INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Projet de Grammaires et Langages

Développement d'un interpréteur pour le langage lutin

H4112

Chef de projet : Raphaël SAUGET

Hugo BARATTOLO

Alexis FOSSART

Gaëtan GOUZI

Yassine HARZALLAH

Valentin LUNGENSTRASS

Maxence SOUCHON

Document de Conception

I/ Grammaire

PROG → PD PI
PD → PD D; | ε
D → var LID | const LIDV
LID → LID, id | id
LIDV → LIDV, id = val | id = val
PI → PI I ; | ε
I → lire id | ecrire E | id := E
E → E opa T | T
T → T opm F | F
F → (E) | id | val

PD = Partie Déclarative
PI = Partie Instructions
LID = Liste d'IDentifiants
LIDV = Liste d'IDentifiants avec Valeurs

On développe la grammaire pour pouvoir numéroter les différentes règles.

R0: PROG → PD PI
R1: PD → PD D;
R2: PD → ε
R3: D → var LID
R4: D → const LIDV
R5: LID → LID, id
R6: LID → id
R7: LIDV → LIDV, id = val
R8: LIDV → id = val
R9: PI → PI I ;
R10: PI → ε
R11: I → lire id
R12: I → ecrire E
R13: I → id := E
R14: E → E opa T
R15: E → T
R16: T → T opm F
R17: T → F
R18: F → (E)
R19: F → id
R20: F → val

II/ Expressions régulières

,	^\s*(,)
;	^\s*(;)
:=	^\s*(:=)
=	^\s*(=)
)	^\s*(\))
(	^\s*(\()
opa	^\s*([+-])
opm	^\s*([*/])
var	^\s*(var)\s
const	^\s*(const)\s
lire	^\s*(lire)\s
ecrire	^\s*(ecrire)\s
id	^\s*([a-zA-Z][a-zA-Z0-9]*)
val	^\s*([0-9]+)

Les expressions régulières commencent toutes par ^\s* pour trouver les motifs correspondants quelque soit le nombre de blancs(espace ou tabulations) situés avant.

Afin ne pas confondre un mot clé avec une variable dont le nom commence de la même manière, les expressions régulières des mots clés terminent par un blanc.
Exemple:

```
var varicelle;
```

Ainsi, l'expression régulière ^\s*(var) va trouver les motifs:

