Université libre de Bruxelles

École Polytechnique de Bruxelles

M-IRIFS

INFO-F403 - Introduction to language theory and compiling - 2023/2024

PASCALmaispresque

Project – Part 3

Members :

Younes El Mokhtari

Ismail Jeq

Teacher :

Gilles GEERAERTS

3 janvier 2024

Table des matières

1 Description of the project

This project aims to build a compiler for PASCALMAISPRESQUE, a simple imperative language. In this third and last part, we generated LLVM code from the output of parser that corresponds to the semantics of the PASCALmaispresque program that is being compiled for the PASCALMAISPRESQUE grammar. The goal is to take in input the PASCALmaispresque program and output LLVM IR code while respecting the semantics of the grammar.

1.1 AST

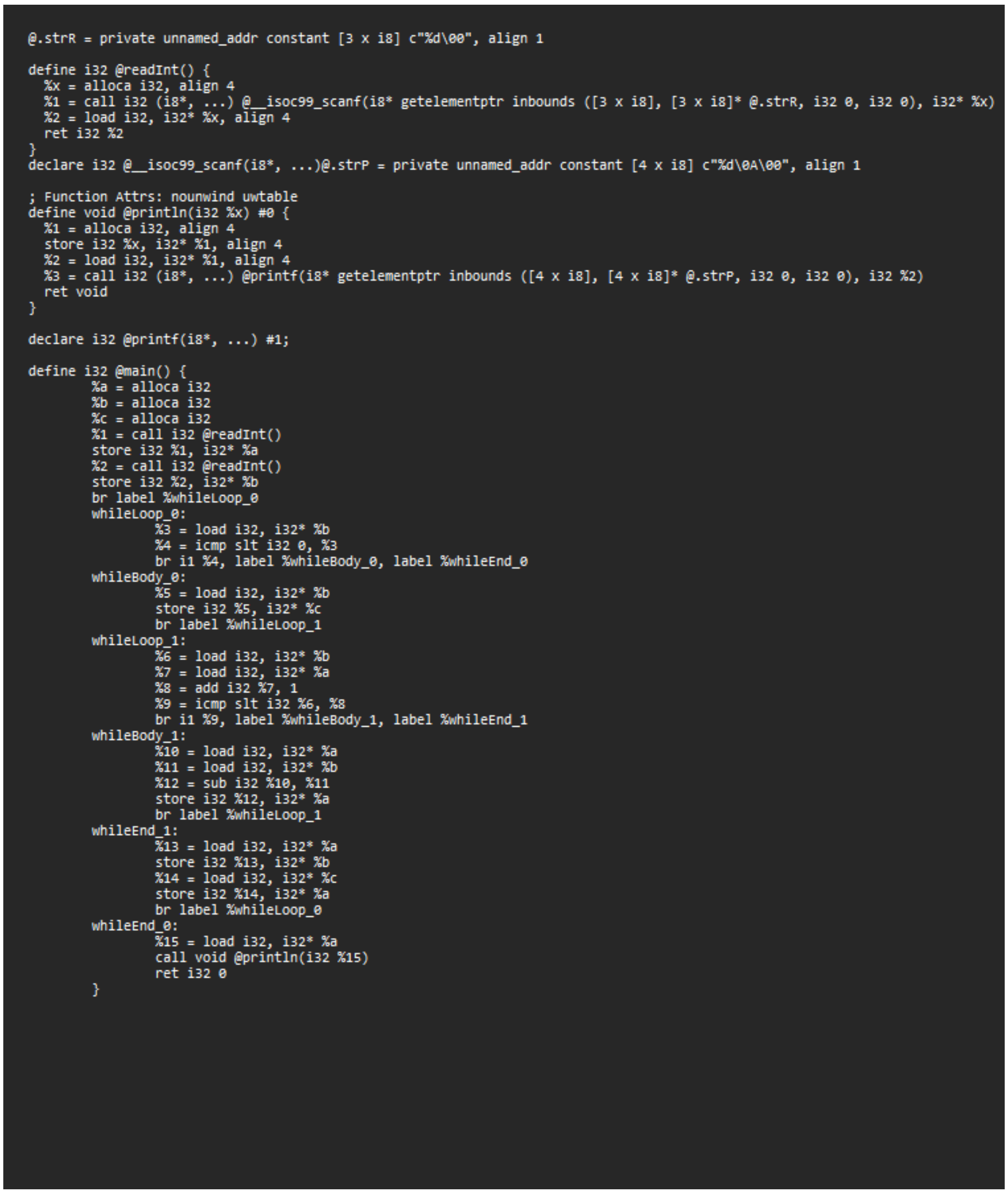
The `AST (Abstract Syntax Tree)` class allows you to create a new Parse Tree by eliminating useless elements for generating LLVM code. It permits to have a less cluttered tree in order to facilitate the generation of the LLVM code.

