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


# Nguyên lý ngôn ngữ lập trình mở rộng (CO300C)

# Đề tài: Sinh mã trung gian cho ngôn ngữ lập trình

GVHD: Ts. Nguyễn Hứa Phùng
Ths. Trần Ngọc Bảo Duy
SVTH: Nguyễn Minh Mỹ - 2013811

TP. HỒ CHÍ MINH, THÁNG 03/2023

# Mục lục

# 1 Giới thiệu

## 1.1 Ngôn ngữ lập trình là gì?

**Ngôn ngữ lập trình** [5](programming language) là ngôn ngữ hình thức bao gồm một tập hợp các lệnh tạo ra nhiều loại đầu ra khác nhau. Ngôn ngữ lập trình được sử dụng trong lập trình máy tính để thực hiện các thuật toán.

Hầu hết các ngôn ngữ lập trình bao gồm các lệnh cho máy tính. Có những máy lập trình sử dụng một tập hợp các lệnh cụ thể, thay vì các ngôn ngữ lập trình chung chung. Kể từ đầu những năm 1800, các chương trình đã được sử dụng để định hướng hoạt động của máy móc như khung dệt Jacquard, hộp nhạc và đàn piano cơ. Các chương trình cho những máy này (chẳng hạn như cuộn giấy của đàn piano) không tạo ra các hành vi khác nhau để đáp ứng với các đầu vào hoặc điều kiện khác nhau.

Ngày nay, ngành công nghiệp phần mềm phát triển mạnh mẽ và có rất nhiều ngôn ngữ lập trình được tạo ra để đáp ứng các nhu cầu đa dạng của người dùng. Từ cấp thấp đến cấp cao, các ngôn ngữ lập trình như R, Python, Java, C++... được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau, từ khoa học dữ liệu đến phát triển ứng dụng di động.

Tuy nhiên, để tạo ra một ngôn ngữ lập trình mới thì không đơn giản chỉ là đặt ra một mục tiêu và thiết kế cú pháp. Một ngôn ngữ lập trình mới cần được thiết kế đầy đủ tính năng và đặc điểm cần có để đáp ứng mục đích sử dụng của nó, từ đó các nhà phát triển có thể thiết kế cú pháp, cấu trúc và cách thức hoạt động của ngôn ngữ.

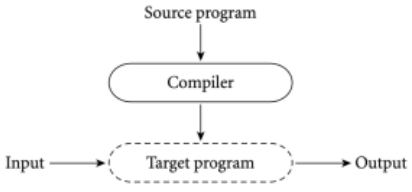
## 1.2 Hiện thực ngôn ngữ

Trước khi một chương trình có thể chạy được, nó phải được dịch thành dạng mà máy tính có thể hiểu và thực thi được. Những hệ thống phần mềm thực hiện việc dịch này gọi chung là **translators**.

Có 3 hình thức của quá trình dịch từ ngôn ngữ gốc sang ngôn ngữ máy đó là: trình biên dịch (Complier), trình thông dịch (Interpreter) và mô hình kết hợp cả 2 quá trình trên (Hybrid systems).

### 1.2.1 Compilation

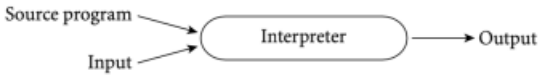
**Compilation** [1] là quá trình biên dịch một chương tình máy tính từ ngôn ngữ lập trình cấp cao (như C++, Java, Python) thành mã máy (binary code) mà máy tính có thể hiểu được. Trong quá trình biên dịch, chương trình sẽ được phân tích cú pháp, biên dịch và liên kết các thư viện để tạo ra một tệp thực thi có thể chạy được trên máy tính. Quá trình biên dịch giúp cho chương trình có thể chạy nhanh hơn, tối ưu hóa và giảm thiểu các lỗi có thể xảy ra khi chạy chương trình.




Hình 1: A Compiler [3]

### 1.2.2 Pure Interpretation

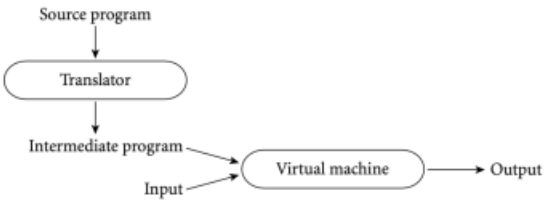
**Pure Interpretation** [4]là phương pháp thực thi mã nguồn của một chương trình máy tính bằng cách đọc và thực thi từng dòng lệnh một mà không cần biên dịch trước. Nó giúp cho quá trình phát triển chương trình trở nên nhanh chóng và thuận tiện hơn, tuy nhiên tốc độ thực thi thường chậm hơn so với các ngôn ngữ được biên dịch trước và không thể tối ưu hóa mã nguồn như biên dịch trước được. Một sô ngôn ngữ lập trình được thực thi thông qua trình thông dịch (interpreter) như Python, Ruby, JavaScript.




