codeI = code intermédiaire

Appels de fonctions au lancement :
Classe Main.java (avec la table des fonctions et table des symboles)

- ➔ NasmGenerator.xtend
  - ◆ NasmGeneratorCodeI.xtend (compile le code source en codeI)
    - DefFunction.java (structure d'une fonction avec ses entrées, sorties, sa table des variables et son codeI)
      - Quadruplet.java (structure d'une ligne de codeI avec une opération, une adresse cible et deux adresses sources)
  - ◆ ameliorationCodeI (effectue la suppression de code mort)
  - ◆ NasmGeneratorCodeF.xtend (compile le code intermédiaire en nasm)

Tableau des différents opérateurs du codeI :

Format des opérations 3@				**Opération**
Opérateur	Cible (RBX)	Source1 (RSI)	Source2 (RDI)	
NOP	-	-	-	Ne fait rien
READ	V1	-	-	Lit un paramètre de la fonction dans V1
WRITE	-	V1	-	Écrit V1 dans une sortie de la fonction
CALL F1				Appelle la fonction F1
NIL	V1	-	-	Affecte nil à V1
SYMB S1	V1	-	-	Affecte S1 à V1
CONS	V1	V2	V3	Construit un arbre avec V2 et V3, et l'affecte à V1
HEAD	V1	V2	-	Affecte l'arbre (hd V2) à V1
TAIL	V1	V2	-	Affecte l'arbre (tl V2) à V1
TRUE	V1	-	-	Affecte (cons nil nil) à V1
IFNZ E1	-	V1	-	Si V1 non null, va à l'étiquette E1
IFNEQ E1	-	V1	V2	Si V1 != V2, va à l'étiquette E1 (comparaison en profondeur de deux arbres binaires)
LABEL E1	-	-	-	Créé l'étiquette E1
GOTO E1	-	-	-	Va à l'étiquette E1

Schéma de traduction code WHILE -> codeI :

En noir : les opérations du code intermédiaire
En bleu : le code intermédiaire des expressions et la place de la variable résultat
Les variables et étiquettes en **gras** sont nouvellement créées (et sont donc précédées d'un **V** = newVar ou **L** = newLab)

Opérations WHILE	**Opérations codeI**
read V1, V2, …, Vn	< READ - V1 - - > < READ - V2 - - > ... < READ - Vn - - >
write V1, …, Vn	< WRITE - - V1 - > ... < WRITE - - Vn - >
nop	< NOP - - - >
V1	si dans une expression booléenne : <IFNZ siVrai - - V1 - > <GOTO siFaux - - - >
V1, V2 := E1, E2	V3 = E1.codeI V4 = E2.codeI < WRITE - - V3 - > < WRITE - - V4 - > <READ - V1 - - > <READ - V2 - - >
(cons E2 … En-1 En)	Vn = En.codeI Vn-1 = En-1.codeI < CONS - **V10** - Vn-1 - Vn > … V2 = E2.codeI < CONS - V10 - V2 - V10 >
(list E2 … En-1 En)	< NIL - **V14** - - > Vn = En.codeI < CONS - **V15** - Vn - V14 > Vn-1 = En-1.codeI < CONS - V15 - Vn-1 - V15 > … V2 = E2.codeI < CONS - V15 - V2 - V15 >
(hd E1)	V1 = E1.codeI < HEAD - **V2** - V1 - >
(tl E1)	V1 = E1.codeI < TAIL - **V2** - V1 - >
not E1	E1.codeI(siFaux,siVrai)

Opérations WHILE	Opérations codel
E1 =? E2	V1 = E1.codel V2 = E2.codel <IFNEQ siFaux - - V1 - V2 > <GOTO siVrai - - - >
E1 and E2 and … and En	E1.codel(siVrai1,siFaux) <LABEL **siVrai1** - - - > … En-1.codel(siVrai12,siFaux) <LABEL **siVrai12** - - - > En.codel(siVrai,siFaux)
E1 or E2 or … or En	E1.codel(siVrai,siFaux1) <LABEL **siFaux1** - - - > … En-1.codel(siVrai,siFaux12) <LABEL **siFaux12** - - - > En.codel(siVrai,siFaux)
Si on a un “and”, “or”, “not” ou “=?” qui ne sont pas déjà dans une expression booléenne, on ajoute le code suivant après leur code intermédiaire :	<LABEL siVrai - - - > <TRUE - **res** - - > <GOTO **fin** - - - > <LABEL siFaux - - - > <NIL - res - - > <LABEL fin - - - >
(f1 E1 E2)	V1 = E1.codel V2 = E2.codel < WRITE - - V1 - > < WRITE - - V2 - > < CALL F1 - - - >
if E then C fi	E.codel(**siVrai**,**siFaux**) <LABEL siVrai - - - > C.codel <LABEL siFaux - - - >

if E then
C1
else
C2
fi
	E.codel(**siVrai**,**siFaux**)
	<LABEL siVrai - - - >
	C1.codel
	<GOTO **fi** - - - >
	<LABEL siFaux - - - >
	C2.codel
	<LABEL fi - - - >

Opérations WHILE	Opérations codel
while E do C od	<LABEL **boucle** - - - > E.codel(**siVrai**,**siFaux**) <LABEL siVrai - - - > C.codel <GOTO boucle - - - > <LABEL siFaux - - - >
for E do C od	**V1** = E.codel <WRITE - - V1 - > <READ - **nbBoucles** - - > <LABEL **boucle** - - - > <IFNZ **forBody** - - **nbBoucles** - > <GOTO **od** - - - > <LABEL forBody - - - > <TAIL - nbBoucles - nbBoucles - > C.codel <GOTO boucle - - - > <LABEL od - - - >
foreach V in E do C od	**V1** = E.codel <WRITE - - V1 - > <READ - **nbBoucles** - - > <LABEL **boucle** - - - > <IFNZ **forBody** - - **nbBoucles** - > <GOTO **od** - - - > <LABEL forBody - - - > <HEAD - V - nbBoucles - > <TAIL - nbBoucles - nbBoucles - > C.codel <GOTO boucle - - - > <LABEL od - - - >

Schéma de traduction code codeI -> codeNASM :

Opérations codeI	Opérations codeNASM
< READ - V1 - - >	mov RSI,0 call read
< WRITE - - V1 - >	mov RSI,0 call write
< NOP - - - >	nop
< NIL - V2 - - >	mov [R8+8],R15

< CONS - V1 - V2 - V3 >	mov RSI,[R8+16]
	mov RDI,[R8+8]
	call cons
	mov [R8+0],RBX

< HEAD - V2 - V1 - >	mov RSI,[R8+0]
	call head
	mov [R8+0],RBX

< TAIL - V2 - V1 - >	mov RSI,[R8+0]
	call tail
	mov [R8+0],RBX

< GOTO siVrai - - - >	jmp .siVrai
< LABELsiVrai - - - >	.siVrai:
< IFNZ siFaux - - V2 - >	cmp R15,[R8+8] jnz .siFaux

< IFNEQ siFaux - - V1 -V2 >	mov RSI,[R8+8]
	mov RDI,[R8+0]
	call equals
	cmp RAX,0
	je .siFaux

< TRUE - V2 - - >	mov [R8+8],R14

< SYMB S4 - V2 - - >	mov R12,48
	add R12,symbols_desc
	mov [R8+8],R12

< CALL F1 - - - >	call F1