Documentation de l'Analyseur Lexical et Syntaxique pour le Langage Deca

Benjelloun Otman
Birée Thomas
Boulouz Amine
El Goumiri Rida
Loginova Aleksandra

January 15, 2025

Analyseur Lexical pour le Langage Deca

Ce document présente en détail l'analyseur lexical (Lexer) du langage Deca, développé avec **ANTLR4**. L'analyse lexicale est une étape cruciale dans le processus de compilation, consistant à transformer le code source en une suite d'unités lexicales (tokens). Ces tokens alimentent ensuite l'analyseur syntaxique (parser). Le Lexer joue un rôle fondamental en identifiant des motifs syntaxiques tels que les mots-clés, les symboles, les identifiants et les littéraux.

Dans le contexte du compilateur Deca, le Lexer est la première étape de traitement du code source. Il se situe avant l'analyse syntaxique et la vérification contextuelle. Cette hiérarchie garantit une structure claire et modulaire.

Configuration de Base

Le lexer est défini comme suit :

```
lexer grammar DecaLexer;

options {
   language=Java;
   superClass = AbstractDecaLexer;
}
```

Cette configuration précise que le code généré sera en Java et que notre lexer hérite de la classe `AbstractDecaLexer`, facilitant l'extension des fonctionnalités.

Catégories de Lexèmes

Mots-réservés

Les mots-réservés définissent la structure du langage Deca. Les règles suivantes sont utilisées :

```
PRINTLN     : 'println';
PRINT       : 'print';
CLASS       : 'class';
EXTENDS     : 'extends';
IF          : 'if';
ELSE        : 'else';
ASM         : 'asm';
RETURN      : 'return';
WHILE       : 'while';
TRUE        : 'true';
FALSE       : 'false';
INSTANCEOF  : 'instanceof';
NEW         : 'new';
NULL        : 'null';
THIS        : 'this';
READINT     : 'readInt';
READFLOAT   : 'readFloat';
```

```
PRINTX      : 'printx';
PRINTLNX    : 'printlnx';
PROTECTED   : 'protected';
```

Symboles et Opérateurs

Les symboles et opérateurs sont essentiels pour définir les expressions et structures syntaxiques :

```
// D limiteurs
OBRACE  : '{';
CBRACE  : '}';
OPARENT : '(';
CPARENT : ')';
SEMI    : ';';
DOT     : '.';
COMMA   : ',';


// Op rateurs arithm tiques
PLUS    : '+';
MINUS   : '-';
TIMES   : '*';
SLASH   : '/';
PERCENT : '%';


// Op rateurs de comparaison et logiques
EQUALS  : '=';
EQEQ    : '==';
NEQ     : '!=';
LT      : '<';
GT      : '>';
AND     : '&&';
OR      : '||';
EXCLAM  : '!';
```

Littéraux

Nombres

Les règles pour les nombres entiers et flottants :

```
fragment DIGIT          : '0'..'9';
fragment POSITIVE_DIGIT : '1' .. '9';
INT                     : '0' | (POSITIVE_DIGIT DIGIT*);

fragment NUM            : DIGIT+;
fragment SIGN           : '+' | '-' | ;
fragment EXP            : ('E' | 'e') SIGN NUM;
fragment DEC            : NUM '.' NUM;
fragment FLOATDEC       : (DEC EXP?) ('F' | 'f')?;
```

```
10 fragment DIGITHEX       : '0' .. '9' | 'A' .. 'F' | 'a' .. 'f';
11 fragment NUMHEX         : DIGITHEX+;
12 fragment FLOATHEX       : ('0x'|'0X') NUMHEX '.' NUMHEX ('P'|'p')
      SIGN NUM ('F' | 'f')?;
13 FLOAT                   : FLOATDEC | FLOATHEX;
```

Exemple : La chaîne `-123.45` sera analysée comme un littéral flottant valide.