1.2 LLVM

The `LLVM` class allows you to generate the LLVM code from the AST by taking into account the semantics of the PASCALmaispresque grammar.

1.3 Main

The `Main` class checks the command line options, parses the input file, and then passes the input to the parser for analysis. After parsing the input, the generated parse tree is simplified to generate the AST and the latter is passed to the LLVM to output the code to stdout.



```
@.strR = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define i32 @readInt() {
  %x = alloca i32, align 4
  %1 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.strR, i32 0, i32 0), i32* %x)
  %2 = load i32, i32* %x, align 4
  ret i32 %2
}
declare i32 @__isoc99_scanf(i8*, ...)@.strP = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @println(i32 %x) #0 {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.strP, i32 0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...) #1;

define i32 @main() {
        %a = alloca i32
        %b = alloca i32
        %c = alloca i32
        %1 = call i32 @readInt()
        store i32 %1, i32* %a
        %2 = call i32 @readInt()
        store i32 %2, i32* %b
        br label %whileLoop_0
        whileLoop_0:
                %3 = load i32, i32* %b
                %4 = icmp slt i32 0, %3
                br i1 %4, label %whileBody_0, label %whileEnd_0
        whileBody_0:
                %5 = load i32, i32* %b
                store i32 %5, i32* %c
                br label %whileLoop_1
        whileLoop_1:
                %6 = load i32, i32* %b
                %7 = load i32, i32* %a
                %8 = add i32 %7, 1
                %9 = icmp slt i32 %6, %8
                br i1 %9, label %whileBody_1, label %whileEnd_1
        whileBody_1:
                %10 = load i32, i32* %a
                %11 = load i32, i32* %b
                %12 = sub i32 %10, %11
                store i32 %12, i32* %a
                br label %whileLoop_1
        whileEnd_1:
                %13 = load i32, i32* %a
                store i32 %13, i32* %b
                %14 = load i32, i32* %c
                store i32 %14, i32* %a
                br label %whileLoop_0
        whileEnd_0:
                %15 = load i32, i32* %a
                call void @println(i32 %15)
                ret i32 0
        }
```

Figure 1 – LLVM code for euclid.pmp

2 Abstract syntax tree

To generate our LLVM code, we opted to run through the parse tree obtained by our parser. However, the parse tree contains unnecessary elements like the prime variables that we introduced during the factorization and certain terminals like the epsilon and the brackets. To do this, we create an AST (Abstract Syntax Tree) from the Parse Tree by eliminating the desired elements. We then obtain the following AST for the euclid.pmp program :

