ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

BÁO CÁO

ĐỀ TÀI MỞ RỘNG KSTN

NGUYÊN LÝ NGÔN NGỮ LẬP TRÌNH

TASK 2: GENERATE MIPS CODE FOR + OPERATOR

Giảng viên hướng dẫn: TS. Nguyễn Hứa Phùng
Sinh viên thực hiện: Nguyễn Đức An
MSSV: 2010102
Email: an.nguyenduc1406@hcmut.edu.vn

Tp. Hồ Chí Minh, Tháng 4/2023

Mục lục

1 Giới thiệu

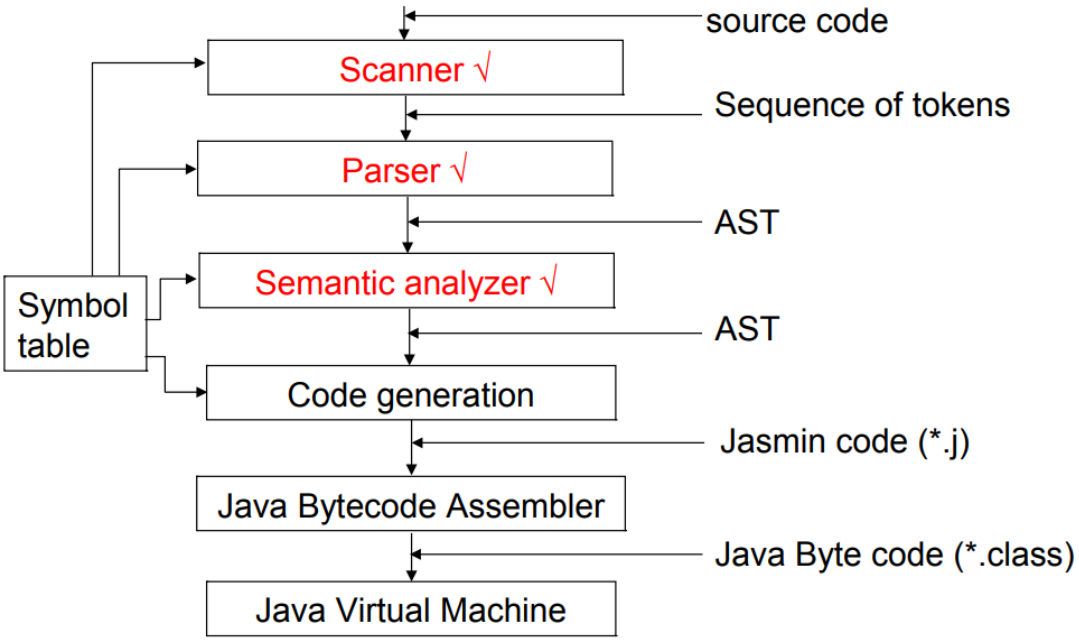
Trong báo cáo phần Task 1 của bài tập lớn, em sẽ trình bày các nội dung sau:

- Trình bày về kiến trúc của hệ thống Code Generation trong initial code.
- Hiện thực sinh mã JVM code cho toán tử + cho hai hoặc nhiều số nguyên và số thực chấm động.

2 Cơ sở lý thuyết

2.1 Kiến trúc compiler

Dưới đây là quy trình sinh mã Java bytecode từ compiler của máy tính.



Hình 1: *Quy trình sinh mã Java Bytecode từ compiler của máy tính*

– Sau quá trình Code Generation, mã nguồn của chương trình sẽ được sinh thành mã Jasmine tương ứng.

– Tiếp tục, từ mã Jasmine sẽ được Java Bytecode Assembler sinh thành Java bytecode.

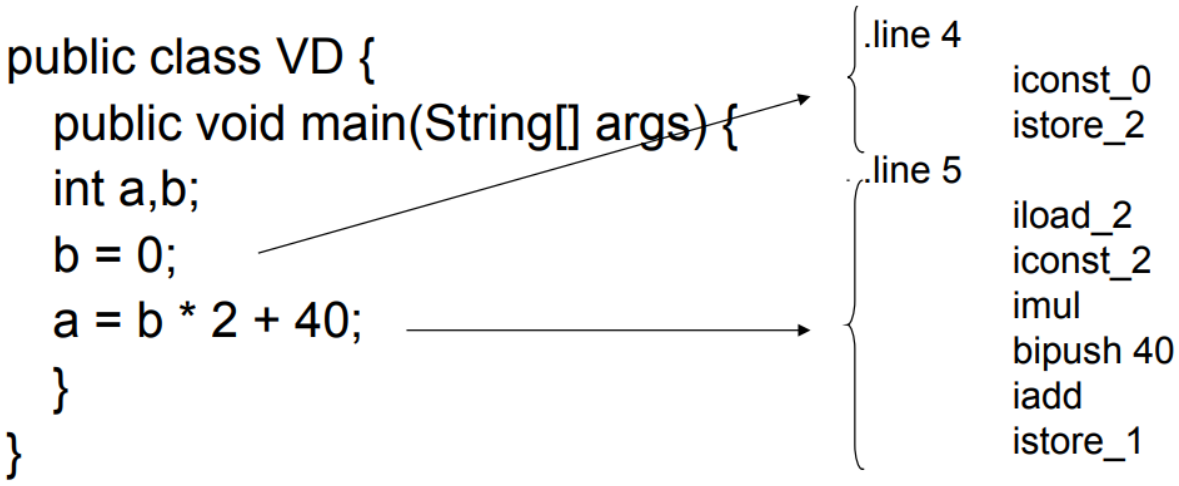
– Cuối cùng Java bytecode sẽ được chạy trên một máy ảo JVM (Java Virtual Machine).