Chaînes de caractères

Les chaînes incluent :

```
1 fragment STRING_CAR : ~('"'|'\\'|'\n');
2 STRING              : '"' (STRING_CAR | '\\"' | '\\\\')*  '"';
3 MULTI_LINE_STRING   : '"' (STRING_CAR | EOL |'\\"' | '\\\\')* '"';
```

Identifiants

Règle pour les identifiants :

```
1 fragment LETTER : 'a'  ..  'z' | 'A' .. 'Z';
2 IDENT           : (LETTER | '$' | '_') (LETTER | DIGIT | '$' | '_')
      *;
```

Commentaires

Gestion des commentaires mono-ligne et multi-lignes :

```
1 COMMENT : ( '//' .*? '\n'
2           | '/*' .*? '*/'
3           ) -> skip;
```

Inclusion de Fichiers

Le Lexer supporte la directive `#include`, facilitant l'intégration de bibliothèques :

```
1 fragment FILENAME   : (LETTER | DIGIT | '.' | '-' | '_')+;
2 INCLUDE             : '#include' (' ')* '"' FILENAME '"' {
3    doInclude(getText());
4 };
```

Note : Cette directive est essentielle pour inclure des définitions ou extensions dans Deca.

Remarques Importantes

- L'ordre des règles est crucial car ANTLR utilise le principe "premier arrivé, premier servi"
- Les mots-clés doivent être définis avant la règle des identifiants
- Les fragments sont des règles auxiliaires non accessibles au parser

- Les espaces et retours à la ligne sont ignorés via la règle EOL

Analyseur Syntaxique du Langage Deca

L'analyseur syntaxique (Parser) est responsable de l'analyse de la structure du programme Deca et de la construction de l'arbre abstrait. Il utilise les tokens fournis par l'analyseur lexical pour construire une représentation hiérarchique du programme.

Configuration du Parser

```
parser grammar DecaParser;

options {
    language = Java;
    superClass = AbstractDecaParser;
    tokenVocab = DecaLexer;
}
```

Le parser est configuré pour :

- Générer du code Java
- Hériter de la classe AbstractDecaParser
- Utiliser le vocabulaire défini dans DecaLexer

Structure Principale

Programme

La règle principale définit la structure d'un programme Deca :

```
prog returns[AbstractProgram tree]
    : list_classes main EOF {
            assert($list_classes.tree != null);
            assert($main.tree != null);
            $tree = new Program($list_classes.tree, $main.tree);
            setLocation($tree, $list_classes.start);
        }
    ;
```

Un programme est composé d'une liste de classes suivie d'un bloc principal (main).

Programme Principal

```
main returns[AbstractMain tree]
    : /* epsilon */ {
            $tree = new EmptyMain();
        }
    | block {
            assert($block.decls != null);
```

```
7               assert($block.insts != null);
8               $tree = new Main($block.decls, $block.insts);
9               setLocation($tree, $block.start);
10          }
11      ;
```

Le programme principal peut être vide ou contenir un bloc d'instructions.

Blocs et Déclarations

Structure d'un Bloc

Un bloc en Deca est une unité syntaxique fondamentale qui contient des déclarations de variables et des instructions. Il est défini comme suit :

```
1 block returns[ListDeclVar decls, ListInst insts]
2     : OBRACE list_decl list_inst CBRACE {
3               assert($list_decl.tree != null);
4               assert($list_inst.tree != null);
5               $decls = $list_decl.tree;
6               $insts = $list_inst.tree;
7           }
8     ;
```

Cette règle définit la structure d'un bloc qui :

- Est délimité par des accolades (`OBRACE` et `CBRACE`)
- Contient une liste de déclarations (`list_decl`)
- Contient une liste d'instructions (`list_inst`)

Listes de Déclarations

Les déclarations sont gérées par les règles suivantes :

```
1  list_decl returns[ListDeclVar tree]
2  @init   {
3               $tree = new ListDeclVar();
4           }
5      : decl_var_set[$tree]*
6      ;
7
8  decl_var_set[ListDeclVar l]
9      : type list_decl_var[$l,$type.tree] SEMI
10     ;
```

Instructions et Expressions

La règle `list_inst` gère les instructions :

```
list_inst returns[ListInst tree]
@init {
    $tree = new ListInst();
}
    : (inst {
        assert($inst.tree != null);
        $tree.add($inst.tree);
    })*
    ;
```

