompiler/testProjet/SommeChiffre.js.bc
--------------------
1
2
3
4
5
6
7
8
9
10
55
```

```
------------------
VM stopping
```

1.1.2) **print() :**

Pour le print on fait la même chose aussi on rajoute dans le lexer.flex le mot clé « print » et dans le parser.cup j'ai décidé d'ajouter le l'analyse du print dans le opened_statement pour évité l'expansion en VoidExpr du coup affichage deux fois.

Donc dans le parser.cup j'ai rajouté la nouvelle règle dans opened_statement avec comme RESULT un nouveau « ast » qui prend l'expression entre parenthèse et terminal print
$RESULT = new\ Print(e, pxleft, rxright);$

Dans le frotend.ast j'ai crée une nouvelle classe qui hérite de Statement, avec la redéfinition des méthodes. Pour la méthode expand j'ai crée un nouveau KAST qui prend l'expansion de l'expression en argument.

Dans le middleend.kast j'ai crée du coup un nouveau KAST, KPrint qui a comme attribut la KExpr.

Dans le backend, dans le Compiler.java pour la méthode visit du KPrint, on utilise d'abord la méthode accepte de la KExpr ensuite on mets en sommet de pile la primitive print et on utilise le call(1) pour dire il prend un argument.

Ensuite dans PrimEnv.java on rajoute la primitive d'affichage

```
primEnv.register(new Primitive(print, print Expr, P_PRINT));
```

Pour la NativeVM Dans le fichier prim.c j'ai rajouté une fonction $do\_print\_prim(stack)$ qui utilise la fonction print de la libraire du c, donc dans la fonction do_print_prim j'ai fait un switch sur le type de la valeur du sommet de la pile et pour chaque type un affichage. (Pour le sprintf j'aurais pu faire avec un printf directement).

Exemples utilisés :

Le fichier printReadInt.js

```
print(readInt());
```

```
----------------
loading bytecode file : ../JCompiler/testProjet/printReadInt.js.bc
-------------------
12
12
-------------------
VM stopping
```

```
var f=true;
var g=12;
print(f);
print(g);
var add = lambda(a, b) {
          return a + b;
        };
print(add);
```

Le fichier printAllType.js

```
----------------
loading bytecode file: ../JCompiler/testProjet/printAllType.js.bc
-------------------
True
12
<<FUN>>

-------------------
VM stopping
```

1.2) Ajout d'une nouvelle structure de données :

Tout d'abord pour l'implémentation de la nouvelle structure de données j'ai définis les nouveaux symboles « cons », « car » et « cdr » dans le lexer.flex.

Et danse le parse.cup j'ai rajouté « cons », « car » et « cdr » en tant quand terminaux et j'ai rajouté aussi un non-terminal « pair » qui est une expression, et readPair qui aussi une expression non terminale qui est utilisé pour « car » et « cd ».

Pour la règle concernant le « cons » je l'ai défini dans pair ayant deux différentes façon de ce représenté, une en utilisant le mot clé cons et des parenthèses et a l'intérieur des parenthèses on deux expression séparés avec une virgule et l'autre avec des crochets.

Pour le RESULT dans les deux cas j'ai crée un nouveau ast « cons » qui prend les deux expressions.

Donc Cons est une classe que j'ai mise dans le frotend.ast et qui hérite de Expr et qui a deux expressions comme attribut, pour la méthode expand de « Cons » nous renvoie un KCons, pour l'expansion j'ai utilisé une arrayList pour ajouter les deux expansions des attributs expr (on peut aussi utiliser les deux expressions sans à avoir utilisé la arrayList).

Dans le middleend j'ai rajouté le KCons, qui a comme attributs « name » et « list » qui est l'arrayList des KExpr, pour le name c'est juste cons (on aurais pu directement le donner dans le KCons pour le name).

Du coup faudra ajouter dans le compiler et KPrettyPrint la définition du KCons.
Dans le backend la méthode visit pour le KCons, tout d'abord j'ai utilisé la méthode accept des deux KExpr ensuite j'ai cherché la primitive de name « cons » et pusher dans la stack

Et finalement on rajoute dans le PrimEnv.java