2.2 Jasmine

– Jasmine có thể hiểu là một Java assembler.

– Trong quá trình gen code, mỗi lệnh trong chương trình Java có thể được mapping thành các lệnh Jasmine code tương ứng

– Ví dụ như chương trình Java sau sẽ được mapping thành Jasmine code tương ứng như sau:



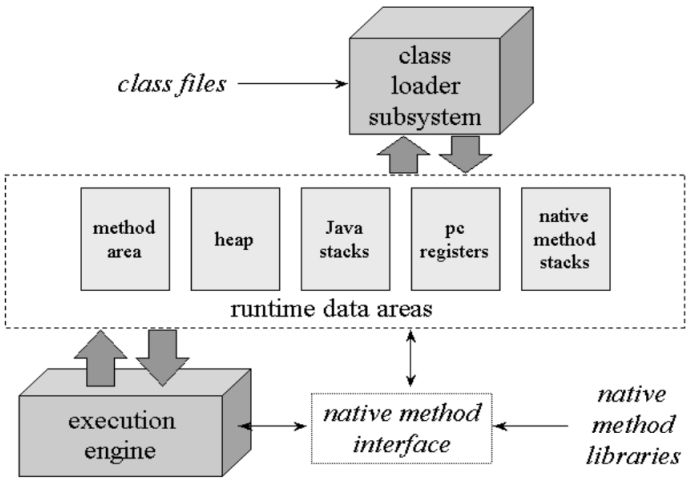
Hình 2: *Chuyển đổi từ mã Java sang Jasmine*

```
ca fe ba be 00 00 00 31 00 1b 0a 00 05 00 0e 09  00 0f 00 10 0a 00 11 00 12 07 00 13 07 00 14 01
00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d 62
65 72 54 61 62 6c 65 01 00 04 6d 61 69 6e 01 00  16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74
72 69 6e 67 3b 29 56 01 00 0a 53 6f 75 72 63 65  46 69 6c 65 01 00 07 56 44 2e 6a 61 76 61 0c 00
06 00 07 07 00 15 0c 00 16 00 17 07 00 18 0c 00  19 00 1a 01 00 02 56 44 01 00 10 6a 61 76 61 2f
6c 61 6e 67 2f 4f 62 6a 65 63 74 01 00 10 6a 61  76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 01 00
03 6f 75 74 01 00 15 4c 6a 61 76 61 2f 69 6f 2f  50 72 69 6e 74 53 74 72 65 61 6d 3b 01 00 13 6a
61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65  61 6d 01 00 07 70 72 69 6e 74 6c 6e 01 00 04 28
49 29 56 00 21 00 04 00 05 00 00 00 00 00 02 00 01 00 06 00 07 00 01 00 08 00 00 00 1d 00 01 00
01 00 00 00 05 2a b7 00 01 b1 00 00 00 01 00 09  00 00 00 06 00 01 00 00 00 01 00 09 00 0a 00 0b
00 01 00 08 00 00 00 35 00 02 00 03 00 00 00 11  03 3d 1c 05 68 10 28 60 3c b2 00 02 1b b6 00 03
b1 00 00 00 01 00 09 00 00 00 12 00 04 00 00 00  04 00 02 00 05 00 09 00 06 00 10 00 07 00 01 00
0c 00 00 00 02 00 0d
```

Hình 3: *Chuyển đổi từ mã Jasmine sang Java Bytecode*

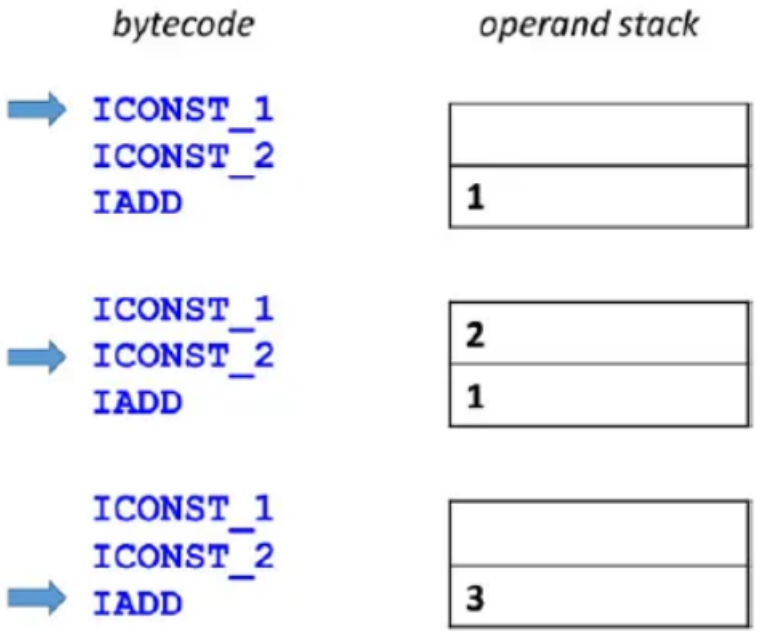
2.3 JVM

– JVM (Java Virtual Machine) là một máy ảo theo mô hình stack-based machine.