Exemple d'instruction :

```
inst returns[AbstractInst tree]
    : e1=expr SEMI {
            assert($e1.tree != null);
            $tree = $e1.tree;
            setLocation($tree, $SEMI);
        }
    | PRINT OPARENT list_expr CPARENT SEMI {
            assert($list_expr.tree != null);
            $tree = new Print(false, $list_expr.tree);
            setLocation($tree, $PRINT);
        }
    ;
```

Règles Avancées : Expressions et Assignations

```
list_expr returns[ListExpr tree]
@init   {
            $tree = new ListExpr();
        }
    : (e1=expr {
            assert($e1.tree != null);
            $tree.add($e1.tree);
        }
       (COMMA e2=expr {
            assert($e2.tree != null);
            $tree.add($e2.tree);
        }
       )* )?
    ;

expr returns[AbstractExpr tree]
    : assign_expr {
            assert($assign_expr.tree != null);
            $tree = $assign_expr.tree;
            setLocation($tree, $assign_expr.start);
        }
    ;

assign_expr returns[AbstractExpr tree]
    : e=or_expr (
```

```
26          /* condition: expression e must be a "LVALUE" */ {
27              if (! ($e.tree instanceof AbstractLValue)) {
28                  throw new InvalidLValue(this, $ctx);
29              }
30          }
31          EQUALS e2=assign_expr {
32              assert($e.tree != null);
33              assert($e2.tree != null);
34              $tree = new Assign((AbstractLValue)$e.tree, $e2.tree);
35              setLocation($tree, $EQUALS);
36          }
37        | /* epsilon */ {
38              assert($e.tree != null);
39              $tree = $e.tree;
40              setLocation($tree,$e.start);
41          }
42        )
43      ;
```

Les règles ci-dessus définissent la gestion des expressions complexes et des assignations avec des vérifications de validité supplémentaires pour garantir la cohérence syntaxique.

Règle : `type`

Cette règle traite les types dans le langage Deca. Elle retourne un identifiant correspondant au type défini.

```
1 type returns[AbstractIdentifier tree]
2     : ident {
3             assert($ident.tree != null);
4             $tree = $ident.tree;
5             setLocation($tree, $ident.start);
6         }
7     ;
```

Explications :

- La règle utilise la sous-règle `ident` pour extraire un identifiant valide.
- Elle vérifie que l'identifiant n'est pas nul et assigne sa localisation dans le code source.
- Retourne un `AbstractIdentifier`.

Règle : `literal`

Cette règle gère les valeurs littérales comme les entiers, flottants, chaînes, booléens, ainsi que les valeurs spéciales `null` et `this`.

```
1 literal returns[AbstractExpr tree]
2     : INT {
3           try {
4             $tree = new IntLiteral(Integer.parseInt($INT.text));
5             setLocation($tree, $INT);
```

```
6         } catch (NumberFormatException e) {
7             $tree = null;
8             throw new DecaRecognitionException("Invalid int : " +
    $INT.text +
9                 " is not valid (out of range or malformed)", this,
    $ctx);
10        }
11     }
12     | fd=FLOAT {
13        try {
14            $tree = new FloatLiteral(Float.parseFloat($fd.text));
15            setLocation($tree, $fd);
16        } catch (NumberFormatException e) {
17            $tree = null;
18            throw new DecaRecognitionException("Invalid float : " +
    $fd.text +
19                " is not valid (out of range or malformed)", this,
    $ctx);
20        }
21     }
22     | STRING {
23        String value = $STRING.text;
24        $tree = new StringLiteral(value.substring(1, value.length()
    - 1)
25              .replace("\\\"", "\"").replace("\\\\", "\\"));
26        setLocation($tree, $STRING);
27     }
28     | TRUE {
29        $tree = new BooleanLiteral(true);
30        setLocation($tree, $TRUE);
31     }
32     | FALSE {
33        $tree = new BooleanLiteral(false);
34        setLocation($tree, $FALSE);
35     }
36     | THIS {
37        $tree = new This();
38        setLocation($tree, $THIS);
39     }
40     | NULL {
41        $tree = new NullLiteral();
42        setLocation($tree, $NULL);
43     }
44     ;
```

Explications :

- Chaque branche correspond à un type de littéral (entier, flottant, chaîne, etc.).
- Les exceptions sont levées pour les valeurs hors des limites ou malformées.
- Retourne un `AbstractExpr`.