Hình 2: A Pure interpreter [3]

### 1.2.3 Hybrid Implementation Systems

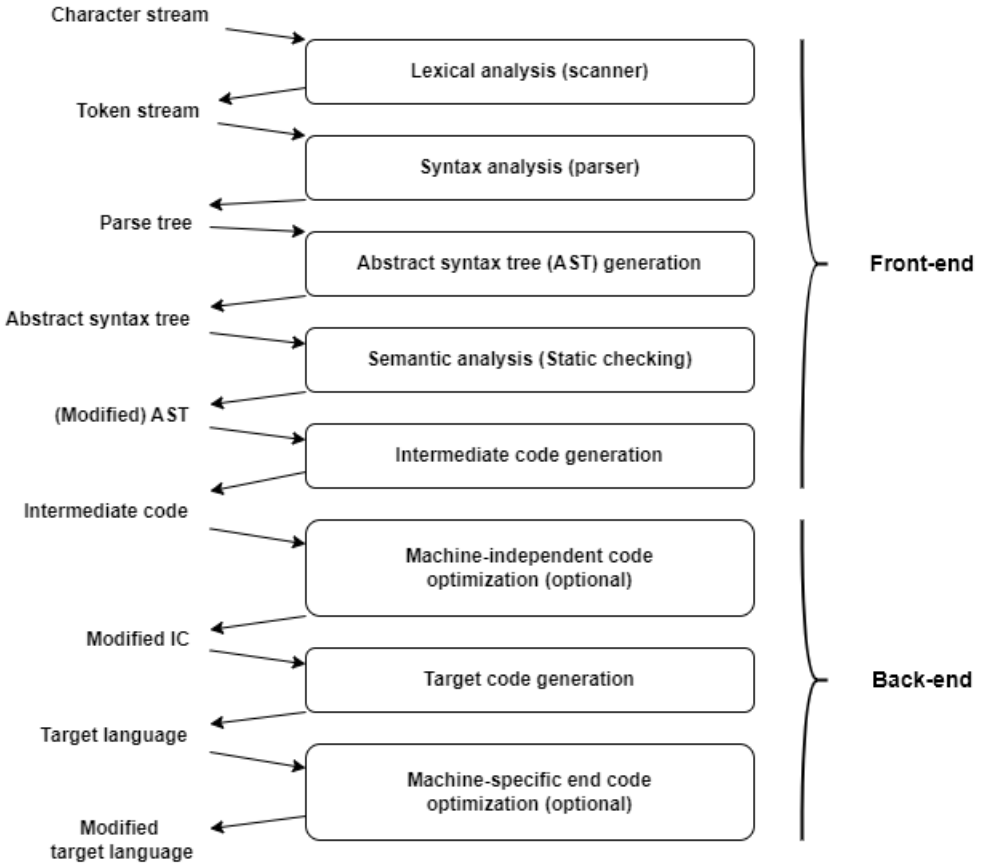
[6] Là một mô hình kết hợp của cả 2 quá trình trên. Nó tận dụng cả ưu điểm của trình biên dịch lẫn trình thông dịch trong quá trình dịch từ ngôn ngữ cấp cao sang mã máy. Một cải tiến của hệ thống này có thể kể đến đó là Just-in-time (JIT) compliers.




Hình 3: A Hybrid system [3]

## 1.3 Cấu trúc của một trình biên dịch

Trong bài báo cáo này sẽ quan tâm đến việc sinh mã trung gian cho một trình biên dịch sẽ đi chi tiết hơn vào cấu trúc của một trình biên dịch. Mỗi quá trình được giải thích ngắn gọn dưới




Hình 4: Cấu trúc của một trình biên dịch [3]