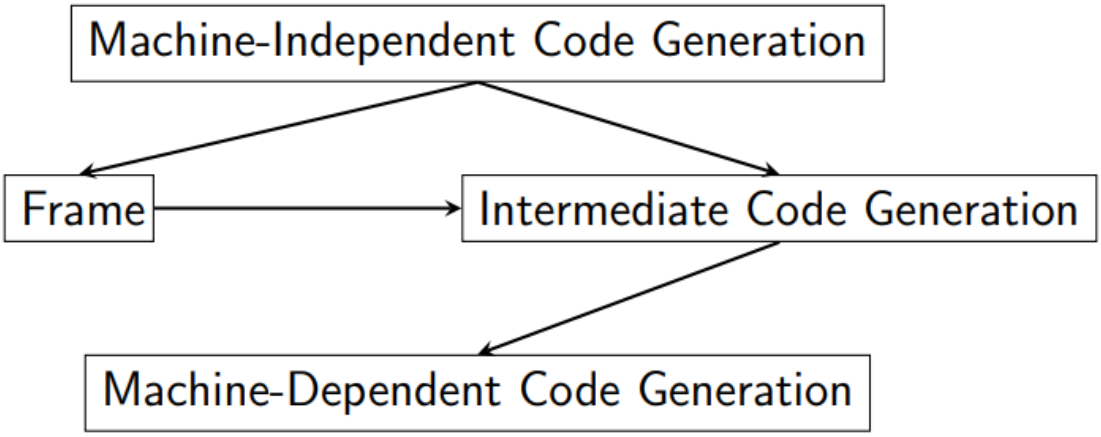
Hình 4: *Kiến trúc máy Java Virtual Machine*

– Thực hiện tính toán phép cộng hai số nguyên trong Jasmine code trên JVM như sau:



Hình 5: *Thực hiện tính toán trên JVM*

2.4 Thiết kế hệ thống Code Generation



Hình 6: *Thiết kế hệ thống Code Generation*

Trong đó:

- **Machine-Dependent Code Generation:** Đây là thành phần ở tầng thấp nhất, phụ thuộc vào loại máy ảo mà nó sinh ra, hiện tại trong initial code thì thành phần này đang sinh ra mã Jasmine.

Trong trường hợp tương lai chúng ta muốn sinh ra mã cho một loại máy khác (ví dụ MIPS), chúng ta cần thay đổi cấu trúc của component này.

- **Intermediate Code Generation:** Đây là thành phần trung gian, là kết nối giữa mã nguồn và mã máy, giúp việc sinh mã hiệu quả, tiết kiệm thời gian hơn nhờ các dịch vụ, phương thức mà khối này cung cấp.
- **Frame:** Khi sinh mã, đối với mỗi hàm, phương thức chúng ta cần một frame riêng để đảm bảo tính cục bộ (local) của các label, giá trị biến của mỗi hàm đó. Frame có vai trò đảm bảo các biến, label giữa các hàm, phương thức là độc lập với nhau.
- **Machine-Independent Code Generation:** Đây là thành phần độc lập với máy ảo, chúng ta có thể hiện thực sinh mã cho ngôn ngữ tùy ý được định nghĩa trước. Tại tầng này, chúng ta có thể sử dụng các hàm, phương thức được cung cấp bởi các khối dưới nó.

3 Kiến trúc Code Generation của initial code

3.1 CodeGenerator.py

– Tương ứng với khối **Machine-Independent Code Generation** trong kiến trúc Code Generation. Chúng ta sẽ tiến hành hiện thực sinh mã JVM tương ứng với ngôn ngữ MT22 trong file này.

– Bao gồm các thành phần chính sau:

- **class CodeGenerator(Utils):** nhận đầu vào từ file **io.java và io.class**, từ đó tiến hành khởi tạo một số hàm sử dụng trong chương trình như getInt, putInt, putIntLn,...
- **class CodeGenVisitor(BaseVisitor, Utils):** nhận vào cây astTree, biến env, _dir và xuất ra output là file Jasmin code **(*.j)**
- Cụ thể, bên trong class CodeGenVisitor chúng ta sẽ tiến hành hiện thực các hàm visit (ví dụ như **visitProgram, visitIntegerLit, visitFloatLit, visitBinExpr, visitUnExpr**,..) tương ứng với cây AST đầu vào.

3.2 CodeGenError.py

– Dùng để raise một số lỗi trong quá trình sinh mã.

– Bao gồm các thành phần chính sau:

- **class IllegalOperandException(Exception):** dùng để xuất ra lỗi khi gặp toán hạng không hợp lệ.
- **class IllegalRuntimeException(Exception):** dùng để xuất ra lỗi trong quá trình runtime.

