

Méthode de compilation

Gestionnaire d'erreur

Le fait d'avoir un gestionnaire d'erreur permet de poursuivre la détection de nouvelles erreurs sans arrêter la compilation dès la première erreur rencontrée.

Table des symboles

- Entrée: identificateur
- Symbole: contient le no de ligne, la valeur, et toutes autres informations liées à la déclaration.

Construction de la TDS

A chaque déclaration, on ajoute une nouvelle entrée dans la table des symboles.

Classe TDS

```
class TDS{ // doit être une instance unique (singleton pattern)
    private static TDS instance = new TDS();

    // permet d'utiliser l'unique instance
    static TDS getInstance();

    // Peut retourner l'erreur double déclaration.
    int ajouter(Entrée, Symbole); // ajoute dans le bloc courant. sous réserve des
    blocs ouverts

    // retourne un symbole ou null
    String identifier(Entrée);

    // permet d'entrer/sortir d'un bloc
    void entreeBloc();
    void sortieBloc();
}
```

Classes Entrée/Symboles

L'entrée correspond à l'identificateur de la dernière unité lexicale reconnue. Par ex dans:

booléen monBool = true; // L'entrée serait "monBool", reconnue sous une unité lexicale booléenne.

Un symbole contient tous les informations attachées à la déclaration de "monBool". Numéro de ligne, valeur, type, etc...



Exemples CUP et interface TDS

```

BLOC  -> debut (1) LDECV LINS (2) fin
LDECV -> DECL LDECV
        | NULL
DECL  -> TYPE ident (3) ';'
        | TYPE (4) ident (PARAM) debut LDECV LINS (6) fin
PARAM -> TYPE(4) ident (7) SUITE_PARAM
        | NULL
SUITE_PARAM -> ', ' TYPE (4) ident (7) SUITE_PARAM
        | NULL
TYPE   -> boolean (8)
        | entier (9)

```

```

// 1
Tds.getInstance().entreeBloc();
// 2
Tds.getInstance().sortieBloc();
// 3
Entree e = new EntreeEntBool ( new Ident(uniteCourante));
Symbole s = new SymboleEntBool(ligne, lastType);
Tds.getInstance().ajouter(e,s);
// 4
Type t = lastType;
// 5
Ident ifonc = new Ident(uniteCourante);
Tds.getInstance().entreeBloc()
lastParam = new Parametres(spf);
// 6
Tds.getInstance().sortieBloc();
Entree em = new EntreeMethode(ifonc, lastParam);
Symbole sm = new SymboleMethode(ligne, t, lastParam);
Tds.getInstance().ajouter(em,sm);
// 7
Ident ipf = new Ident(uniteCourante);
Entree epf = new EntreeVarLocPar(ipf);
Symbole spf = new SymboleVarPar(ligne, t);
lastParam.ajouter(spf);
Tds.getInstance().ajouter(epf,spf);
// 8
lastType = TypeBooleen.getInstance();
// 9
lastType = TypeEntier.getInstance();

```

Implémentation de TDS

- Utiliser un dictionnaire et une pile

Classe Type

```

public abstract class Type {
    /** Conforme de 2 types
    * @param: other un autre type
    * @return: vrai si this est conforme $ other
    */
    public abstract boolean estConforme( Type other );
} // class Type

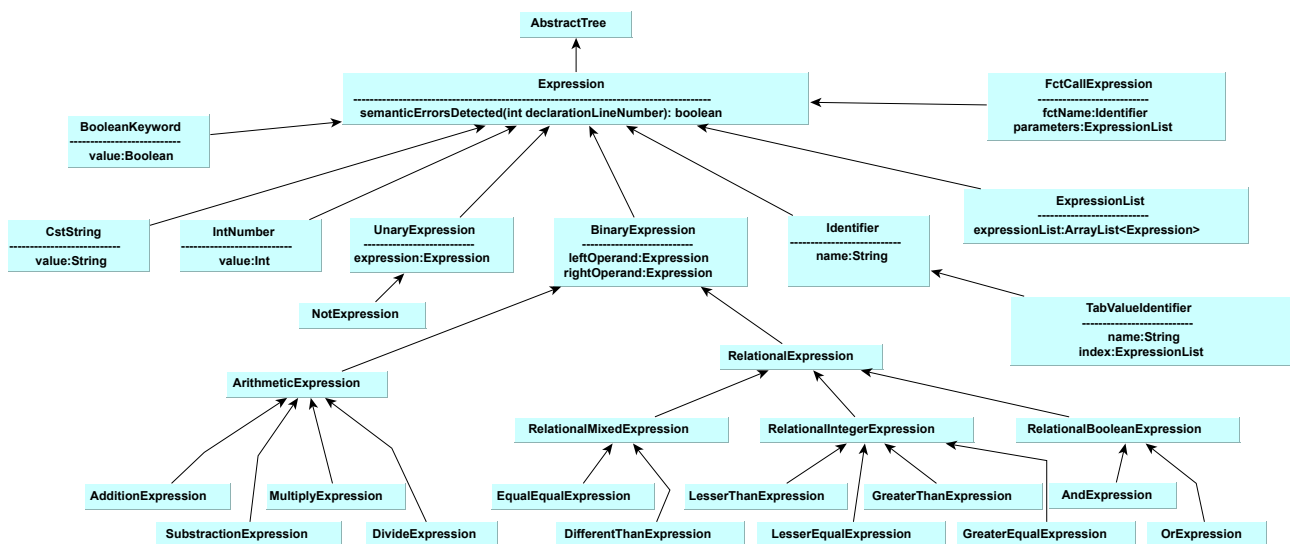
public class TypeEntier extends Type {
    public boolean estConforme(Type other) {
        return other instanceof TypeEntier;
    } // estConforme
} // TypeEntier