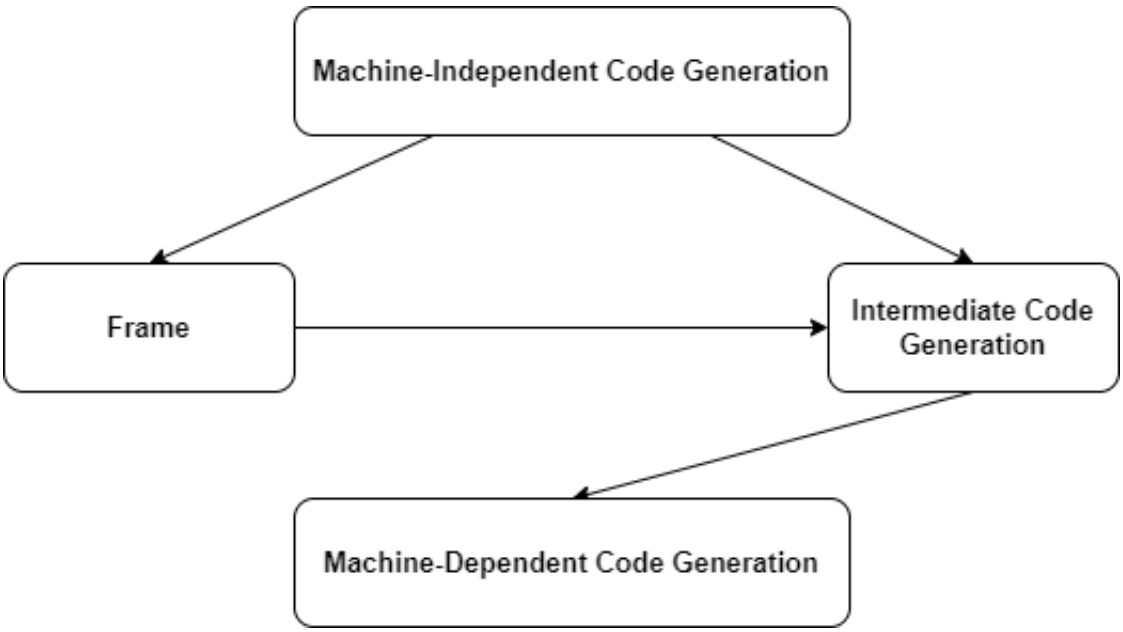
đây:

- **Lexical analysis**: hay còn gọi là phân tích từ vựng là quá trình phân tích và chuyển đổi chuỗi ký tự đầu vào thành các đơn vị từ vựng (tokens), nhằm tạo điều kiện cho việc phân tích cú pháp.
- **Syntax analysis**: hay còn gọi là phân tích cú pháp là quá trình phân tích và kiểm tra tính hợp lệ của cấu trúc ngữ pháp của chương trình, để đảm bảo rằng chương trình được viết theo đúng quy tắc của ngôn ngữ lập trình.
- **Abstract syntax tree generation**: là quá trình tạo ra một cây cú pháp trừu tượng (AST) từ các đơn vị từ vựng đã được phân tích và xác định cú pháp. AST giúp đại diện cho cấu trúc ngữ pháp của chương trình trong một cấu trúc dữ liệu phù hợp, giúp dễ dàng phân tích và thực thi chương trình.

- **Semantic analysis**: hay còn gọi là phân tích ngữ nghĩa là quá trình kiểm tra tính hợp lệ của ý nghĩa của chương trình dựa trên ngữ nghĩa của ngôn ngữ lập trình. Nó bao gồm kiểm tra tính đúng đắn của các kiểu dữ liệu, phạm vi của biến, tính xác thực của các toán tử và hàm, v.v. Quá trình này giúp đảm bảo rằng chương trình có ý nghĩa và thực thi đúng như ý đồ của người lập trình.

  Các khái niệm còn lại sẽ được mô tả ở dưới phần kiến trúc tổng quát của quá trình sinh mã.

# 2 Kiến trúc tổng quát của quá trình sinh mã




Hình 5: Kiến trúc quá trình tạo mã [3]

## 2.1 Machine-Dependent Code Generation

Machine dependent code generation là quá trình tạo ra mã máy tính (executable code) cho một kiến trúc phần cứng và hệ điều hành cụ thể. Điều này đảm bảo rằng phần mềm được tối ưu hóa cho các hệ thống cụ thể và có thể chạy hiệu quả trên chúng.

## 2.2 Intermediate Code Generation

Intermediate code generation là quá trình tạo ra một ngôn ngữ trung gian, ví dụ như mã bytecode, từ mã nguồn của một chương trình. Mã trung gian này được sử dụng để tối ưu hóa và dễ dàng di động khi triển khai chương trình trên nhiều nền tảng khác nhau.

## 2.3 Frame

Frame là một khái niệm trong lập trình máy tính, ám chỉ một khu vực bộ nhớ được sử dụng để lưu trữ các biến cục bộ, các địa chỉ trả về và các giá trị trung gian khác trong quá trình thực thi của một hàm hoặc thủ tục.

## 2.4 Machine-Independent Code Generation

Machine independent code generation là quá trình tạo ra mã nguồn (code) có thể chạy trên nhiều loại máy tính khác nhau, độc lập với kiến trúc phần cứng và hệ điều hành mà nó chạy trên đó. Mục đích của quá trình này là để đảm bảo tính di động và sự dễ dàng trong việc triển khai và phát triển phần mềm trên nhiều nền tảng khác nhau mà không cần phải viết lại mã nguồn cho mỗi nền tảng.