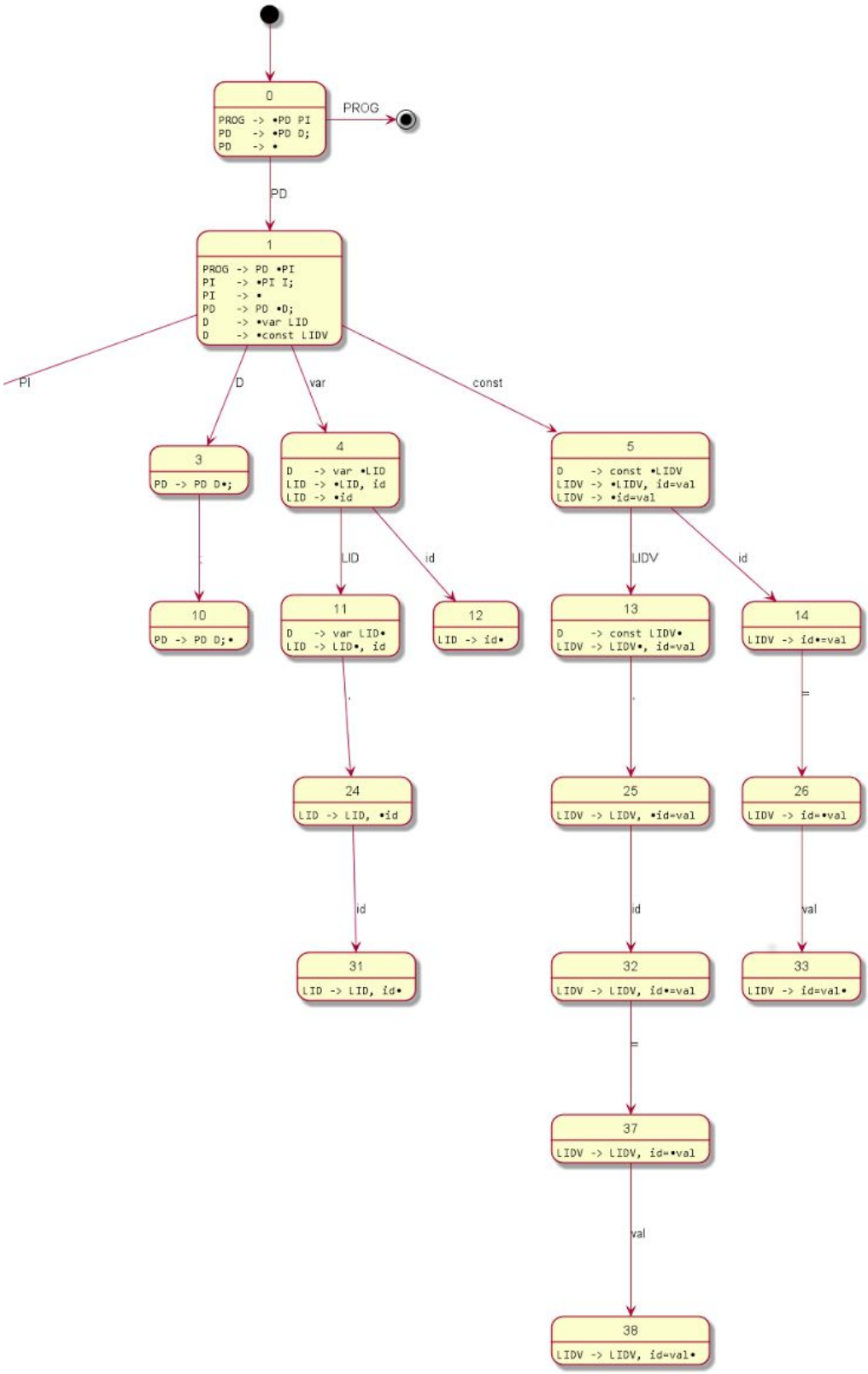
```
var varicelle;
```

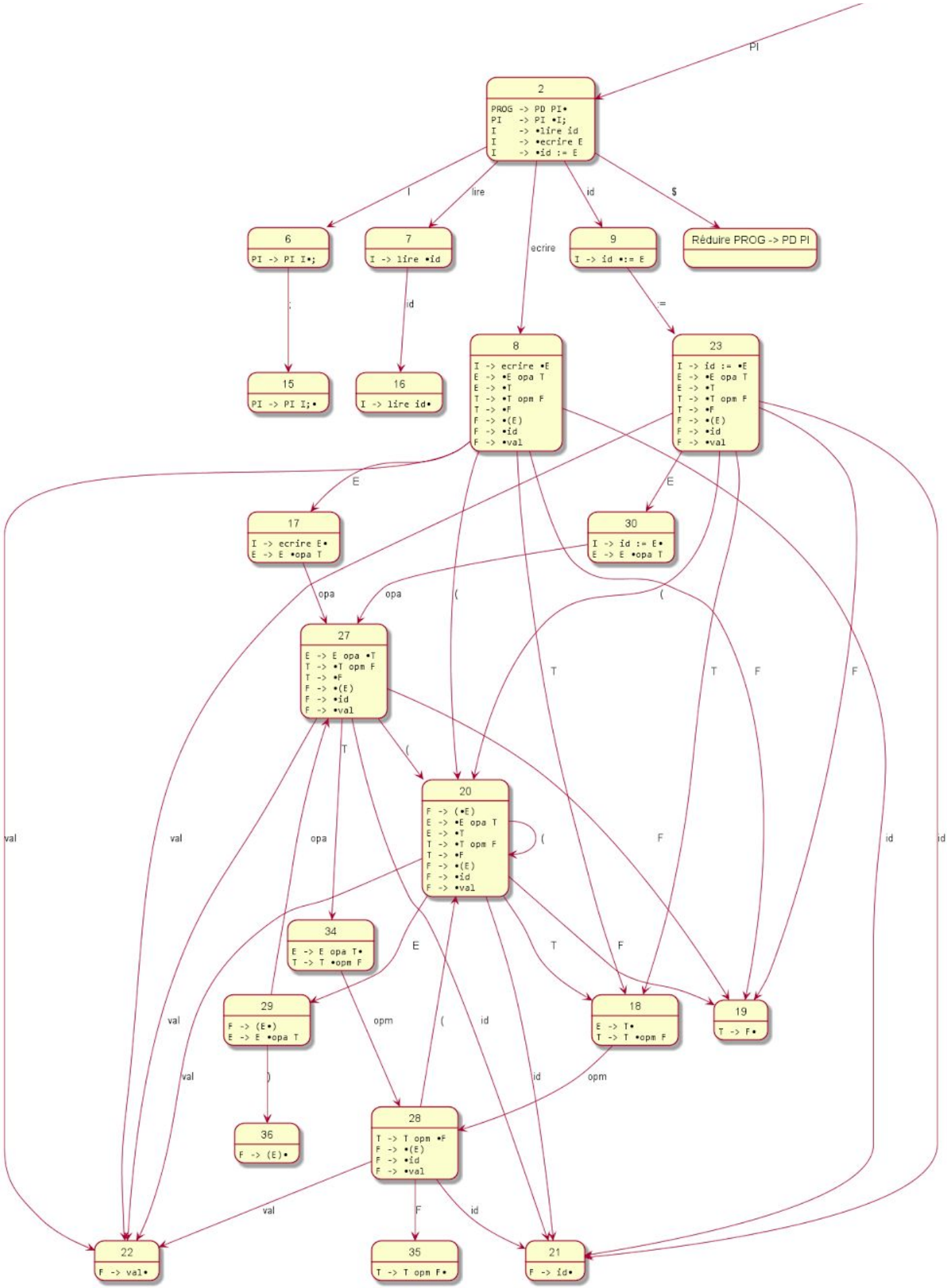
Alors que l'expression régulière ^\s*(var)\s va trouver le motif:

```
var varicelle;
```