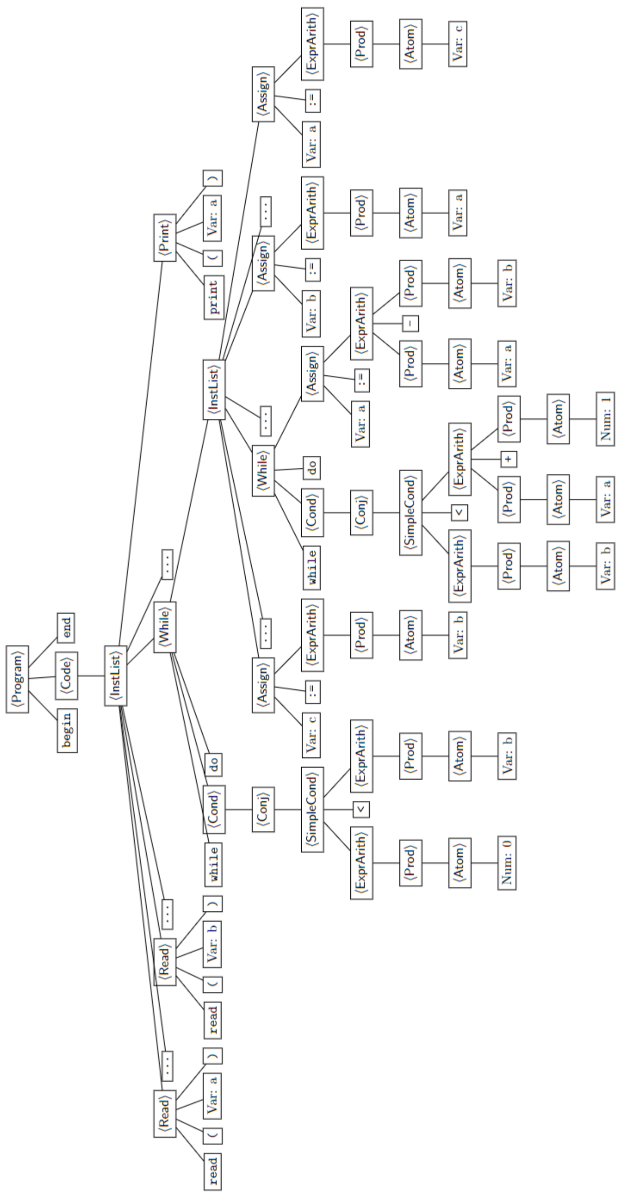


FIGURE 2 – Abstract syntax tree for euclid.pmp

3 LLVM

To generate the LLVM code, you must respect the semantics of the PASCALmais-presque program given in the appendix to the project. An LLVM class was created to generate the LLVM code from the AST of the program. This class contains several attributes :

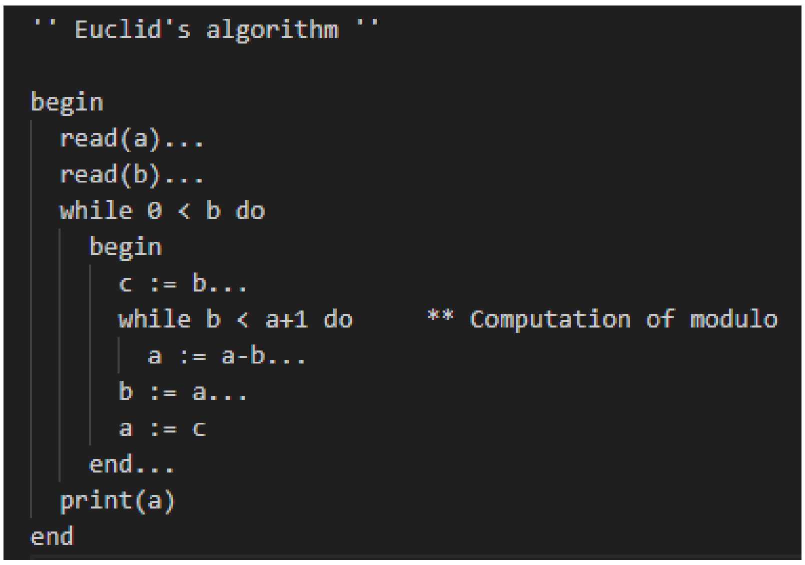
- `namedVariables` : a list of named variables that allows you to have the variables used in memory and to allocate memory at the start of the code
- `numberedVariableCounter` : a counter for local variables
- `instructionCounter` : a instruction counter to determine the name of the labels for for `if` and `while` conditions
- `ifTrueLabel, ifFalseLabel, ifEndLabel, whileLoopLabel, whileBodyLabel, whileEndLabel` : labels for `if` or `while` condition to differientiate the conditions