# 3 Task 1: Sinh mã JVM cho biểu thức 2 ngôi với toán tử cộng

## 3.1 MT22.g4

```
grammar MT22;

@lexer::header {
from lexererr import *
}

options{
        language=Python3;
}

program  : mptype 'main' LB RB LP body? RP EOF ;

mptype: INTTYPE | VOIDTYPE ;

body: funcall SEMI;

exp: exp ADD exp1 | exp1;
exp1: LB exp RB | funcall | INTLIT | FLOATLIT;

funcall: ID LB exp? RB ;

INTTYPE: 'int' ;

VOIDTYPE: 'void'  ;

ID: [a-zA-Z]+ ;

INTLIT: [0-9]+;
FLOATLIT: INTLIT DECPART EXPPART {
self.text = self.text.replace("_", "");
} | INTLIT DECPART {
self.text = self.text.replace("_", "");
}| INTLIT EXPPART  {
self.text = self.text.replace("_", "");
}| DECPART EXPPART;
fragment DECPART: '.' [0-9]*;
fragment EXPPART: [eE] [-+]? [0-9]+ ;

ADD: '+';

LB: '(' ;
```

```
RB: ')' ;

LP: '{';

RP: '}';

SEMI: ';' ;

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines


ERROR_CHAR: .;
UNCLOSE_STRING: .;
ILLEGAL_ESCAPE: .;
```

**Giải thích**

- MT22.g4 là 1 file gồm các tập luật sinh với các kí tự kết thúc và không kết thúc, biểu diễn một ngôn ngữ mới được định nghĩa lại. Một cách tổng quát nhất, trong đó chương trình chỉ bao gồm một hàm 'main' không có tham số đầu vào, có giả trị trả về là là kiểu số nguyên (integer type) hoặc kiểu void (void type) và nội dung của hàm được thể hiện trong phần body (có thể không có) được bao bởi cặp dấu '{' '}'.
- Nội dung của phần body chỉ bao gồm 1 dòng lệnh đó là lời gọi hàm ('funcal'), một lời gọi hàm thì bắt đầu bởi một danh hiệu (ID) và 1 biểu thức (có thể không có) được bao bởi cặp dấu '(' ')'.
- Một biểu thức thì thể hiện hoặc là một biểu thức 2 ngôi của toán tử cộng, hoặc là một giá trị nguyên, một giá trị thực, hoặc là một lời gọi hàm khác (funcall), hoặc là một biểu thức con.
- Trong ngữ cảnh lần này, một phép toán cộng là một phép toán chỉ giữa các giá trị của các literal, nghĩa là chỉ phép cộng giữa số nguyên với số nguyên, số thực với số thực và số nguyên với số thực. Bên cạnh đó phép cộng cũng được thể hiện dưới dạng kết hợp trái, nghĩa là sẽ được cộng dồn từ trái sang nếu có nhiều toán tử trong một biểu thức.

## 3.2 CodeGenerator.py

Sau khi đã định nghĩa ngữ cảnh cho ngôn ngữ mới là MT22, tiếp tục đến với quá trình sinh ra cây Abstract syntax tree (AST). Quá trình sinh cây sẽ được thể hiện rõ hơn trong file ASTGeneation.py.

Vì đề tài tập trung chủ yếu đến quá trình sinh mã trung gian nên quá trình kiểm tra ngữ nghĩa (semantic analysis) có thể sẽ tạm bỏ qua trong lần này, và tiếp tục quá trình hiện thực các phương thức cần thiết cho quá trình sinh mã.

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

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

**Giải thích**: visitIntegerLit() là một phương thức của lớp CodeGenVisitor cho phép việc viếng thăm vào một integer literal để có thể tạo ra một mã lệnh tương ứng của Jasmin. Ở đây sẽ trả về 2 giá trị, giá trị thứ nhất là lệnh push 1 số nguyên lên stack của Jasmin thông qua việc gọi API emitPUSHICONST của Emitter, và giá trị thứ 2 là một object IntegerType()

```
    def visitFloatLit(self, ast, o):
        #ast: FloatLiteral
        ctxt = o
        frame = ctxt.frame
        return self.emit.emitPUSHFCONST(str(ast.val), frame), FloatType()
```