De cette manière, il est possible de nommer une variable d'un nom commençant de la même manière qu'un mot-clé.

III/ Automate

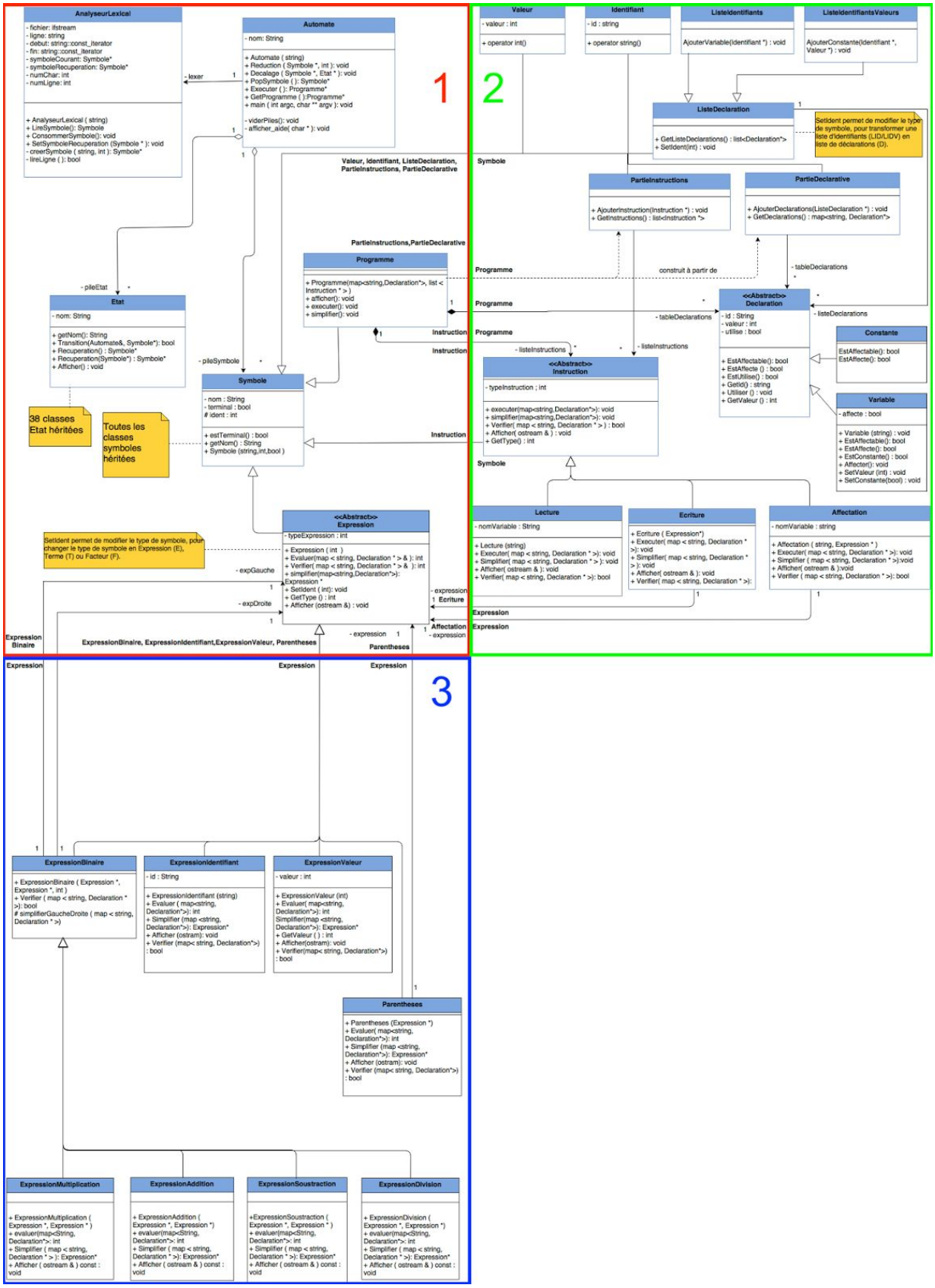


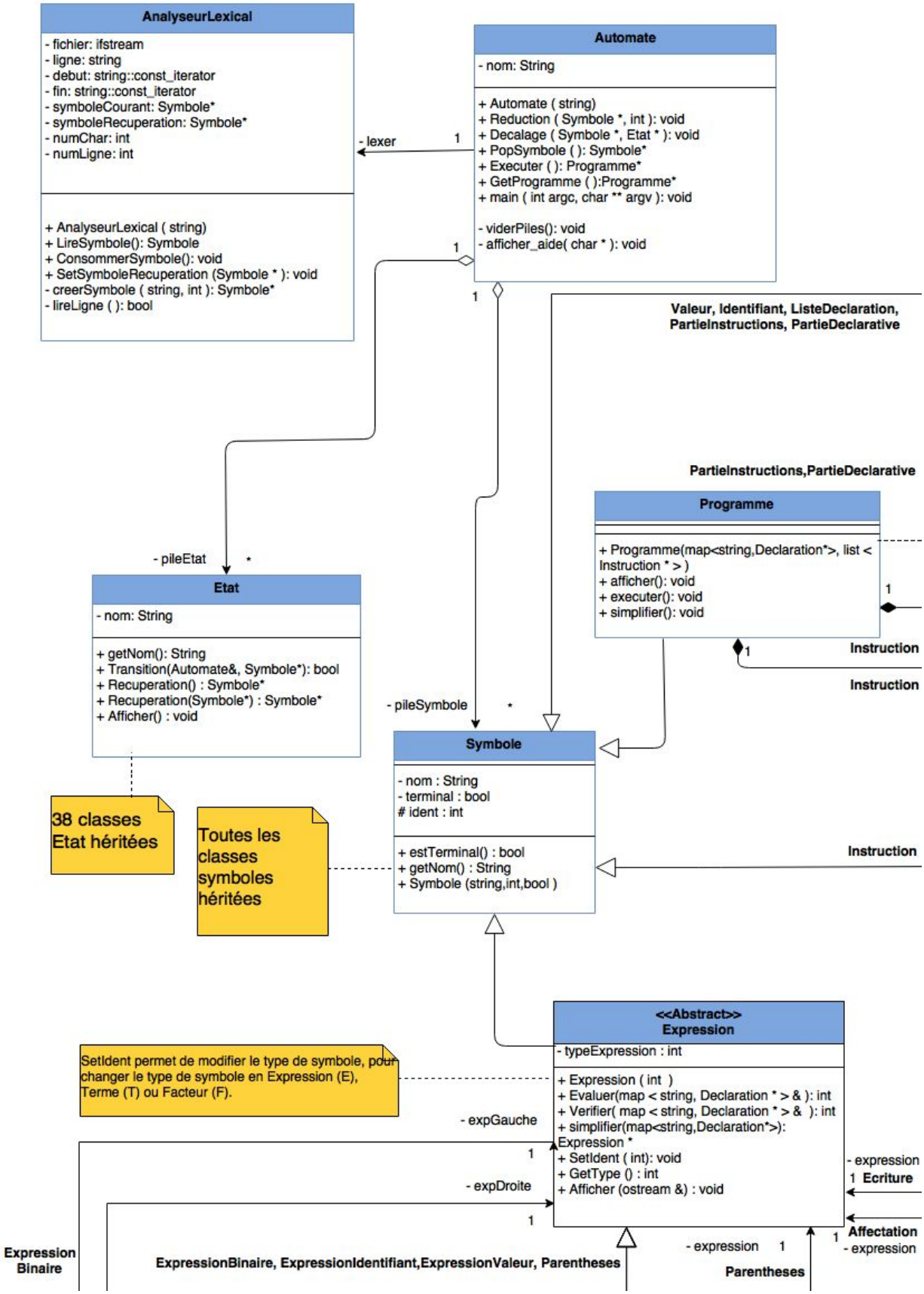

PI
2
PROG -> PD PI•
PI -> PI •I;
I -> •lire id
I -> •ecrire E
I -> •id := E
I
lire
id
$
6
PI -> PI I•;
7
I -> lire •id
ecrire
9
I -> id •:= E
Réduire PROG -> PD PI
;
id
=
15
PI -> PI I;•
16
I -> lire id•
8
I -> ecrire •E
E -> •E opa T
E -> •T
T -> •T opm F
T -> •F
F -> •(E)
F -> •id
F -> •val
23
I -> id := •E
E -> •E opa T
E -> •T
T -> •T opm F
T -> •F
F -> •(E)
F -> •id
F -> •val
E
E
17
I -> ecrire E•
E -> E •opa T
30
I -> id := E•
E -> E •opa T
opa
opa
(
(
27
E -> E opa •T
T -> •T opm F
T -> •F
F -> •(E)
F -> •id
F -> •val
T
T
F
F
T
(
20
F -> (•E)
E -> •E opa T
E -> •T
T -> •T opm F
T -> •F
F -> •(E)
F -> •id
F -> •val
(
val
val
opa
F
id
id
34
E -> E opa T•
T -> T •opm F
E
T
F
29
F -> (E•)
E -> E •opa T
18
E -> T•
T -> T •opm F
19
T -> F•
val
opm
(
id
val
)
id
opm
36
F -> (E)•
28
T -> T opm •F
F -> •(E)
F -> •id
F -> •val
val
F
id
22
F -> val•
35
T -> T opm F•
21
F -> id•