3.3 Emitter.py

– Tương ứng với khối **Intermediate Code Generation** trong kiến trúc Code Generation.

– Bao gồm các phương thức chính sau (trong quá trình hiện thực toán tử +):

- **getJVMType**: chuyển đổi kiểu dữ liệu từ source code đến kiểu dữ liệu tương ứng ở Jasmine, ví dụ từ IntegerType từ ngôn ngữ MT22 sang kiểu I của Jasmine code.

- **getFullType**: trả về full type kiểu dữ liệu của source code, ví dụ int, void,...
- **emitPUSHICONST**: thêm một hằng số nguyên vào trong stack.
  - Trong khoảng **[-1, 5]** sử dụng **emitICONST**.
  - Trong khoảng **[-128, 127]** sử dụng **emitBIPUSH**.
  - Trong khoảng **[-32768, 32767]** sử dụng **emitSIPUSH**.
- **emitPUSHFCONST**: thêm một hằng số thực có chấm động vào trong stack.
  - Nếu giá trị là **[0.0, 1.0, 2.0]** sử dụng **emitFCONST**
  - Các trường hợp còn lại sử dụng **emitLDC**
- **emitALOAD**: load một biến cục bộ (local variable) vào operand stack.
- **emitASTORE**: lưu trữ giá trị từ operand stack vào một biến cục bộ (local variable).
- **emitVAR**: khởi tạo một biến trong một frame, bao gồm các thông tin sau: giá trị, tên biến, kiểu, fromLabel, toLabel, frame.
- **emitADDOP**: nhận vào 3 tham số là lexeme (ví dụ toán tử "+"), kiểu, frame, kết quả trả ra là toán tử +.
- **emitI2F**: được sử dụng trong trường hợp ép kiểu từ int thành float.

3.4 Frame.py

– Tương ứng với khối **Frame** trong kiến trúc Code Generation.

– Bao gồm các phương thức chính sau:

- **getCurrIndex**: lấy vị trí index hiện tại của frame.
- **setCurrIndex**: gán vị trí index hiện tại bằng một số nguyên.
- **push**: thêm một phần tử vào stack, cập nhật lại kích thước hiện tại của stack.
- **pop**: xóa một phần tử của stack, cập nhật lại kích thước hiện tại của stack.
- **getStackSize**: lấy kích thước hiện tại của stack.
- **enterScope**: vào trong scope.
- **exitScope**: thoát ra khỏi scope.
- **getStartLabel**: trả về label bắt đầu.
- **getNewLabel**: tạo label mới và trả về label đó.
- **getEndLabel**: trả về label kết thúc.

3.5 MachineCode.py

– Tương ứng với khối **Machine-Dependent Code Generation** của kiến trúc Code Generation.

– Bao gồm các thành phần chính sau:

- **emitICONST**: thêm một hằng số nguyên trong khoảng [-1, 5] vào operand stack.
- **emitBIPUSH**: thêm một hằng số nguyên trong khoảng [-128, 127] vào operand stack.
- **emitSIPUSH**: thêm một hằng số nguyên trong khoảng [-32768, 32767] vào operand stack.
- **emitFCONST**: thêm một hằng số thực trong khoảng [0.0, 1.0, 2.0] vào operand stack.

- **emitLDC**: thêm một hằng số thực trong các trường hợp còn lại vào operand stack.
- **emitILOAD**: load một hằng số nguyên vào operand stack.
- **emitISTORE**: lưu trữ giá trị một hằng số nguyên từ operand vào biến cục bộ.
- **emitFLOAD**: load một hằng số thực vào operand stack.
- **emitFSTORE**: lưu trữ giá trị một hằng số thực từ operand vào biến cục bộ.
- **emitALOAD**: load một biến cục bộ vào operand stack.
- **emitASTORE**: lưu trữ giá trị từ operand stack vào biến cục bộ.

4 Hiện thực quá trình sinh JVM code cho toán tử +

4.1 Giai đoạn Lexer

– Đầu tiên, em sẽ tiến hành chỉnh sửa lexer cho **INTLIT** và **FLOATLIT** theo mô tả của ngôn ngữ MT22 trong file **MT22.g4**.

```
fragment DECIMAL_PART: '.' [0-9]*;
fragment EXPONENT_PART: [eE] [+-]? [0-9]+;
fragment INT_PART: [1-9] [0-9]* | [0] | [1-9] [0-9]* ('_' [0-9]+)*;

INTLIT: [1-9] [0-9]* | [0]
       | [1-9] [0-9]* ('_' [0-9]+)* {self.text = self.text.replace("_", "")};

FLOATLIT: INT_PART DECIMAL_PART EXPONENT_PART {self.text = self.text.replace("_", "")}
        | INT_PART DECIMAL_PART {self.text = self.text.replace("_", "")}
        | INT_PART EXPONENT_PART {self.text = self.text.replace("_", "")}
        | DECIMAL_PART EXPONENT_PART;
```