Règle : `ident`

Cette règle extrait et retourne un identifiant sous forme d'`AbstractIdentifier`.

```
ident returns[AbstractIdentifier tree]
    : IDENT {
            assert($IDENT != null);
            $tree = new Identifier(getDecacCompiler().createSymbol(
   $IDENT.text));
            setLocation($tree, $IDENT);
        }
    ;
```

Explications :

- Vérifie que l'identifiant est valide.
- Crée un symbole unique pour cet identifiant via le compilateur Deca.
- Retourne un `AbstractIdentifier`.

Liste des Classes

```
list_classes returns[ListDeclClass tree]
@init   {
           $tree = new ListDeclClass();
        }
    :
      (c1=class_decl {
              assert($c1.tree != null);
              $tree.add($c1.tree);
        }
      )*
    ;
```

Explications :

- Cette règle génère une liste de déclarations de classes (`ListDeclClass`).
- Chaque classe est extraite à l'aide de la règle `class_decl`.
- Les classes sont ajoutées de manière incrémentale à la liste.

Déclaration d'une Classe

```
class_decl returns[AbstractDeclClass tree]
    : CLASS name=ident superclass=class_extension OBRACE class_body
   CBRACE {
            assert($name.tree != null);
            $tree = new DeclClass($name.tree, $superclass.tree,
   $class_body.tree);
            setLocation($tree, $CLASS);
        }
    ;
```

Explications :

- La règle `class_decl` représente une classe avec :
  - Un nom de classe.
  - Une extension optionnelle.
  - Un corps de classe.
- Le corps de la classe est délimité par des accolades {}.

Extension de Classe

```
class_extension returns[AbstractIdentifier tree]
    : EXTENDS ident {
         assert($ident.tree != null);
         $tree = $ident.tree;
        }
    | /* epsilon */ {
            $tree = null;
        }
    ;
```

Explications :

- Cette règle permet d'ajouter une superclasse optionnelle à une classe.
- Si aucun mot-clé `EXTENDS` n'est présent, l'extension est considérée comme nulle.

Corps de Classe

```
class_body returns[ListDeclClassBody tree]
@init {
    $tree = new ListDeclClassBody();
    ListDeclField fields = new ListDeclField();
    ListDeclMethod methods = new ListDeclMethod();
}
    : (m=decl_method {
        assert($m.tree != null);
        $tree.add($m.tree);
        }
      | f=decl_field_set{
        assert($f.tree != null);
        $tree.add($f.tree);
      }
      )*
    ;
```

Explications :

- Le corps de classe peut contenir :
  - Des déclarations de méthodes (`decl_method`).

- Des ensembles de champs (`decl_field_set`).

- Les éléments sont ajoutés au fur et à mesure dans la liste `ListDeclClassBody`.

Déclaration des Champs et des Méthodes

Ensemble de Champs

```
1 decl_field_set returns[AbstractDeclClassBody tree]
2     : v=visibility t=type list_decl_field SEMI {
3         assert($v.tree != null);
4         assert($t.tree != null);
5         assert($list_decl_field.tree != null);
6         $tree = new DeclFieldSet($v.tree, $t.tree, $list_decl_field.
      tree);
7      }
8     ;
```

Explications :

- Chaque champ possède une visibilité (`visibility`), un type (`type`), et une ou plusieurs déclarations de champs (`list_decl_field`).

- Les informations sont encapsulées dans une instance de `DeclFieldSet`.

- Les champs sont terminés par un point-virgule (`SEMI`).

Visibilité des Champs

```
1 visibility returns[Visibility tree]
2     : /* epsilon */ {
3     $tree = Visibility.PUBLIC;
4         }
5     | PROTECTED {
6         $tree = Visibility.PROTECTED;
7         }
8     ;
```

Explications :

- La visibilité par défaut est `PUBLIC`.