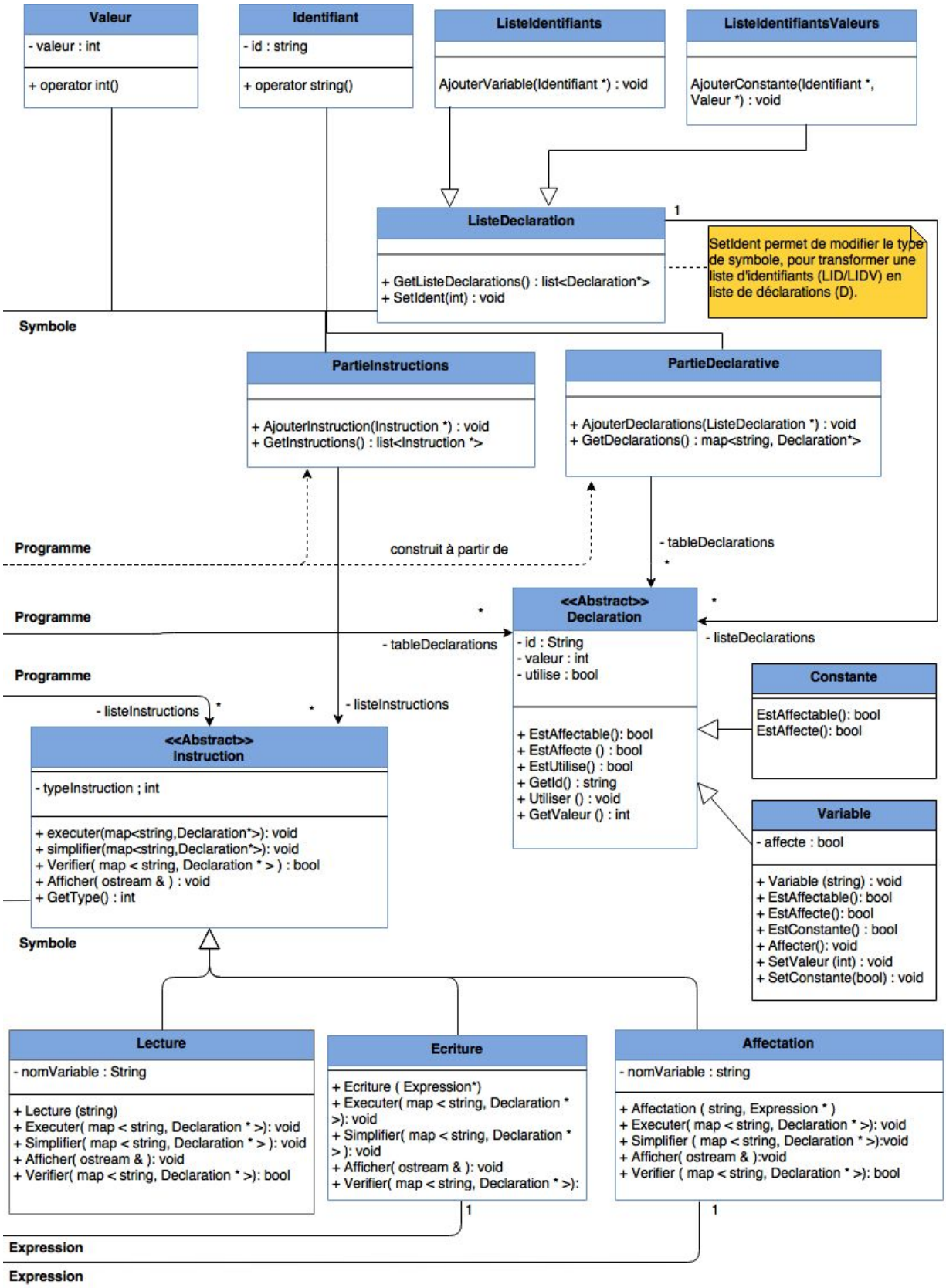
IV/ Table d'analyse

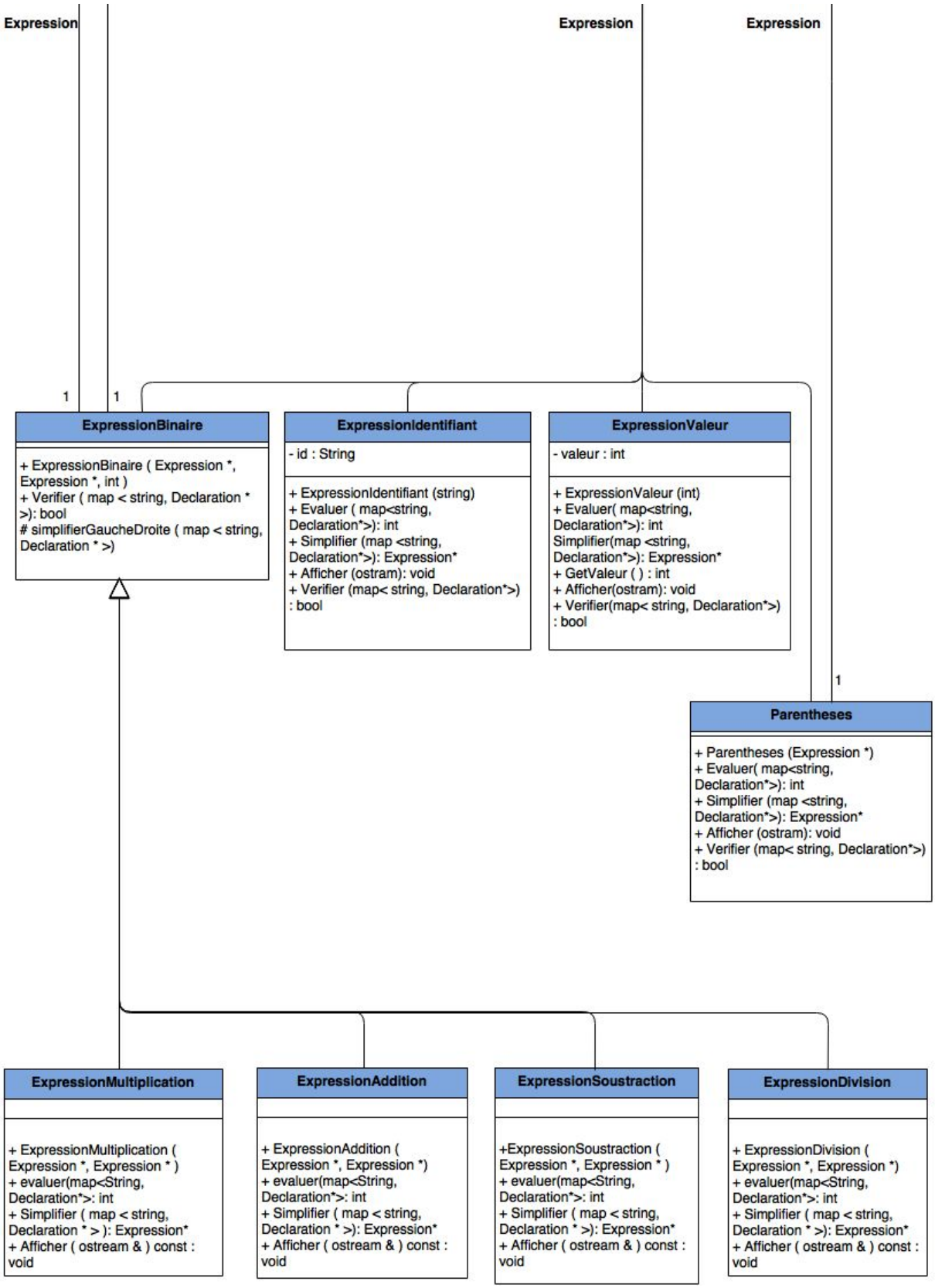
	var	const	lire	ecrire	,	;	:=	=	(	)	id	val	opa	opm	$	PROG	PD	D	LID	LIDV	PI	I	E	T	F
E0	R2	R2	R2	R2							R2				R2	A	E1								
E1	E4	E5	R10	R10							R10				R10			E3			E2				
E2			E7	E8							E9				R0							E6			
E3						E10																			
E4											E12								E11						
E5											E14									E13					
E6						E15																			
E7											E16														
E8									E20		E21	E22											E17	E18	E19
E9							E23																		
E10	R1	R1	R1	R1							R1				R1										
E11					E24	R3																			
E12					R6	R6																			
E13					E25	R4																			
E14								E26																	
E15			R9	R9							R9				R9										
E16						R11																			
E17						R12							E27												
E18						R15				R15			R15	E28											
E19						R17				R17			R17	R17											
E20									E20		E21	E22											E29	E18	E19
E21						R19				R19			R19	R19											
E22						R20				R20			R20	R20											
E23									E20		E21	E22											E30	E18	E19
E24											E31														
E25											E32														
E26												E33													
E27									E20		E21	E22												E34	E19
E28									E20		E21	E22													E35
E29										E36			E27												
E30						R13							E27												
E31					R5	R5																			
E32								E37																	
E33					R8	R8																			
E34						R14				R14			R14	E28											
E35						R16				R16			R16	R16											
E36						R18				R18			R18	R18											
E37												E38													
E38					R7	R7																			