```
primEnv.register(new Primitive("cons","cons new pair","P_CONS"));
```

Dans la NativeVM dans le fichier prim.c on a déjà ce qui faut pour P_CONS j'ai juste enlevé le commentaire pour cons

Pour les règles concernant maintenant le « car » et « cdr », j'ai utilisé deux règles pour chacun, une règle prenant une paire et l'autre prenant un IDENTIFIER. J'ai utilisé cette façon pour pouvoir utilisé l'affection d'une paire a une variable « let » ou « var ».

Donc dans le frotend.ast j'ai rajouté quatre classes héritant toute de Expr « Car », « Car2 », « Cdr », « Cdr2 », comme je l'avais dis ci-dessus les classes comportant « 2 » utilise dans nom de variable pour pouvoir les retrouvés sois dans l'environnement locale ou globale, ou par mesure de sécurité dans environnement des primitives.
Pour celle prenant une Expr en argument (une paire plus précisément), dans la méthode expand on revois directement KCar ou KCdr prenant l'expression expansé, pour les deux dernières la méthode expand renvoie KCar2 ou KCdr 2 prenant le nom de la variable a la construction.

Dans le middleend j'ai rajouté aussi ces quatre classes « KCar », « KCar2 », « KCdr » et « KCdr2 »

Dans le backend compiler aussi j'ai donnée une définition de ces quatre méthodes, celle prenant la paire on utilise sa méthode accept et on push la primitive « car » ou « cdr » on utilisant call(1), pour celle prenant un nom de variable faudra cherché dans les environnements.

Enfin en rajoute dans PrimEnv.java

```
primEnv.register(new Primitive("car","first elem","P_CAR"));
primEnv.register(new Primitive("car","first elem","P_CAR"));
```

Dans la NativeVM reste qu'a enlève les commentaires parce que on les définitions de « car » et « cdr ».

Toujours dans la NativeVM, j'ai un peu modifier la primitive d'affichage ou j'ai rajouté une fonction print_pair qui prend une value_t en argument, et regarde le type de value_t si c'est une paire elle s'appelle récursivement jusqu'à tomber sur un type terminal. Cette fonction je l'ai incorporé dans « do_print_prim » dans case T_PAIR.

Exemples utilisés :

Le fichier ConsCarCdr.js

```
var p=cons(1,2);
let q=cons(3,4);
print(p);
print(q);
print(car(p));
print(cdr(q));
```

```
-----------------
loading bytecode file: ../JCompiler/testProjet/ConsCarCdr.js.bc
-------------------
cons(1,2)
cons(3,4)
1
4
<unit>
```

Le fichier ConsCdrCar2.js

```
var p= [1,2];
let q=[3,4];
print(car(p)+cdr(p)+car(q)+cdr(q));
var r=[1,[2,[3,4]]];
print(r);
```

```
----------------
loading bytecode file: ../JCompiler/testProjet/ConsCdrCar2.js.bc
------------------
10
cons(1,cons(2,cons(3,4)))
<unit>
------------------
```

1.3) Ajout de multi-déclarations pour simplifier le code :

Ici pour la multi-déclarations on a tous ce qui nous faut dans le lexer.flex, dans le parser.cup j'ai tout d'abord ajouter un non terminal multAffect pour la multi-déclarations des « var » qui est un Statement , donc pour le RESULT des « var » j'ai crée un nouveau ast MultAffect1 qui prend les deux vars « var1 » et « var2 » et la paire, Et un autre ast qui prend les deux vars et un IDENTIFIER de la paire. Pour le premier la méthode d'expansion renvoie un KSeq, ainsi la list des Statement contient les affectations var1 du premier élément de la paire et var2 le deuxième élément utilisant les classes « Car » et « Cdr » et « Var ». Pour la deuxième elle utilise « Car2 » et « Cdr2 » et « Var ».

Je n'ai rien fait dans le middleend et le backend sachant que toute est déjà ajouté.

Pour le « Let » j'ai rajouté 4 règles pour la multi-déclarations toujours dans statements, tout d'abord j'ai utilisé le même concept que la multi déclaration de var j'ai créé deux nouveaux ast, un prenant la paire l'autre prenant l'IDENTIFIER de la paire.

Donc la classe MultiAffect2 qui hérite de Statement j'ai utilisé le Let déjà défini pour affecter le deuxième élément de la paire a la deuxième variable, le choix de l'expansion de la deuxième valeur est dans le souci de pouvoir l'utilisé dans la fermeture de la première variable, donc j'ai utilisé une KClosure par la suite pour la fermeture avec la List Statement suivante, ensuite j'ai fait le choix aussi d'utilisé le KCall qui prend la fermeture précédente et l'affectation de la première variable au première élément de la list et j'ai décidé de renvoyer KReturn pour la méthode expand de MultiAffect2 qui prend a la construction le résultat du KCall.

Pour MultiAffect2P la même chose a la différence de l'utilisation de

« Car2 » et « Cdr2 ».

Pas de changement dans la NativeVM ni dans le backend ou le middleend.

Exemples utilisés :

Le fichier MultiAffect1.js