– Bên cạnh đó, em sẽ tiến hành thêm lexer **ADD** và **FLOATTYPE** cho dấu "+" và kiểu số thực. Ngoài ra, để mở rộng cho phép cộng số âm, em sẽ hiện thực thêm lexer của dấu "–".

```
INTTYPE: 'int';
VOIDTYPE: 'void';
FLOATTYPE: 'float';
ADD: '+';
SUB: '-';
```

- **Lưu ý:**
  – Đối với hằng số nguyên, ngôn ngữ MT22 chấp nhận dấu '_' có thể là một thành phần của INTLIT. Ví dụ 1_23 vẫn được xem là một hằng số nguyên hợp lệ trong ngôn ngữ MT22 và sau đó sẽ được xử lý và lưu trữ như với giá trị 123.
  – Đối với hằng số thực, bao gồm ít nhất 2 trong 3 thành phần int, decimal và exponent. Ví dụ 1.23 hoặc 1.2e-3 là các hằng số thực hợp lệ.

4.2 Giai đoạn Parser

– Tại đây em sẽ hiện thực thêm parser cho toán tử + thông qua parser **addexpr** theo kiểu **Binary Expression, Infix và Left-associate**, tương tự như mô tả của ngôn ngữ MT22. Ngoài ra, để mở rộng cho phép cộng số âm (Unexpr), em sẽ hiện thực thêm phép **signexpr** theo kiểu **Unary Expression, Prefix, Right-associate** như sau:

```
addexpr: addexpr ADD signexpr | signexpr;

signexpr: SUB signexpr | operand;

operand: subexpr | INTLIT | FLOATLIT;
```

– Bên cạnh đó, để có thể dễ dàng trong quá trình test output và mở rộng cho phép cộng hai số thực, em sẽ chỉnh sửa **mptype** và **exp**. Ngoài ra, em có mở rộng phép cộng cho các **subexpr**.

```
mptype: INTTYPE | FLOATTYPE | VOIDTYPE;

exp: funcall | signexpr | addexpr | INTLIT | FLOATLIT;

subexpr: LB exp RB;
```

4.3 Giai đoạn ASTGen

– Đầu tiên, em sẽ tiến hành hiện thực các phương thức **visitAddexpr**, **visitSignexpr visitOperand**, **visitSubexpr** trong file **ASTGeneration.py**

```
# addexpr: addexpr ADD signexpr | signexpr;
def visitAddexpr(self, ctx: MT22Parser.AddexprContext):
    if ctx.ADD():
        left = self.visit(ctx.addexpr())
        right = self.visit(ctx.signexpr())
        return BinExpr(ctx.ADD().getText(), left, right)
    else:
        return self.visit(ctx.signexpr())

# signexpr: SUB signexpr | operand;
def visitSignexpr(self, ctx: MT22Parser.SignexprContext):
    if ctx.SUB():
        return UnExpr(ctx.SUB().getText(), self.visit(ctx.signexpr()))
    else:
        return self.visit(ctx.operand())

# operand: subexpr | INTLIT | FLOATLIT;
def visitOperand(self, ctx: MT22Parser.OperandContext):
    if ctx.subexpr():
        return self.visit(ctx.subexpr())
    elif ctx.FLOATLIT():
        if str(ctx.FLOATLIT().getText())[0:2] == '.e' or
```

```
            str(ctx.FLOATLIT().getText())[0:2] == '.E':
            return FloatLit(0.0)
        return FloatLit(float(ctx.FLOATLIT().getText()))
    else:
        return IntegerLit(int(ctx.INTLIT().getText()))

# subexpr: LB exp RB;;
def visitSubexpr(self, ctx: MT22Parser.SubexprContext):
    return self.visit(ctx.exp())
```

– Ngoài ra, em sẽ chỉnh sửa các phương thức **visitMptype** và **visitExp** để tương ứng với phần mở rộng cho phép cộng số thực.

```
def visitMptype(self, ctx: MT22Parser.MptypeContext):
    if ctx.INTTYPE():
        return IntegerType()
    elif ctx.FLOATTYPE():
        return FloatType()
    else:
        return VoidType()

# exp: funcall | signexpr | addexpr | INTLIT | FLOATLIT;
def visitExp(self, ctx: MT22Parser.ExpContext):
    if ctx.funcall():
        return self.visit(ctx.funcall())
    elif ctx.signexpr():
        return self.visit(ctx.signexpr())
    elif ctx.addexpr():
        return self.visit(ctx.addexpr())
    elif ctx.FLOATLIT():
        if str(ctx.FLOATLIT().getText())[0:2] == '.e' or
            str(ctx.FLOATLIT().getText())[0:2] == '.E':
            return FloatLit(0.0)
        return FloatLit(float(ctx.FLOATLIT().getText()))
    else:
        return IntegerLit(int(ctx.INTLIT().getText()))
```

4.4 Giai đoạn CodeGen

4.4.1 Hiện thực sinh mã JVM code cho toán tử + số nguyên

– Đầu tiên, em sẽ hiện thực phương thức **visitIntegerLit** trong file **CodeGenerator.py**. Ở đây, chúng ta sẽ sử dụng phương thức **emitPUSHICONST** được cung cấp từ class **Emitter** được hiện thực trong file **Emitter.py** để push một hằng số nguyên vào trong frame.

```
def visitIntegerLit(self, ast, o):
    #ast: IntLiteral
    #o: Any
```

```
        ctxt = o
        frame = ctxt.frame
        return self.emit.emitPUSHICONST(ast.val, frame), IntegerType()
```