V/ Diagramme de classes




AnalyseurLexical
- fichier: ifstream
- ligne: string
- debut: string::const_iterator
- fin: string::const_iterator
- symboleCourant: Symbole*
- symboleRecuperation: Symbole*
- numChar: int
- numLigne: int
+ AnalyseurLexical (string)
+ LireSymbole(): Symbole
+ ConsommerSymbole(): void
+ SetSymboleRecuperation (Symbole *): void
- creerSymbole (string, int): Symbole*
- lireLigne ( ): bool
- lexer
1
Automate
- nom: String
+ Automate (string)
+ Reduction (Symbole *, int): void
+ Decalage (Symbole *, Etat *): void
+ PopSymbole ( ): Symbole*
+ Executer ( ): Programme*
+ GetProgramme ( ):Programme*
+ main (int argc, char ** argv): void
- viderPiles(): void
- afficher_aide(char *): void
1
1
Valeur, Identifiant, ListeDeclaration,
PartieInstructions, PartieDeclarative
PartieInstructions,PartieDeclarative
Programme
+ Programme(map<string,Declaration*>, list <
Instruction * >)
+ afficher(): void
+ executer(): void
+ simplifier(): void
1
1
Instruction
Instruction
- pileEtat
*
Etat
- nom: String
+ getNom(): String
+ Transition(Automate&, Symbole*): bool
+ Recuperation() : Symbole*
+ Recuperation(Symbole*) : Symbole*
+ Afficher() : void
- pileSymbole
*
Symbole
- nom : String
- terminal : bool
ident : int
+ estTerminal() : bool
+ getNom() : String
+ Symbole (string,int,bool)
Instruction
38 classes
Etat héritées
Toutes les
classes
symboles
héritées
<<Abstract>>
Expression
- typeExpression : int
+ Expression (int)
+ Evaluer(map < string, Declaration * > &): int
+ Verifier(map < string, Declaration * > &): int
+ simplifier(map<string,Declaration*>):
Expression *
+ SetIdent (int): void
+ GetType () : int
+ Afficher (ostream &) : void
SetIdent permet de modifier le type de symbole, pour
changer le type de symbole en Expression (E),
Terme (T) ou Facteur (F).
- expGauche
1
- expDroite
1
- expression
1 Ecriture
1
Affectation
- expression
Expression
Binaire
ExpressionBinaire, ExpressionIdentifiant,ExpressionValeur, Parentheses
- expression
1
Parentheses