```
var p=cons(1,2);
var [a , b] = p;
print(a);
print(b);
let q =cons(3,4);
let [c , d] = q;
print(c);
print(d);
```

```
-----------------
loading bytecode file: ../JCompiler/testProjet/MultiAffect1.js.bc
-------------------
1
2
3
4
-------------------
```

Le fichier MultiAffect2.js

```
var p =[4,5];
var [a , b] = p;
let q= [6,7];
let [c , d] = q;

print(a+b+c+d);
```

```
-----------------
loading bytecode file: ../JCompiler/testProjet/MultiAffect2.js.bc
-------------------
22
<unit>
-------------------
VM stopping
```

1.4) Ajout de la valeur nil pour la construction des listes :

Pour la valeur nil j'ai crée directement un nouveau type nil, du coup tout d'abord j'ai rajouté le mot clé nil dans le lexer.flex .

Dans le parser.cup j'ai rajouté aussi un terminal nil qui est une expression et donc une nouvelle règle dans « expr » qui a comme résultat $RESULT = new\ Nil\ (nxleft, nxright)$;

J'ai crée un nouveau ast dans le frotend qui a pour nom Nil qui étends Expr et qui a aucun attribut, pour sa méthode expand renvoie un KNil.

Dans le middleend j'ai rajouté du coup le KAST KNil qui n'a aucun attribut

Dans backend.bytecode j'ai rajouté une nouvelle classe Nil qui a pour opcode « 6 » et de taille « 1 »

Toujours dans backend.bytecode j'ai ajouté dans la méthode getValueSet j'ai ajouté une nouvelle valeur Nil.

Ensuite toujours dans le backend , le compiler j'ai rajouté la définition de visit prenant en argument KNil qui fait juste un push de nil.

Pour finir dans la NativeVM j'ai fait plusieurs ajouts et plusieurs modifications.

Commençant par le fichier prim.c, j'ai modifier la fonction « do_eq_prim » pour tenir compte de la valeur nil, du coup en prenant les éléments successifs au sommet de la pile on regarde si les types sont égaux en renvois true si les valeurs sont égales, false sinon, sinon si l'élément au sommet de la pile a pour type T_NIL et celui le suivant a pour type T_PAIR, du coup on met false au sommet de la pile.

J'ai aussi rajouté « Nil » dans l'affichage.

Dans le fichier value.c j'ai rajouté une fonction qui prépare une valeur de type T_NIL dont on ce préoccupe pas de la data. J'ai aussi rajouté une une nouvelle fonction value_is_Nil qui regarde si on a pour type T_NIL

Dans le fichier vm.c dans la fonction « vm_execute_instr », dans le case I_PUSH, j'ai rejoué case T_NIL qui ajoute une nouvelle valeurs de type T_NIL.

Exemples utilisés :

Le fichier Nil1.js :

```
var p=cons(1,nil);
print(cdr(p));
```

```
-----------------
loading bytecode file: ../JCompiler/testProjet/Nil1.js.bc
-------------------
Nil
-------------------
VM stopping
```

Le fichier Nil2.js :

```
var maliste = cons(14,cons(1, cons(2, cons(4, nil)))) ;
function length(l) {if (l == nil) {return 0;} else {return (1 + (length(cdr(l))));}}
length(maliste) ;
```

```
-----------------
loading bytecode file: ../JCompiler/testProjet/Nil2.js.bc
-------------------
4
-------------------
VM stopping
```

Exemple Générale :

Le fichier All.js :

```
var maliste = cons(14,cons(1, cons(2, cons(4, nil)))) ;
function length(l) {if (l == nil) {return 0;} else {return (1 + (length(cdr(l))));}}
```

```
length(maliste) ;
print(maliste);
var add = lambda(a, b) {
          return a + b;
        };
var mult = lambda(a, b) {
       return a * b;
   };
var div = lambda(a, b) {
       return a / b;
   };
print(add);
var[a,b]=cons(2,4);
let [c,d]=cons(8,16);
var e=mult(add(a,b),div(d,c));
print(e);
var f=readInt();
print(e*f);
```

```
-----------------
loading bytecode file: ../JCompiler/testProjet/All.js.bc
-------------------
4
cons(14,cons(1,cons(2,cons(4,Nil))))
<<FUN>>

12
1
12
-------------------
VM stopping
```