- Le mot-clé `PROTECTED` peut être utilisé pour indiquer une visibilité protégée.

Liste de Champs

```
1 list_decl_field returns[ListDeclField tree]
2 @init {
3     $tree = new ListDeclField();
4 }
5     : dv1=decl_field {
6             assert($dv1.tree != null);
7             $tree.add($dv1.tree);
```

```
8     }
9         (COMMA dv2=decl_field {
10                assert($dv2.tree != null);
11                $tree.add($dv2.tree);
12        }
13      )*
14    ;
```

Explications :

- Plusieurs champs peuvent être déclarés simultanément, séparés par des virgules (`COMMA`).
- Chaque champ est extrait à l'aide de la règle `decl_field`.

Déclaration d'un Champ

```
1 decl_field returns[AbstractDeclField tree]
2     : i=type j=ident{
3             assert($i.tree != null);
4             $tree = new DeclField($i.tree, $j.tree, new
    NoInitialization());
5             setLocation($tree, $i.start);
6         }
7       (EQUALS e=expr {
8               assert($e.tree != null);
9               $tree = new DeclField($i.tree, $j.tree, new
    Initialization($e.tree));
10              setLocation($tree, $i.start);
11        }
12      )?;
```

Explications :

- Chaque champ est défini par un type et un identifiant.
- Une initialisation optionnelle peut être ajoutée avec le symbole `EQUALS`.

Déclaration d'une Méthode

```
1 decl_method returns[AbstractDeclMethod tree]
2 @init {
3     ListParam params = new ListParam();
4     AbstractMethodBody body = null;
5 }
6     : type ident OPARENT params=list_params CPARENT (block {
7                 assert($block.decls != null);
8                 assert($block.insts != null);
9                 body = new MethodBody($block.decls, $block.insts);
10                setLocation(body, $block.start);
11        }
12      | ASM OPARENT code=multi_line_string CPARENT SEMI {
13                body = new AsmMethodBody($code.text);
```

```
14                  setLocation(body, $code.start);
15          }
16        ) {
17          assert($type.tree != null);
18          assert($ident.tree != null);
19          $tree = new DeclMethod($type.tree, $ident.tree, params, body
    );
20          setLocation($tree, $ident.start);
21          }
22      ;
```

Explications :

- Une méthode est composée d'un type de retour, d'un identifiant et d'une liste de paramètres.
- Le corps de la méthode peut être un bloc standard ou un bloc assembleur (ASM).
- La règle retourne un `DeclMethod`.

Liste des Paramètres

```
1  list_params returns[ListParam tree]
2  @init {
3      $tree = new ListParam();
4  }
5      : (p1=param {
6             assert($p1.tree != null);
7             $tree.add($p1.tree);
8          } (COMMA p2=param {
9           assert($p2.tree != null);
10          $tree.add($p2.tree);
11         }
12       )*)?
13     ;
```

Explications :

- La règle extrait une liste de paramètres .
- Chaque paramètre est obtenu à l'aide de la règle `param`.
- Les paramètres sont séparés par des virgules.

Chaînes Multilignes

```
1  multi_line_string returns[String text, Location location]
2      : s=STRING {
3              assert($s != null);
4              $text = $s.text;
5              $location = tokenLocation($s);
6          }
```

```
7     | s=MULTI_LINE_STRING {
8             assert($s != null);
9             $text = $s.text;
10            $location = tokenLocation($s);
11        }
12    ;
```

Explications :

- La règle traite les chaînes simples et les chaînes multilignes.
- Elle retourne le texte de la chaîne et sa localisation dans le code source.

Déclaration d'un Paramètre

```
1 param returns[AbstractDeclParam tree]
2     : type ident {
3             assert($type.tree != null);
4             assert($ident.tree != null);
5             $tree = new DeclParam($type.tree, $ident.tree);
6             setLocation($tree, $ident.start);
7         }
8     ;
```

Explications :

Conclusion

Cette documentation présente les principales règles de l'analyseur syntaxique Deca. En suivant cette structure et en s'appuyant sur des outils comme ANTLR, il est possible de construire et valider efficacement le parser du compilateur Deca.