– Để hiện thực được toán tử + cho phép cộng hai số nguyên, em sẽ tiếp tục hiện thực phương thức **visitBinExpr** (thông tin về các phương thức có thể tìm thấy ở file **Visitor.py**).

```
    def visitBinExpr(self, ast, o):
        frame = o.frame
        e1c, e1t = self.visit(ast.left, o)
        e2c, e2t = self.visit(ast.right, o)
        typeop = IntegerType()
        return e1c + e2c + self.emit.emitADDOP(str(ast.op), typeop, frame), typeop
```

– Đối với phần mở rộng thêm cho phép cộng số nguyên âm, em sẽ tiếp tục hiện thực thêm phương thức **visitUnExpr** như sau.

```
    def visitUnExpr(self, ast, o):
        val, typ = self.visit(ast.val, o)
        frame = o.frame
        if str(ast.op) == '-':
            return val + self.emit.emitNEGOP(typ, frame), typ
        else:
            return val
```

- **Giải thuật:**
  - Đối với quá trình hiện thực phép + của lớp **visitBinExpr**, chúng ta sẽ lần lượt visit vào các nhánh left và right của cây AST để lấy được giá trị của các operand, hàm visit sẽ trả về hai giá trị ec và et tương ứng với giá trị và kiểu của từng operand, ở đây e1c, e1t là giá trị và kiểu của cây con trái và e2c, e2t là giá trị và kiểu của cây con phải.
  - Trong trường hợp nếu một trong các toán hạng của phép cộng là số âm, nó phải được xử lý ở lớp **visitUnExpr** trước.
  - Để thực hiện phép cộng hai phương thức, ta sẽ lần lượt push các operand e1c và e2c vào trong stack, đối với toán tử "+", chúng ta sẽ sử dụng phương thức **emitADDOP** nhận vào ba tham số là lexeme (ở đây là operator của cây AST, cụ thể là dấu "+"), kiểu, frame.
  - Kết quả trả ra là hàm trên là đoạn mã Jasmine (JVM code) trong file **MT22class.j** và kết quả của phép cộng số nguyên khi test trong **CodeGenSuite.py**

4.4.2 Hiện thực sinh mã JVM code cho toán tử + số thực

– Để thực hiện phép cộng cho số thực, em sẽ tiến hành chỉnh sửa và hiện thực thêm các phương thức **getJVMType** trong class Emitter của file **Emitter.py** và hiện thực thêm các hàm **putFloat** và **putFloatLn** trong trong class CodeGeneration của file **CodeGenerator.py**

- Đối với file **Emitter.py**

```
def getJVMType(self, inType):
    typeIn = type(inType)
    if typeIn is IntegerType:
        return "I"
    elif typeIn is FloatType:
        return "F"
    elif typeIn is cgen.StringType:
        return "Ljava/lang/String;"
    elif typeIn is VoidType:
        return "V"
    elif typeIn is cgen.ArrayPointerType:
        return "[" + self.getJVMType(inType.eleType)
    elif typeIn is MType:
        return "(" + "".join(list(map(lambda x: self.getJVMType(x),
            inType.partype))) + ")" + self.getJVMType(inType.rettype)
    elif typeIn is cgen.ClassType:
        return "L" + inType.cname + ";"
```

- Đối với file **CodeGenerator.py**

```
class CodeGenerator(Utils):
    def __init__(self):
        self.libName = "io"

    def init(self):
        return [Symbol("getInt", MType(list(), IntegerType()),
            CName(self.libName)),
                Symbol("putInt", MType([IntegerType()],
                      VoidType()), CName(self.libName)),
                Symbol("putIntLn", MType(
                    [IntegerType()], VoidType()), CName(self.libName)),
                Symbol("putFloat", MType(
                    [FloatType()], VoidType()), CName(self.libName)),
                Symbol("putFloatLn", MType(
                    [FloatType()], VoidType()), CName(self.libName))
                ]
```

– Tiếp theo, em sẽ hiện thực phương thức **visitFloatLit** trong file **CodeGenerator.py**. Ở đây, chúng ta sẽ sử dụng phương thức **emitPUSHFCONST** được cung cấp từ class **Emitter** được hiện thực trong file **Emitter.py** để push một hằng số thực vào trong frame.

```
def visitFloatLit(self, ast, o):
    #ast: FloatLiteral
    #o: Any

    ctxt = o
    frame = ctxt.frame
    return self.emit.emitPUSHFCONST(str(ast.val), frame), FloatType()
```

– Sau đó, để hiện thực được toán tử + cho phép cộng hai số thực hoặc số thực với số nguyên, em sẽ tiếp tục cập nhật phương thức **visitBinExpr** như sau.

```
def visitBinExpr(self, ast, o):
    frame = o.frame
    e1c, e1t = self.visit(ast.left, o)
    e2c, e2t = self.visit(ast.right, o)
    typeop = IntegerType()
    if (type(e1t) is FloatType or type(e2t) is FloatType):
        typeop = FloatType
        if type(e1t) is IntegerType:
            e1c = e1c + self.emit.emitI2F(frame)
        if type(e2t) is IntegerType:
            e2c = e2c + self.emit.emitI2F(frame)
    return e1c + e2c + self.emit.emitADDOP(str(ast.op), typeop, frame), typeop
```