**Giải thích**: tương tự như phương thức visitIntegerLit(), visitFloatLit() cũng trả về một lệnh push 1 số thực lên stack của Jasmin thông qua API emitPUSHFCONST, và 1 object FloatType()

```
    def visitBinExpr(self, ast, o):
        # ast: BinExpr
        # o: any
        ctxt = o

        lexem = ast.op
        left = ast.left
        right = ast.right
        str_left, typ_left = self.visit(left,ctxt)

        str_right, typ_right = self.visit(right,ctxt)
        if type(typ_left) is FloatType or type(typ_right) is FloatType:
            self.emit.printout(str_left)
            if type(typ_left) is IntegerType:
                str_i2f = self.emit.emitI2F(ctxt.frame)
                self.emit.printout(str_i2f)

            self.emit.printout(str_right)
            if type(typ_right) is IntegerType:
                str_i2f = self.emit.emitI2F(ctxt.frame)
                self.emit.printout(str_i2f)
            return self.emit.emitADDOP(lexem, FloatType(), ctxt.frame),
            ↪  FloatType()
```

```
        else:
            self.emit.printout(str_left)
            self.emit.printout(str_right)
            return self.emit.emitADDOP(lexem, IntegerType(),ctxt.frame),
            ↪  IntegerType()
```

**Giải thích**: visitBinExpr cũng là một phương thức của lớp CodeGenVisitor cho phép viếng tham vào một biểu thức 2 ngôn, trong ngữ cảnh này cụ thể là phép cộng. Dưới đây là phần ý tưởng hiện thực của phương thức này.

- Visit lần lượt vào 2 giá trị 'left' và 'right' của cây ast, giá trị trả về lưu vào các biến như trên.
- Sẽ có 2 trường hợp xảy ra, đó là cộng 2 giá trị nguyên và cộng 2 giá trị có tồn tại ít nhất một số thực.
- Đối với trường hợp đầu tiên, ta chỉ việc ghi các lệnh push giá trị lên stack từ các giá trị visit trả về lên một buffer (Sau khi các lệnh được ghi lên hết, buff sẽ được ghi xuống file Jasmin và thực hiện biên dịch). Quá trình này được thực hiện thông qua API printout() của Emitter. Cuối cùng một lệnh add 2 số được gọi thông qua API ADDOP()
- Đối với trường hợp còn lại, ta cũng làm tương tự, tuy nhiên sẽ kiểm tra nếu giá trị nào là nguyên sẽ được chuyển đổi thành giá trị thực trước khi được push lên stack. Quá trình này sẽ được thực hiện thông qua API emitI2F() của Emitter.

# 4 Task 2: Sinh mã MIPS cho toán tử cộng. Hiện thực giải thuật cấp phát thanh ghi

## 4.1 Quá trình sinh mã MIPS

Một số sửa đổi ở các file để thuận tiện quá quá trình sinh mã MIPS

### 4.1.1 MachineCode.py

```
class MIPSCode:
    END = "\n"
    INDENT = "\t"
    curRes = 0
    def emitILOAD(self, i):
        MIPSCode.curRes += 1
        reg = '$vR' + str(MIPSCode.curRes)
        instr =  MIPSCode.INDENT + "li "+ reg + "," + str(i) +
        ↪  MIPSCode.END
        return instr, reg

    def emitIADD(self, r, i):
        MIPSCode.curRes +=1
        reg = '$vR' + str(MIPSCode.curRes)
        instr = MIPSCode.INDENT + "addi " + reg + "," + r + "," + str(i)
        ↪  + MIPSCode.END
        return instr, reg
    def emitADD(self, r1, r2):
        MIPSCode.curRes +=1
        reg = '$vR' + str(MIPSCode.curRes)
        instr = MIPSCode.INDENT + "add " + reg + "," + r1 + "," + r2 +
        ↪  MIPSCode.END
        return instr, reg

    def emitMOVE(self, r1, r2):
        return MIPSCode.INDENT + "move " + r1 + "," + r2 + MIPSCode.END

    def emitICALLPRINT(self):
        return MIPSCode.INDENT + "li $v0,1" + MIPSCode.END

    def emitSYSCALL(self):
        return MIPSCode.INDENT + "syscall" + MIPSCode.END
```

**Giải thích:**

- Sử dụng biến curRes để hỗ trợ việc thêm một thanh ghi ảo mới, thanh ghi ảo có dạng $vRi
- emitILOAD(): trả về lệnh mips load giá trị nguyên vào thanh ghi ảo.

- emitIADD(): cộng giá trị nguyên với giá trị của một thanh ghi, kết quả được lưu vào thanh ghi ảo khác.
- emitADD(): thực hiện cộng 2 thanh ghi, giá trị kết quả lưu vào một thanh ghi ảo khác.
- emitMOVE(r1,r2): chuyển giá trị từ thanh ghi r2 sang thanh ghi r1
- emitICALLPRINT(): lệnh load giá trị 1 vào thanh ghi $v0 để in giá trị đối số ra màn hình console
- emitSYSCALL(): gọi lệnh syscall