Ces règles définissent la structure des classes dans le langage Deca, en intégrant la gestion des superclasses, des champs et des méthodes. Elles permettent de construire l'arbre syntaxique correspondant à une hiérarchie de classes.

Gestion des Erreurs

Le parser inclut des vérifications systématiques pour garantir la cohérence de la grammaire. Par exemple :

- `assert($list_decl.tree ≠ null)` : Valide la présence d'une liste de déclarations.
- `assert($inst.tree ≠ null)` : Vérifie la validité d'une instruction.

Tests et Validation

Pour tester le parser, utilisez les outils fournis :

- `test_synt <fichier.deca>` pour vérifier la syntaxe.
- Scripts de non-régression pour assurer la conformité aux spécifications.

Vérification contextuelle et décoration de l'arbre

Chaîne d'éxecution de la vérification contextuelle

La vérification de l'arbre abstrait se fait à partir de l'appel à la méthode `verifyProgram`. La chaîne de vérification débute. Un appel est fait aux méthodes `verifyClasses` et `verifyMain` sur les attributs `classes` et `main` de l'instance de `Program`. L'arbre est ainsi parcouru.

Deca sans-objet

La vérification pour le main, (et donc à fortiori le Deca sans-objet) se fait en une seule passe. On vérifie la liste des déclarations des variables, et ensuite la liste des instructions.

Vérification de la liste des déclarations des variables

Pour chaque `DeclVar` de `ListDeclVar`, `verifyDeclVar` est invoqué. La méthode `verifyDeclVar` s'occupe ensuite de ce qui suit:
Vérification de l'identifieur de type :

```
1     // Step 1: Verify the type of the variable (e.g., 'int', 'float
          ')
2     Type t1 = type.verifyType(compiler);
3
4     // Ensure the type is valid for a variable declaration
5     if (!t1.isBoolean() && !t1.isInt() && !t1.isFloat() && !t1.
          isString()) {
6         throw new ContextualError("Invalid type for variable
              declaration: " + t1, this.getLocation());
7     }
```

Remarque : l'appel à la méthode `newGlobalVar` de la classe `DecacCompiler` incrémente l'attribut `globalVarCount` de cette même classe, cet attriut sera utilisé lors de la génération de code pour gérer les éventuels débordements de piles (Cela sera abordé avec en détails dans la section de génération de code).

Création et affectation de la définition de variable dans l'environnement actuel (en l'occurence ici, l'environnement global) dans le cas où celle-ci n'a pas déjà été déclarée. Une erreur adéquate est produit au cas échéant :

```
1     // No previous decalaration -> Create a VariableDefinition for
          this variable and
2     // associate it with the name
3     this.varDef = new VariableDefinition(t1, varName.getLocation());
4     varName.setType(t1);
5     varName.setDefinition(this.varDef);
6
7     // Declare the variable in the local environment
8     try {
9         localEnv.declare(varName.getName(), this.varDef);
```

```
10      } catch (EnvironmentExp.DoubleDefException e) {
11          throw new ContextualError(
12                  "Variable '" + varName.getName() + "' is already
                       defined in the environement of expressions",
13                  varName.getLocation());
14      }
```

Vérification de l'expression représentant le nom de variable(le mécanisme de vérification de `AbstractExpr` et des classes qui en dérivent sera détaillé ultérieurement) :

```
1       // verify variable name is valid
2       varName.verifyExpr(compiler, localEnv, currentClass);
```

Vérification d'une éventuelle initialisation de la variable, et gestion d'une variable balise de l'état d'initialisation (ou non) de la variable dans sa `VariableDefintion` (cela nous permettra de lever un warning lorsqu'une variable non initialisée, et pour laquelle aucune valeur n'est assignée, est utilisée) :

```
1       // Verify the initialization expression (both Init and NoInit)
2       initialization.verifyInitialization(compiler, t1, localEnv,
           currentClass);
3       if (initialization instanceof Initialization) {
4           varName.getVariableDefinition().setInitializationStatus(true
               );
5       }
```

Vérification d'`AbstractExpr`

La classe `AbstractExpr` est parent de plusieurs classes, dont on distinguera trois catégories différentes : les littéraux (`*Lieral`), les identifieurs (`Identifier`), et les opérations, qu'elles soient unaires, binaires ou de comparaison. (il y a en fait une autre catégorie, celle des mots clé réservés du langage Deca, qu'on abordera ultérieurement).