– Giải thuật:

- Đầu tiên, chúng ta sẽ lần lượt visit vào các nhánh left và right của cây AST để lấy được giá trị của các operand, hàm visit sẽ trả về hai giá trị ec và et tương ứng với giá trị và kiểu của từng operand, ở đây e1c, e1t là giá trị và kiểu của cây con trái và e2c, e2t là giá trị và kiểu của cây con phải.
- Ở đây chúng ta sẽ tiến hành kiểm tra, nếu đầu vào là hai số nguyên thì kết quả trả ra là một số nguyên **IntergerType**, các trường hợp còn lại ví dụ như số nguyên với số thực, số thực với số thực thì kết quả trả ra là một số thực **FloatType**.
- Để thực hiện phép cộng hai phương thức, ta sẽ lần lượt push các operand e1c và e2c vào trong stack, đối với toán tử "+", chúng ta sẽ sử dụng phương thức **emitADDOP** nhận vào ba tham số là lexeme (ở đây là operator của cây AST, cụ thể là dấu "+"), kiểu, frame.
- Kết quả trả ra là hàm trên là đoạn mã Jasmine (JVM code) trong file **MT22class.j** và kết quả của phép cộng các số nguyên hoặc số thực khi test trong **CodeGenSuite.py**

5 Kiểm thử chương trình

Chạy test CodeGen với lệnh: **python run.py test CodeGenSuite**

```
PS C:\Users\ducan\Desktop\initial\src> python run.py test CodeGenSuite
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Generated: MT22Class.class
Tests run  11
Errors  []
[]
Test output
 ...........
----------------------------------------------------------------------
Ran 11 tests in 2.719s

OK
```

Hình 7: *Kết quả khi chạy test với IDE trên VSCode*

5.1 Phép cộng số nguyên không dấu

5.1.1 Testcase 1: Phép cộng hai số nguyên

- **Testcase:**

```
def test_add_int_1(self):
    input = """void main() {putInt(5 + 2);}"""
    expect = "7"
    self.assertTrue(TestCodeGen.test(input, expect, 502))
```

- **Output:**
  - File **502.txt** trong thư mục **solution**:

```
7
```

– File **MT22Class.j** trong thư mục **solution/502**:

```
.source MT22Class.java
.class public MT22Class
.super java.lang.Object

.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
  iconst_5
  iconst_2
  iadd
  invokestatic io/putInt(I)V
Label1:
  return
.limit stack 2
.limit locals 1
.end method

.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

5.1.2 Testcase 2: Phép cộng nhiều số nguyên

- **Testcase:**

```
def test_add_int_2(self):
    input = """void main() {putInt(5 + 2 + 3);}"""
    expect = "10"
    self.assertTrue(TestCodeGen.test(input, expect, 503))
```

- **Output:**

– File **503.txt** trong thư mục **solution**:

```
10
```

– File **MT22Class.j** trong thư mục **solution/503**:

```
.source MT22Class.java
.class public MT22Class
.super java.lang.Object

.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
  iconst_5
  iconst_2
  iadd
  iconst_3
  iadd
  invokestatic io/putInt(I)V
Label1:
  return
.limit stack 2
.limit locals 1
.end method

.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

5.1.3 Testcase 3: Phép cộng số nguyên có ngoặc

- **Testcase:**

```
def test_add_int_3(self):
    input = """void main() {putInt(5 + (2 + 5));}"""
    expect = "12"
    self.assertTrue(TestCodeGen.test(input, expect, 504))
```

- **Output:**
  - File **504.txt** trong thư mục **solution**:

```
12
```

  - File **MT22Class.j** trong thư mục **solution/504**:

```
.source MT22Class.java
.class public MT22Class
```

```
.super java.lang.Object

.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
  iconst_5
  iconst_2
  iconst_5
  iadd
  iadd
  invokestatic io/putInt(I)V
Label1:
  return
.limit stack 3
.limit locals 1
.end method

.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

5.2 Phép cộng số thực

5.2.1 Testcase 1: Phép cộng số thực với số nguyên

- **Testcase:**

```
def test_add_float_1(self):
    input = """void main() {putFloat(7.5 + 2);}"""
    expect = "9.5"
    self.assertTrue(TestCodeGen.test(input, expect, 505))
```

- **Output:**
  - File **505.txt** trong thư mục **solution**:

```
9.5
```

  - File **MT22Class.j** trong thư mục **solution/505**:

```
.source MT22Class.java
.class public MT22Class
.super java.lang.Object
```

```
.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
  ldc 7.5
  iconst_2
  i2f
  fadd
  invokestatic io/putFloat(F)V
Label1:
  return
.limit stack 2
.limit locals 1
.end method