The function `generateCode` takes an AST in parameter and check the label of the parseTree to determine which function to call. The label of the AST will always be a variable like <Program>, <Code>, <InstList>, <Assign>, <If>, <While>, <Print> or <Read> (the other variables like <ExprArith>, <Prod>, <Atom>, <Cond>, <Conj>, <SimpleCond> are treated in sub functions).

Each variable has a function that will read the AST and output the LLVM code according to the semantics. For example, <Program> will have three children (begin, <Code> and end) and we output the start of a LLVM code regarding to begin, call another the function `code` to treat the AST with from the Code tree and output the end of a LLVM code regarding to end. This process is the same for the other functions as long as we run through the AST until the end.

4 Example files

We tested for the file euclid.pmp if the generation of LLVM code was accepted by llvm-as tool and was correct. This can be seen on Figure 3, 4, 5 and 6 :



```
'' Euclid's algorithm ''

begin
  read(a)...
  read(b)...
  while 0 < b do
    begin
      c := b...
      while b < a+1 do      ** Computation of modulo
        a := a-b...
      b := a...
      a := c
    end...
  print(a)
end
```

FIGURE 3 – euclid.pmp

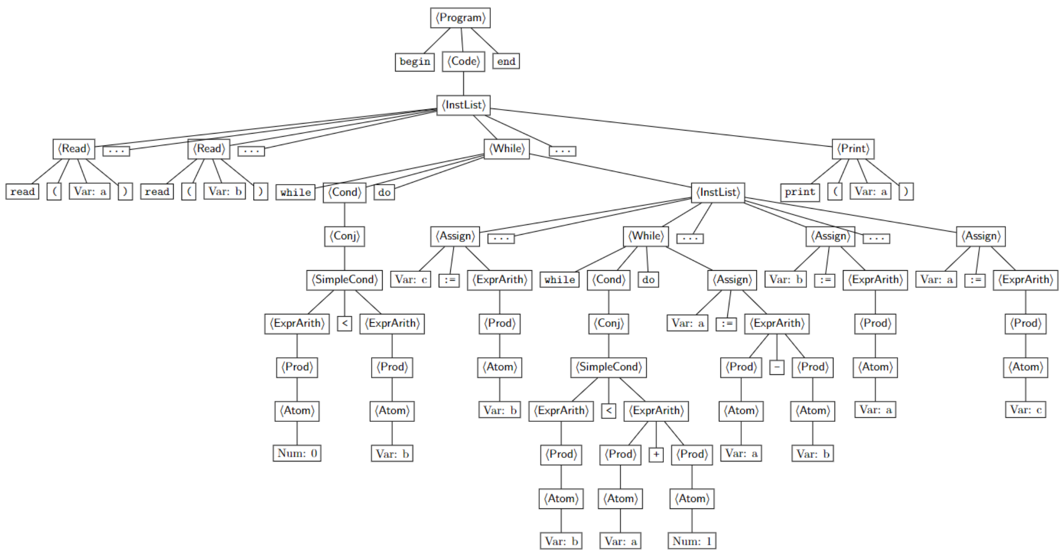
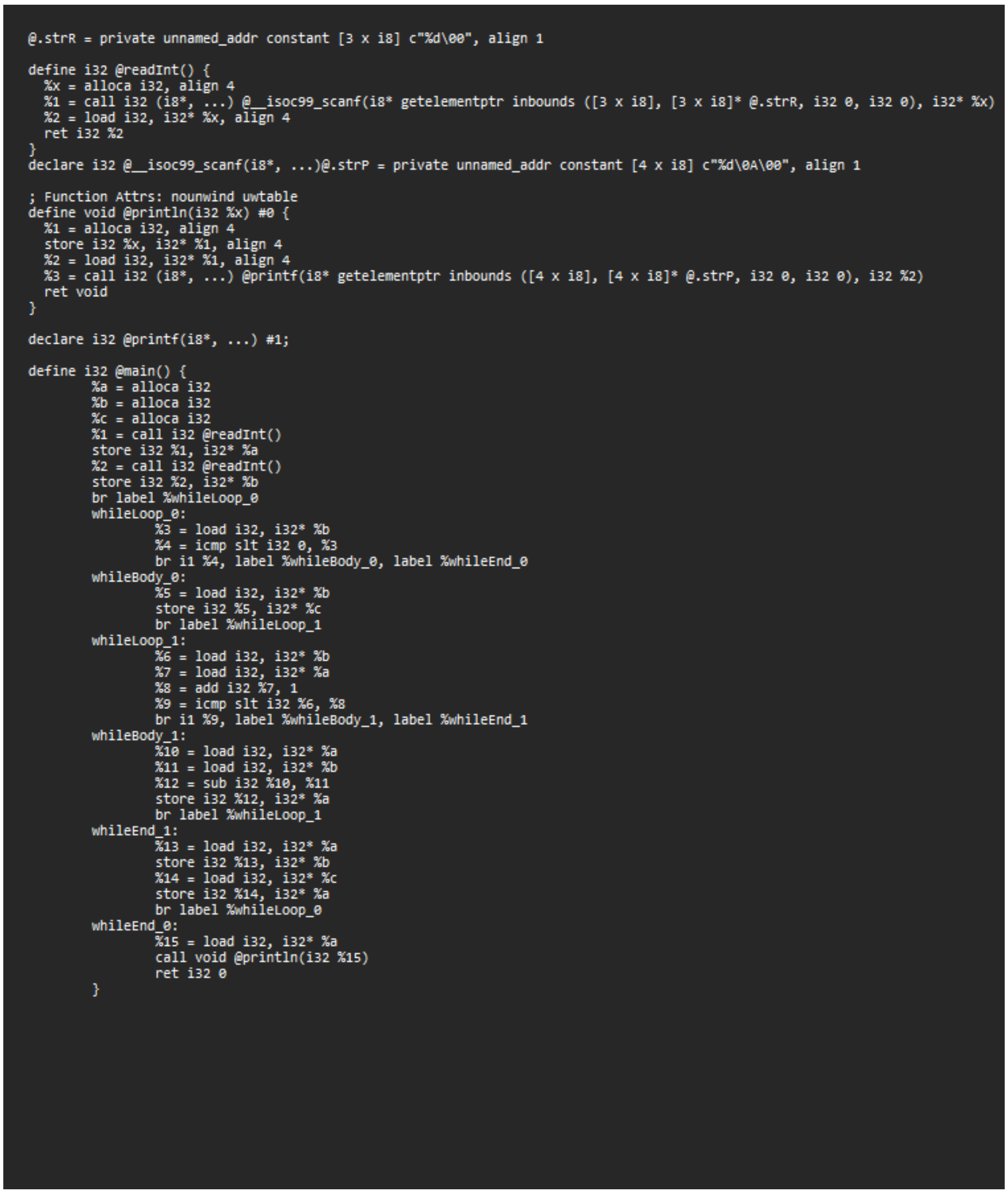