### 4.1.2 Emitter.py

```
    def __init__(self, filename):
        self.filename = filename
        self.buff = list()
        self.jvm = MIPSCode()
        self.curLine = 0
        self.liveRanges = {}
        self.registers = ["$zero","$at","$v0","$v1","a0","$a1","$a2",
                          "$a3","$t0","$t1","$t2","$t3","$t4","$t5",
                          "$t6","$t7","$s0","$s1","$s2","$s3","$s4",
                          "$s5","$s6","$s7","$t8","$t8","$k0","$k1",
                          "$gp","$sp","$fp","$ra"]
```

**Giải thích:**

- self.jvm = MIPSCode(): thay thế mã jvm bằng mã MIPS
- self.curLine: theo dỏi thứ tự của dòng lệnh đang xét
- self.liveRanges: lưu thời gian tồn tại của các thanh ghi có trong chương trình.
- self.registers: list chứa 32 thanh ghi của mars mips.

```
    def emitILOAD(self, i):
        instr, retReg =  self.jvm.emitILOAD(i)
        if retReg in self.liveRanges:
            self.liveRanges[retReg][1] = self.curLine - 1
        else:
            self.liveRanges[retReg] = [self.curLine, self.curLine]

        self.curLine += 1
        return instr, retReg
```

**Giải thích:** Dùng api emitILOAD từ file MachineCode, trả về 2 giá trị là câu lệnh load số nguyên và thanh ghi kết quả. Đồng thời thực hiện việc xác định và cập nhật thời gian sống của các thanh ghi có trong dòng lệnh.

```
    def emitIADD(self, r, i):
        instr, retReg =  self.jvm.emitIADD(i)
        for reg in [r, retReg]:
            if reg in self.liveRanges:
                self.liveRanges[reg][1] = self.curLine - 1
            else:
                self.liveRanges[reg] = [self.curLine, self.curLine]

        self.curLine += 1
        return instr, retReg
```

**Giải thích:** Dùng api emitIADD từ file MachineCode, trả về 2 giá trị là câu lệnh cộng số nguyên với thanh ghi và thanh ghi kết quả. Đồng thời thực hiện việc xác định và cập nhật thời gian sống của các thanh ghi có trong dòng lệnh.

```
    def emitADD(self, r1, r2):
        instr, retReg =  self.jvm.emitADD(r1, r2)

        for reg in [r1, r2, retReg]:
            if reg in self.liveRanges:
                self.liveRanges[reg][1] = self.curLine - 1
            else:
                self.liveRanges[reg] = [self.curLine, self.curLine]

        self.curLine += 1
        return instr, retReg
```

**Giải thích:** Dùng api emitADD từ file MachineCode, trả về 2 giá trị là câu lệnh cộng 2 thanh ghi và thanh ghi kết quả. Đồng thời thực hiện việc xác định và cập nhật thời gian sống của các thanh ghi có trong dòng lệnh.

```
    def emitMOVE(self, r1, r2):
        for reg in [r1, r2]:
            if reg in self.liveRanges:
                self.liveRanges[reg][1] = self.curLine
            else:
                self.liveRanges[reg] = [self.curLine, self.curLine]

        self.curLine += 1
        return self.jvm.emitMOVE(r1, r2)
```

**Giải thích:** Dùng api emitMOVE từ file MachineCode, trả về 2 giá trị là câu lệnh chuyển giá trị từ thanh ghi r1 sang r2 và thanh ghi kết quả. Đồng thời thực hiện việc xác định và cập nhật thời gian sống của các thanh ghi có trong dòng lệnh.

```
    def emitICALLPRINT(self):
        self.curLine += 1
        return self.jvm.emitICALLPRINT()
```

```
    def emitSYSCALL(self):
        self.curLine += 1
        return self.jvm.emitSYSCALL()
```

### 4.1.3 CodeGenerator.py

Các method được thiết kế lại để đơn giản và chỉ phù hợp cho phép cộng các số nguyên nên sẽ không thể hiện được các ràng buộc chặt chẽ về các điều kiện khác.

```
    def visitProgram(self, ast, c):
        #ast: Program
        #c: Any

        e = SubBody(self.env)
        for x in ast.decls:
            e = self.visit(x, e)
        # generate default constructor
        # self.genMETHOD(FuncDecl("<init>", None, list(), None,
        ↪  BlockStmt( list())), c)

        self.emit.mapping()
        self.emit.emitEPILOG()
        return c
```