.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

5.2.2 Testcase 2: Phép cộng số thực với số thực

- **Testcase:**

```
def test_add_float_2(self):
    input = """void main() {putFloat(7.5 + 1.4);}"""
    expect = "8.9"
    self.assertTrue(TestCodeGen.test(input, expect, 506))
```

- **Output:**
  - File **506.txt** trong thư mục **solution**:

```
8.9
```

  - File **MT22Class.j** trong thư mục **solution/506**:

```
.source MT22Class.java
.class public MT22Class
.super java.lang.Object

.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
```

```
    ldc 7.5
    ldc 1.4
    fadd
    invokestatic io/putFloat(F)V
  Label1:
    return
  .limit stack 2
  .limit locals 1
  .end method

  .method public <init>()V
  .var 0 is this LMT22Class; from Label0 to Label1
  Label0:
    aload_0
    invokespecial java/lang/Object/<init>()V
  Label1:
    return
  .limit stack 1
  .limit locals 1
  .end method
```

5.2.3 Testcase 3: Phép cộng số nguyên với số thực

- **Testcase:**

```
def test_add_float_3(self):
    input = """void main() {putFloat(7 + 2.5);}"""
    expect = "9.5"
    self.assertTrue(TestCodeGen.test(input, expect, 507))
```

- **Output:**
  - File **507.txt** trong thư mục **solution**:

```
    9.5
```

  - File **MT22Class.j** trong thư mục **solution/507**:

```
  .source MT22Class.java
  .class public MT22Class
  .super java.lang.Object

  .method public static main([Ljava/lang/String;)V
  .var 0 is args [Ljava/lang/String; from Label0 to Label1
  Label0:
    bipush 7
    i2f
    ldc 2.5
    fadd
    invokestatic io/putFloat(F)V
```

```
  Label1:
    return
  .limit stack 2
  .limit locals 1
  .end method

  .method public <init>()V
  .var 0 is this LMT22Class; from Label0 to Label1
  Label0:
    aload_0
    invokespecial java/lang/Object/<init>()V
  Label1:
    return
  .limit stack 1
  .limit locals 1
  .end method
```

5.3 Phép cộng số nguyên có dấu

5.3.1 Testcase 1: Phép cộng số nguyên dương với số nguyên âm

- **Testcase:**

```
    def test_add_sign_int_1(self):
        input = """void main() {putInt(-5 + 2);}"""
        expect = "-3"
        self.assertTrue(TestCodeGen.test(input, expect, 508))
```

- **Output:**
  - File **508.txt** trong thư mục **solution**:

```
    -3
```

  - File **MT22Class.j** trong thư mục **solution/508**:

```
  .source MT22Class.java
  .class public MT22Class
  .super java.lang.Object

  .method public static main([Ljava/lang/String;)V
  .var 0 is args [Ljava/lang/String; from Label0 to Label1
  Label0:
    iconst_5
    ineg
    iconst_2
    iadd
    invokestatic io/putInt(I)V
  Label1:
    return
```

```
.limit stack 2
.limit locals 1
.end method

.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

5.3.2 Testcase 2: Phép cộng số nguyên âm với số thực

- **Testcase:**

```
def test_add_sign_int_2(self):
    input = """void main() {putFloat(-5 + 2.5);}"""
    expect = "-2.5"
    self.assertTrue(TestCodeGen.test(input, expect, 509))
```

- **Output:**
  - File **509.txt** trong thư mục **solution**:

```
-2.5
```

  - File **MT22Class.j** trong thư mục **solution/509**:

```
.source MT22Class.java
.class public MT22Class
.super java.lang.Object

.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
  iconst_5
  ineg
  i2f
  ldc 2.5
  fadd
  invokestatic io/putFloat(F)V
Label1:
  return
.limit stack 2
.limit locals 1
.end method
```

```
.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

5.3.3 Testcase 3: Phép cộng hai số nguyên âm

- **Testcase:**

```
def test_add_sign_int_3(self):
    input = """void main() {putInt(-5 + (-2));}"""
    expect = "-7"
    self.assertTrue(TestCodeGen.test(input, expect, 510))
```

- **Output:**
  - File **510.txt** trong thư mục **solution**:

```
-7
```

  - File **MT22Class.j** trong thư mục **solution/510**:

```
.source MT22Class.java
.class public MT22Class
.super java.lang.Object

.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String; from Label0 to Label1
Label0:
  iconst_5
  ineg
  iconst_2
  ineg
  iadd
  invokestatic io/putInt(I)V
Label1:
  return
.limit stack 2
.limit locals 1
.end method

.method public <init>()V
.var 0 is this LMT22Class; from Label0 to Label1
```

```
Label0:
  aload_0
  invokespecial java/lang/Object/<init>()V
Label1:
  return
.limit stack 1
.limit locals 1
.end method
```

6 Kết luận

Như vậy, qua bài tập lớn 1 của phần mở rộng môn Nguyên lý ngôn ngữ lập trình, em đã hoàn thành được việc sinh mã JVM code cho toán tử + đối với các số nguyên và số thực, trong tương lai em sẽ tiếp tục nghiên cứu và sinh mã MIPS cho ngôn ngữ MT22 trong các bài tập lớn sau.