Les littéraux

Ces littéraux implémentent chacun la méthode `verifyExpr`, qui ne font que renvoyer le type du littéral (e.x `compiler.environmentType.INT` pour `IntLiteral`)

Les identifieurs

Un nouvel identifieur est créé à chaque fois que le parser en détecte un, il est cependant nécessaire de préserver l'unicité de la relation entre le symbole de l'identifier (son "nom") et la définition initialisée à sa déclaration. La méthode `declare` de la classe `EnvironmentExp` s'en charge (c'est d'ailleurs ici qu'on se charge de la gestion de l'erreur de définition double, comme on garantit l'unicité du symbole):

```
1   private final Map<Symbol, ExpDefinition> envExpr;
2
3   [...]
4   public void declare(Symbol name, ExpDefinition def) throws
       DoubleDefException {
5       if (envExpr.containsKey(name)) {
```

```
6           throw new DoubleDefException("Symbol already used in the
                environment of expressions: " + name);
7       }
8       envExpr.put(name, def);
9   }
```

Ainsi, la méthode `verifyExpr` de la classe `Identifier` fait correspondre la définition appropriée et se l'attribue. Elle fait cela pour le type également :

```
1   @Override
2   public Type verifyExpr(DecacCompiler compiler, EnvironmentExp
        localEnv,
3           ClassDefinition currentClass) throws ContextualError {
4       if (this.name == null) {
5           throw new ContextualError("The identifier's name cannot be
                null.", getLocation());
6       }
7
8       setDefinition(localEnv.get(getName(), getLocation()));
9       setType(getDefinition().getType());
10
11      return getType();
12  }
```

Vérification des opérations (opérations arithmétiques prises comme exemple)

La vérification des opération arithmétiques est effectuée directement dans la classe parent `AbstractOpArith`. On vérifie, "récursivement", l'opérande de droite de l'expression, puis l'opérande gauche. On atteint toujours un identifieur ou un littéral, qui renvoient leurs types, il s'en suit l'attribution des types adéquats (avec prise en compte des conversions implicites d'entiers en flottants par ajout des nodes `ConvFloat` lorsque cela est nécessaire) :

```
1   Type leftType = getLeftOperand().verifyExpr(compiler, localEnv,
        currentClass);
2   Type rightType = getRightOperand().verifyExpr(compiler, localEnv,
        currentClass);
3
4   // Check type compatibility between the left and right operands
5   if (leftType.isInt() && rightType.isInt()) {
6       this.setType(compiler.environmentType.INT);
7       return compiler.environmentType.INT;
8   } else if (leftType.isFloat() && rightType.isInt()) {
9       compiler.setPossibleOverflow();
10      this.setType(compiler.environmentType.FLOAT);
11      setRightOperand(new ConvFloat(getRightOperand()));
12      return compiler.environmentType.FLOAT;
13  } else if (leftType.isInt() && rightType.isFloat()) {
14      compiler.setPossibleOverflow();
15      setLeftOperand(new ConvFloat(getLeftOperand()));
16      this.setType(compiler.environmentType.FLOAT);
17      return compiler.environmentType.FLOAT;
```

```
} else if (leftType.isFloat() && rightType.isFloat()) {
    compiler.setPossibleOverflow();
    this.setType(compiler.environmentType.FLOAT);
    return compiler.environmentType.FLOAT;
} else {
    throw new ContextualError("Incompatible types for binary
       operation: "
            + leftType + " and " + rightType, this.getLocation());
}
```

On gère ici l'erreur d'incompatibilité des types des opérandes des opérations arithmétiques. Les vérifications des opérations logiques et des comparaisons se font de façon analogue.