```

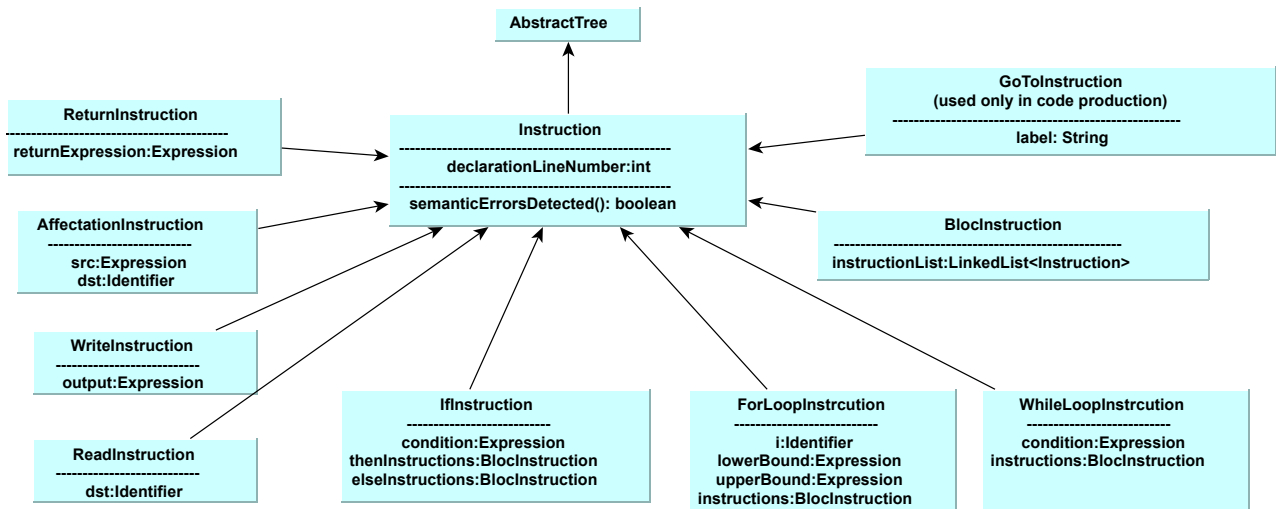
Abre abstrait

- Conceptionné avec le composite pattern
- Un arbre est construit à chaque réduction d'une règle.
- Lorsqu'une instruction se passe bien (dérivation), l'arbre abstrait de l'instruction est construit et accessible. L'arbre est rangé dans la **pile des arbres**
- L'analyse réussie de chaque non terminal entraîne l'empilement de l'arbre abstrait correspondant. A la fin de l'analyse, la pile ne contient plus qu'un seul arbre: celui du texte complet.