**Giải thích:**

- self.emit.mapping(): thực hiện việc ánh xạ các thanh ghi ảo sang các thanh ghi vật lý với giải thuật cấp phát thanh ghi Graph Coloring.
- self.emitEPILOG(): thực hiện việc ghi tất cả các dòng lệnh mips từ buffer xuống file .asm

```
    def visitFuncCall(self, ast, o):
        #ast: FuncCall
        #o: Any

        ctxt = o
        nenv = ctxt.sym
        # sym = self.lookup(ast.name, nenv, lambda x: x.name)
        # cname = sym.value.value
        # ctype = sym.mtype
        in_ = ("", list())
```

```
        resReg = None
        for x in ast.args:
            str1, typ1 = self.visit(x, Access(nenv, False, True))
            in_ = (in_[0] + str1[0], in_[1].append(typ1))
            resReg = str1[1]
        self.emit.printout(in_[0])
        self.emit.printout(self.emit.emitMOVE("$a0", resReg))
        self.emit.printout(self.emit.emitICALLPRINT())
        self.emit.printout(self.emit.emitSYSCALL())
```

**Giải thích:**

- Sau khi visit các đối số và được kết quả sẽ thực hiện quá trình in kết quả ra màn hình
- Sử dụng hàm emitMove("$a0", resReg) để thực hiện chuyển giá trị từ thanh ghi kết quả vào thanh ghi đối số là $a0
- Sử dụng 2 hàm emitICALLPRINT() và emitSYSCALL() để thực hiện việc in số nguyên là giá trị từ đối số $a0.

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

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

```
    def visitBinExpr(self, ast, o):

        lexem = ast.op
        left = ast.left
        right = ast.right

        (instrLeft, regLeft), typLeft = self.visit(left,o)
        self.emit.printout(instrLeft)

        (instrRight, regRight), typRight = self.visit(right,o)
        self.emit.printout(instrRight)

        return self.emit.emitADD(regLeft,regRight), IntegerType()
```

## 4.2 Một số giải thuật cấp phát thanh ghi

**Cấp phát thanh ghi** là một phương pháp quan trọng trong giai đoạn cuối cùng của trình biên dịch. Các thanh ghi có tốc độ truy cập nhanh hơn nhiều so với bộ nhớ cache. Tuy nhiên các thanh ghi thường có kích thước nhỏ (chỉ khoảng vài trăm Kb). Do đó việc sử dụng tối thiểu các

thanh ghi cho việc cấp phát cấp phát biến là cần thiết. Có một vài giải thuật cấp phát thanh ghi, và dưới đây là ba trong số các giải thuật phổ biến được sử dụng.

1. **Naive Register Allocation**[2]
2. **Linear Scan Algorithm** [2]
3. **Graph Coloring (Chaitin's Algorithm)** [2]

Trong bài báo cáo này sẽ sử dụng giải thuật Chaitin để hiện thực cho quá trình cấp phát thanh ghi.

## 4.3 Graph Coloring (Chaitin's Algorithm) [2]

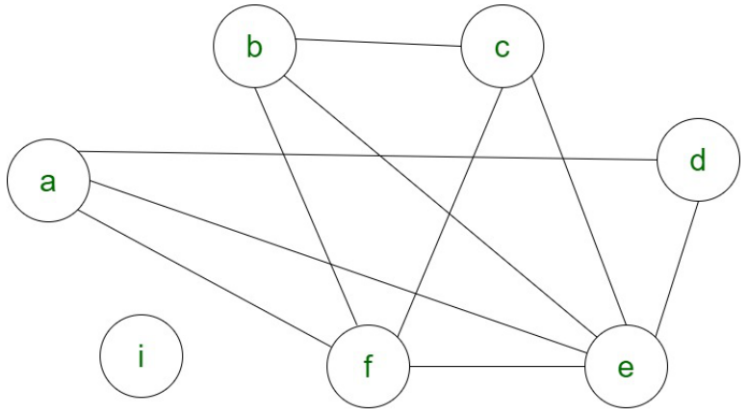
### 4.3.1 Ý tưởng

- Quá trình cấp phát thanh ghi được xem như là một bài toán tô màu đồ thị.
- Các node đại diện cho "thời gian sống" (live range) của biến.
- Các cạnh thể hiện sự kết nối giữa 2 thời gian sống. Nghĩa là 2 node bất kì có khoảng thời gian sống chồng chéo nhau (cùng tồn tại trong 1 khoảng thời gian) sẽ có một cạnh kết nối giữa chúng.
- Gán màu cho các node sao cho không có 2 node liền kề nào có chung màu
- Số lượng màu sắc đại diện cho số thanh ghi tối thiểu được yêu cầu sử dụng trong cả đoạn chương trình.