⟨Program⟩
begin ⟨Code⟩ end
⟨InstList⟩
⟨Read⟩ ... ⟨Read⟩ ... ⟨While⟩ ... ⟨Print⟩
read (Var: a) read (Var: b) while ⟨Cond⟩ do ⟨InstList⟩ print (Var: a)
⟨Conj⟩ ⟨Assign⟩ ... ⟨While⟩ ... ⟨Assign⟩ ... ⟨Assign⟩
⟨SimpleCond⟩ Var: c := ⟨ExprArith⟩ while ⟨Cond⟩ do ⟨Assign⟩ Var: b := ⟨ExprArith⟩ Var: a := ⟨ExprArith⟩
⟨ExprArith⟩ < ⟨ExprArith⟩ ⟨Prod⟩ ⟨Conj⟩ Var: a := ⟨ExprArith⟩ ⟨Prod⟩ ⟨Prod⟩
⟨Prod⟩ ⟨Prod⟩ ⟨Atom⟩ ⟨SimpleCond⟩ ⟨Prod⟩ - ⟨Prod⟩ ⟨Atom⟩ ⟨Atom⟩
⟨Atom⟩ ⟨Atom⟩ Var: b ⟨ExprArith⟩ < ⟨ExprArith⟩ ⟨Atom⟩ ⟨Atom⟩ Var: a Var: c
Num: 0 Var: b ⟨Prod⟩ ⟨Prod⟩ + ⟨Prod⟩ Var: a Var: b
⟨Atom⟩ ⟨Atom⟩ ⟨Atom⟩
Var: b Var: a Num: 1

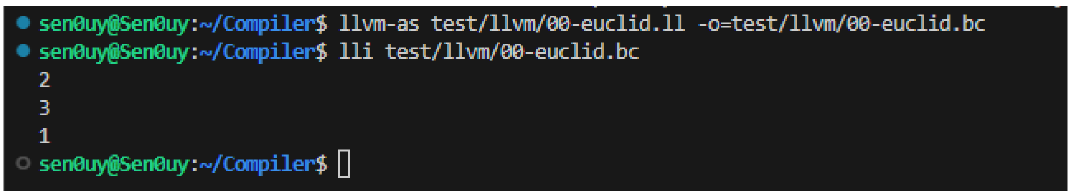
FIGURE 4 – Abstract syntax tree of euclid.pmp

Other example files can be found in test/_input folder of the project.



```
@.strR = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define i32 @readInt() {
  %x = alloca i32, align 4
  %1 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.strR, i32 0, i32 0), i32* %x)
  %2 = load i32, i32* %x, align 4
  ret i32 %2
}
declare i32 @__isoc99_scanf(i8*, ...)@.strP = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @println(i32 %x) #0 {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.strP, i32 0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...) #1;

define i32 @main() {
        %a = alloca i32
        %b = alloca i32
        %c = alloca i32
        %1 = call i32 @readInt()
        store i32 %1, i32* %a
        %2 = call i32 @readInt()
        store i32 %2, i32* %b
        br label %whileLoop_0
        whileLoop_0:
                %3 = load i32, i32* %b
                %4 = icmp slt i32 0, %3
                br i1 %4, label %whileBody_0, label %whileEnd_0
        whileBody_0:
                %5 = load i32, i32* %b
                store i32 %5, i32* %c
                br label %whileLoop_1
        whileLoop_1:
                %6 = load i32, i32* %b
                %7 = load i32, i32* %a
                %8 = add i32 %7, 1
                %9 = icmp slt i32 %6, %8
                br i1 %9, label %whileBody_1, label %whileEnd_1
        whileBody_1:
                %10 = load i32, i32* %a
                %11 = load i32, i32* %b
                %12 = sub i32 %10, %11
                store i32 %12, i32* %a
                br label %whileLoop_1
        whileEnd_1:
                %13 = load i32, i32* %a
                store i32 %13, i32* %b
                %14 = load i32, i32* %c
                store i32 %14, i32* %a
                br label %whileLoop_0
        whileEnd_0:
                %15 = load i32, i32* %a
                call void @println(i32 %15)
                ret i32 0
        }
```

FIGURE 5 – LLVM code of euclid.pmp



```
● sen0uy@Sen0uy:~/Compiler$ llvm-as test/llvm/00-euclid.ll -o=test/llvm/00-euclid.bc
● sen0uy@Sen0uy:~/Compiler$ lli test/llvm/00-euclid.bc
  2
  3
  1
○ sen0uy@Sen0uy:~/Compiler$ 
```

FIGURE 6 – Run of euclid.pmp with llvm-as and lli commands