Classes d'expressions



Classes d'instructions



Exemples:

Affectation:

```
INSTR -> AFFECTATION
AFFECTATION -> ACCES '=' EXPR ';'

```

```
Expr source = (Expr) (pilesArbres.depiler());
Ident dst = (Ident) (pilesArbres.depiler());
pileArbres.empiler(new Affection(dst, src));

```

Condition:

```
INSTR --> CONDITION
CONDITION -> si EXPR alors LINSTR sinon LINSTR finsi (4)
              | si EXPR alors LINSTR finsi (5)

```

```
/*(4)*/ Linstr sinon = (Linstr) (pilesArbres.depiler());
        Linstr alors = (Linstr) (pilesArbres.depiler());
        Expr ec = (Expr) (pilesArbres.depiler());
        pileArbres.empiler(new Si(ec, alors, sinon));
/*(5)*/ Linstr alors = (Linstr) (pilesArbres.depiler());
        Expr ec = (Expr) (pilesArbres.depiler());
        pileArbres.empiler(new Si(ec, alors, null));

```

Expression (*arithmétique binaire*)

```
EXPR -> EXPR OPBIN EXPR (6)
        | OPERANDE

```

Exemple à partir d'un code

```

si (a == b) alors
    x = 1
sinon
    x = 3 * x
finsi

```

1. On réduit/détecte/empile les identifiants *a* et *b*. On réduit/détecte/empile l'expression *Egal* en dépilant les identifiants *a* et *b*. On final, on se retrouve avec l'expression *egal* en haut de la pile d'arbre.

```

/*(8)*/ // depiler a et b
String nom = (String) pileArbres.depiler();
String nom = (String) pileArbres.depiler();
// empile a et b sous forme d'identifiant
pileArbres.empiler(new Ident(nom));
pileArbres.empiler(new Ident(nom));

/*(13)*/ // depiler a et b cast (expr)
Expr operandeDroite = (Expr) pileArbres.depiler();
Expr operandeGauche = (Expr) pileArbres.depiler();
// empiler Expr Egal
pileArbres.empiler(new Egal(operandeGauche, operandeDroite));

```



2. On réduit/détecte/empile l'expression *alors*, puis, ensuite l'expression *sinon*. On se retrouvera avec cet arbre abstrait à la fin



Syntaxe du langage Hepial

```

programme identifiant
    DECLARATIONS VARIABLES / FONCTIONS
debutprg
    INSTRUCTIONS*
finprg

```

Types possibles

- entier
- booleen

Déclarations

Constantes

```
constante entier ident = 3;
constante entier ident = (3+5);
constante booleen ident = vrai;
constante booleen ident = faux;
```

Variables

```
booleen ident;
entier ident1, ident2, ident3;
```

Tableaux

entiers et booleans

```
entier ident [(2+3) .. 10]; // tab[5], tab[6], tab[7], tab[8], tab[9], tab[10]
entier ident [(4+3), 3..4]; // deux dimensions
boolean ident [7 .. 9];
entier ident [ fct1() .. ident1 ];
entier ident [ tab[1] .. (9+1)];
```

Manipulations de variables

```
maVar = var2;
maVar = tab[1];
tab[3] = 4;
mavVar = 56;
```

Conditions

```
si EXPRESSION alors
    CORPS
sinon
    COPRS
finsi
```

Boucles

```
// while
tantque EXPRESSION faire
    CORPS
fintantque

// for
pour ident allantde EXPR a EXPR faire
    CORPS
finpour
```

Fonctions

```
entier maFonction( boolean param1, entier param2 )
    entier declaVar1;
    boolean declaVar2[0..1];
debutfonc
    INSTRUCTIONS*
    (retour EXPR)+
finfonc
```

Fonctions systèmes?

```
lire maVar; // équivaut à maVar = scan(); ?
ecrire maVar; // équivaut à print(maVar); ?
ecrire "ceci est une chaine de caractères \" avec un guillemet au milieu";
```

Production du Code

- Exemples: <http://www.cs.sjsu.edu/~pearce/modules/lectures/co/jvm/jasmin/demos/demos.html>
- Commandes JVM: <https://cs.au.dk/~mis/dOvs/jvmspec/ref-Java.html>
- Build the jasmin code:

```
java -jar jasmin.jar examples/HelloWorld.j
```

- Dans Hepial, on va déclarer que des fonctions statiques. *du coup, elles ne prennent pas de place dans le stack?*

Questions

- < > <= acceptent des booléens?? **bof... utiliser le bon sens..**
- qu'est-ce que c'est que ça? <> **C'est le !=**
- qu'est-ce que ça veut dire? Quel opérateur conditionnel? Relationnel? **PAS A CONSIDÉRER**

C7: L'opérateur conditionnel accepte un premier opérande booléen. Si les deux derniers opérandes sont entiers (resp. booléens), le type de l'expression est entier (resp. booléens).

•