Exercice

Modifier la fonction précédente pour renvoyer *a*+b+1.

@

```
define i32 @addition(i32 %a, i32 %b) #0 {
entry:
  %r = add nsw i32 %b, %a
  %result = add nsw i32 %r, 1
  ret i32 %result
}
```

Attention ! LLVM adopte une représentation SSA, les valeurs manipulées sont des constantes, donc non modifiables.

SSA : *Static Single Assignment*, https://fr.wikipedia.org/wiki/Static_single_assignment_form

Le code suivant est incorrect :

```
define i32 @addition(i32 %a, i32 %b) #0 {
entry:
  %result = add nsw i32 %b, %a
  %result = add nsw i32 %result, 1  <--- Erreur, %result déjà défini
  ret i32 %result
}
```

Exercice

Compiler en LLVM **sans optimisation** le code suivant :

```
#include <stdint.h>

uint32_t somme (uint32_t n) {
  uint32_t r = 0 ;
  while (n > 0) {
    r += n ;
    n-- ;
  }
  return r ;
}
```

Exercice : solution

```
#include <stdint.h>

uint32_t somme (uint32_t n){
  uint32_t r = 0 ;
  while (n > 0) {
    r += n ;
    n-- ;
  }
  return r ;
}
```

L'argument et r sont placés en mémoire

Expression `n > 0`

Instruction `r += n ;`

Instruction `n-- ;`

Instruction `return r ;`

```
define i32 @somme(i32 %n) #0 {
  %1 = alloca i32, align 4
  %r = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 0, i32* %r, align 4
  br label %2

; <label>:2 ; preds = %5, %0
  %3 = load i32* %1, align 4
  %4 = icmp ugt i32 %3, 0
  br i1 %4, label %5, label %11

; <label>:5 ; preds = %2
  %6 = load i32* %1, align 4
  %7 = load i32* %r, align 4
  %8 = add i32 %7, %6
  store i32 %8, i32* %r, align 4
  %9 = load i32* %1, align 4
  %10 = add i32 %9, -1
  store i32 %10, i32* %1, align 4
  br label %2

; <label>:11 ; preds = %2
  %12 = load i32* %r, align 4
  ret i32 %12
}
```

Exercice

Compiler en LLVM **avec optimisation** le code suivant :

```
#include <stdint.h>

uint32_t somme (uint32_t n) {
  uint32_t r = 0 ;
  while (n > 0) {
    r += n ;
    n-- ;
  }
  return r ;
}
```

Compiler en LLVM **avec optimisation** le code suivant :

Code obtenu

```
define i32 @somme(i32 %n) #0 {
  %1 = icmp eq i32 %n, 0
  br i1 %1, label %12, label %.lr.ph

.lr.ph:                                           ; preds = %0
  %2 = add i32 %n, -1
  %3 = mul i32 %2, %2
  %4 = zext i32 %2 to i33
  %5 = add i32 %n, -2
  %6 = zext i32 %5 to i33
  %7 = mul i33 %4, %6
  %8 = lshr i33 %7, 1
  %9 = trunc i33 %8 to i32
  %10 = add i32 %3, %n
  %11 = sub i32 %10, %9
  br label %12

; <label>:12                                      ; preds = %.lr.ph, %0
  %r.0.lcssa = phi i32 [ %11, %.lr.ph ], [ 0, %0 ]
  ret i32 %r.0.lcssa
}
```

Le code optimisé calcule la somme de n premiers nombres

```
.lr.ph:
; preds = %0
  %2 = add i32 %n, -1
  %3 = mul i32 %2, %2
  %4 = zext i32 %2 to i33
  %5 = add i32 %n, -2
  %6 = zext i32 %5 to i33
  %7 = mul i33 %4, %6
  %8 = lshr i33 %7, 1
  %9 = trunc i33 %8 to i32
  %10 = add i32 %3, %n
  %11 = sub i32 %10, %9
  br label %12
```

Code LLVM	Signification
%2 = add i32 %n, -1	%2 = n-1
%3 = mul i32 %2, %2	%3 = $(n-1)^2 = n^2-2n+1$
%4 = zext i32 %2 to i33	%4 = n-1
%5 = add i32 %n, -2	%5 = n-2
%6 = zext i32 %5 to i33	%6 = n-2
%7 = mul i33 %4, %6	%7 = $(n-2)(n-1) = n^2-3n+2$
%8 = lshr i33 %7, 1	%8 = $(n^2-3n+2) / 2$
%9 = trunc i33 %8 to i32	%9 = $(n^2-3n+2) / 2$
%10 = add i32 %3, %n	%10 = n^2-n+1
%11 = sub i32 %10, %9	%11 = $n^2-n+1 - (n^2-3n+2)/2 = n(n+1)/2$

Pourquoi LLVM engendre-t'il un code aussi compliqué ?