Valeur
- valeur : int
+ operator int()
Identifiant
- id : string
+ operator string()
ListeIdentifiants
AjouterVariable(Identifiant *) : void
ListeIdentifiantsValeurs
AjouterConstante(Identifiant *, Valeur *) : void
ListeDeclaration
+ GetListeDeclarations() : list<Declaration*>
+ SetIdent(int) : void
1
SetIdent permet de modifier le type de symbole, pour transformer une liste d'identifiants (LID/LIDV) en liste de déclarations (D).
Symbole
PartieInstructions
+ AjouterInstruction(Instruction *) : void
+ GetInstructions() : list<Instruction *>
PartieDeclarative
+ AjouterDeclarations(ListeDeclaration *) : void
+ GetDeclarations() : map<string, Declaration*>
Programme
construit à partir de
- tableDeclarations
*
Programme
*
<<Abstract>>
Declaration
- id : String
- valeur : int
- utilise : bool
+ EstAffectable(): bool
+ EstAffecte () : bool
+ EstUtilise() : bool
+ GetId() : string
+ Utiliser () : void
+ GetValeur () : int
*
- listeDeclarations
- tableDeclarations
Programme
- listeInstructions
*
*
- listeInstructions
Constante
EstAffectable(): bool
EstAffecte(): bool
<<Abstract>>
Instruction
- typeInstruction ; int
+ executer(map<string,Declaration*>): void
+ simplifier(map<string,Declaration*>): void
+ Verifier(map < string, Declaration * >) : bool
+ Afficher(ostream &) : void
+ GetType() : int
Variable
- affecte : bool
+ Variable (string) : void
+ EstAffectable(): bool
+ EstAffecte(): bool
+ EstConstante() : bool
+ Affecter(): void
+ SetValeur (int) : void
+ SetConstante(bool) : void
Symbole
Lecture
- nomVariable : String
+ Lecture (string)
+ Executer(map < string, Declaration * >): void
+ Simplifier(map < string, Declaration * >): void
+ Afficher(ostream &): void
+ Verifier(map < string, Declaration * >): bool
Ecriture
+ Ecriture (Expression*)
+ Executer(map < string, Declaration * >): void
+ Simplifier(map < string, Declaration * >): void
+ Afficher(ostream &): void
+ Verifier(map < string, Declaration * >):
Affectation
- nomVariable : string
+ Affectation (string, Expression *)
+ Executer(map < string, Declaration * >): void
+ Simplifier (map < string, Declaration * >):void
+ Afficher(ostream &):void
+ Verifier (map < string, Declaration * >): bool
1
1
Expression
Expression


Expression
Expression
Expression
1
1
1
ExpressionBinaire
+ ExpressionBinaire (Expression *, Expression *, int)
+ Verifier (map < string, Declaration * >): bool
# simplifierGaucheDroite (map < string, Declaration * >)
ExpressionIdentifiant
- id : String
+ ExpressionIdentifiant (string)
+ Evaluer (map<string, Declaration*>): int
+ Simplifier (map <string, Declaration*>): Expression*
+ Afficher (ostram): void
+ Verifier (map< string, Declaration*>) : bool
ExpressionValeur
- valeur : int
+ ExpressionValeur (int)
+ Evaluer(map<string, Declaration*>): int
Simplifier(map <string, Declaration*>): Expression*
+ GetValeur ( ) : int
+ Afficher(ostram): void
+ Verifier(map< string, Declaration*>) : bool
Parentheses
+ Parentheses (Expression *)
+ Evaluer(map<string, Declaration*>): int
+ Simplifier (map <string, Declaration*>): Expression*
+ Afficher (ostram): void
+ Verifier (map< string, Declaration*>) : bool
ExpressionMultiplication
+ ExpressionMultiplication (Expression *, Expression *)
+ evaluer(map<String, Declaration*>: int
+ Simplifier (map < string, Declaration * >): Expression*
+ Afficher (ostream &) const : void
ExpressionAddition
+ ExpressionAddition (Expression *, Expression *)
+ evaluer(map<String, Declaration*>: int
+ Simplifier (map < string, Declaration * >): Expression*
+ Afficher (ostream &) const : void
ExpressionSoustraction
+ExpressionSoustraction (Expression *, Expression *)
+ evaluer(map<String, Declaration*>: int
+ Simplifier (map < string, Declaration * >): Expression*
+ Afficher (ostream &) const : void
ExpressionDivision
+ ExpressionDivision (Expression *, Expression *)
+ evaluer(map<String, Declaration*>: int
+ Simplifier (map < string, Declaration * >): Expression*
+ Afficher (ostream &) const : void