Remarque : Ici on remarque aussi un appel à la méthode `setPossibleOverflow` de la classe `DecacCompiler`. On notifie le compilateur de la possibile présence d'une erreur overflow sur les opérations arithmétiques lorsque cela est nécessaire. Nous nous pencherons sur la gestion des erreurs runtime des programmes compilés en détails dans la partie de génération de code.

Vérification de `ListInst`

Similairement à `ListDeclVar`, `ListInst` dispose de la méthode `verifyListInst` qui invoque `verifyInst` pour chacune des instances d'`AbstractInst` concernées. **Remarque :** la classe `AbstractExpr` dispose en fait elle aussi d'une méthode `AbstractInst`. Cette dernière ne fait cependant qu'appeler la méthode `VerifyExpr`.

Vérification d'`AbstractPrint`

La vérification des instructions `Print` et `Println` se fait dans la classe parent `AbstractPrint`. Elle s'occupe simplement de vérifier que chacune des `AbstractExpr` de son attribut `ListExpr arguments` est d'un type accepté par ces méthodes, elle lève une erreur appropriée sinon :

```
    getArguments().verifyListExpr(compiler, localEnv, currentClass,
       returnType);

    for (AbstractExpr expr : getArguments().getList()) {
        Type exprType = expr.getType();
        if (!exprType.isFloat() && !exprType.isInt() && !exprType.
           isString()) {
            String errorMessage = String.format(
                    "print%s%s: invalid argument type '%s', expected
                        'int', 'float', or 'string'",
                    this.getSuffix(), this.getPrintHex() ? "x" : "",
                        expr.getType().toString());
            throw new ContextualError(errorMessage, expr.getLocation
               ());
        }
    }
```

Vérification de `IfThenElse`, `While`, `NoOperation`

Comme son nom l'indique, la vérification de `NoOperation` ne fait rien. Celles de `IfThenElse` et `While` vérifient, quant à elles, que leurs conditions sont bien des booléens, et invoque `verifyListInst` pour les instructions des blocs if/while. Un choix d'implémentation additionnel a été effectué au niveau de la classe `IfThenElse` : on représente les branches else if par une `Arraylist` d'instances de classes `IfThenElse` (dont l'attribut `elseBranch` est une instance de `IfThenElse` qui n'aura, par construction, aucune instruction).

```
1     @Override
2     protected void verifyInst(DecacCompiler compiler, EnvironmentExp
          localEnv,
3             ClassDefinition currentClass, Type returnType)
4             throws ContextualError {
5         Type condType = this.condition.verifyExpr(compiler, localEnv
              , currentClass);
6         this.condition.setType(condType);
7         this.condition.verifyCondition(compiler, localEnv,
              currentClass);
8         for (IfThenElse elseIf : elseIfBranches) {
9             elseIf.verifyInst(compiler, localEnv, currentClass,
                  returnType);
10        }
11        this.thenBranch.verifyListInst(compiler, localEnv,
              currentClass, returnType);
12        this.elseBranch.verifyListInst(compiler, localEnv,
              currentClass, returnType);
13    }
```

La vérification de `While` est analogue.

Génération de code assembleur IMA

Les méthodes de génération de code principales sont les `genCodeInst` et `genCodePrint`. Similairement à l'étape de vérification contextuelle et décoration de l'arbre abstrait, cela est fait en deux passes (la génération de code pour le langage Deca sans-objet se fait, elle, en une seule passe).
Encore une fois, l'arbre est parcouru, et chaque méthode `codeGenInst` de chaque node de l'arbre est appelée.

La gestion des registres suit une politique de registres disponibles (unused) et indisponibles (used). Pendant l'évaluation d'une expression, des registres sont "alloués" à la demande, et la pile est utilisée si aucun registre n'est disponible. Ces registres sont libérés adéquatement. (Lorsque par exemple une expression arithmétique est évaluée, elle est stockée dans le registre de l'une de ses opérandes, l'autre registre est donc "libéré").

WIP