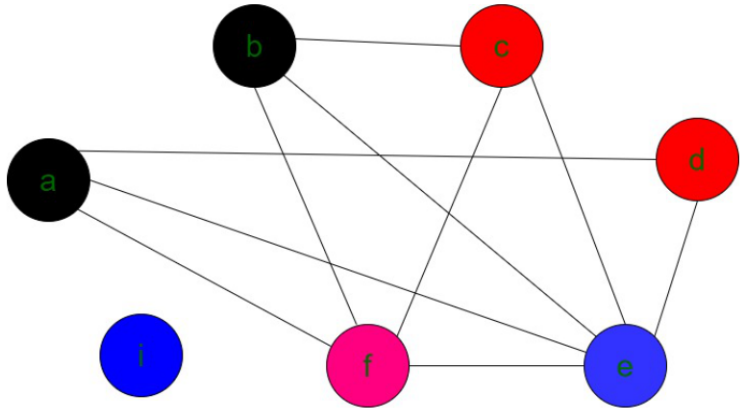
**Một đồ thị có k màu sẽ được ánh xạ đến k thanh ghi**

### 4.3.2 Các bước hiện thực

1. Chọn một node bất kì có bậc nhỏ hơn k (k là giá sự khởi tạo ban đầu).
2. Đưa node đó lên stack và xóa tất cả cạnh xuất phát từ nó.
3. Kiểm tra xem những cạnh còn lại có bậc nhỏ hơn k không. Nếu có thì chuyển đến bước 5, không thì chuyển đến bước 7.
4. Nếu có ít nhất một node còn lại có bậc nhỏ hơn k, đưa node đó lên stack.
5. Nếu không còn node nào để đẩy vào stack và tất cả các node đã có mặt trong stack, bắt đầu lấy từng node ra khỏi stack và tô màu cho node đó sao cho màu của node đó khác với màu của tất cả các node kề nó.
6. Số màu được gán cho các node của đồ thị sẽ là số lượng thanh ghi tối thiểu cần thiết để lưu trữ các giá trị của các node đó.
7. Chuyển một số node vào bộ nhớ ngoài dựa trên thời gian sống của chúng và sau đó thử lại thuật toán với cùng giá trị k. Nếu vẫn không thể tô màu được tất cả các node, tăng giá trị k lên 1 và thử lại toàn bộ thuật toán từ bước 1.




Hình 6: Before Coloring [2]




Hình 7: Final Graph [2]

### 4.3.3 Ưu điểm

- Hiệu quả: Thuật toán Graph Coloring là một trong những thuật toán phân màu đồ thị hiệu quả nhất, giúp tối ưu hóa bộ nhớ trong quá trình biên dịch mã nguồn.
- Dễ triển khai: Thuật toán Graph Coloring có cấu trúc đơn giản và dễ triển khai, không yêu cầu nhiều kiến thức toán học phức tạp.
- Tính linh hoạt: Thuật toán Graph Coloring có thể được áp dụng cho nhiều loại đồ thị khác nhau, bao gồm đồ thị hướng và đồ thị vô hướng.
- Tính ổn định: Thuật toán Graph Coloring là một thuật toán ổn định và có thể được sử dụng trong nhiều ứng dụng khác nhau.

### 4.3.4 Nhược điểm

- Không đảm bảo tối ưu tuyệt đối: Thuật toán Graph Coloring có thể không đảm bảo tối ưu tuyệt đối, khiến cho bộ nhớ vẫn có thể không được tối ưu hoàn toàn.

- Không thể giải quyết mọi trường hợp: Thuật toán Graph Coloring không thể giải quyết được mọi trường hợp, ví dụ như các đồ thị có chứa chu trình lớn.
- Độ phức tạp cao: Độ phức tạp của thuật toán Graph Coloring là $O(n^2)$, nghĩa là nó có thể trở nên chậm khi xử lý các đồ thị lớn.

## 4.4 Áp dụng giải thuật Graph coloring cho quá trình cấp phát thanh ghi

### 4.4.1 Tạo file RegisterAllocation.py

```
class Graph:
    def __init__(self, liveRanges):
        self.liveRanges = liveRanges
        self.adjList = self.initAdjList()

    def initAdjList(self):
        adjList = {}

        for node in self.liveRanges.keys():
            adjList[node] = []

        for u in self.liveRanges.keys():
            for v in self.liveRanges.keys():
                if v != u:
                    if (self.liveRanges[v][1] >= self.liveRanges[u][0])
                    ↪  and (self.liveRanges[u][1] >=
                    ↪  self.liveRanges[v][0]):
                        adjList[u].append(v)

        return adjList

    def nodes(self):
        return self.adjList.keys()
    def neighbors(self, node):
        return self.adjList[node]
```

**Giải thích:** Xây dựng đồ thị thể hiện sự liên kết giữa các thanh ghi.

1. Khởi tạo một empty dictionary (adjList) để lưu trữ danh sách kề. Sau đó, với mỗi node trong liveRanges, phương thức thêm một key tương ứng vào adjList và gán giá trị là một empty list.
2. Kiểm tra mỗi cặp node (u, v) trong liveRanges và thêm v vào danh sách kề của u nếu v và u không giống nhau và phạm vi sống của chúng có sự chồng chéo. Điều này có nghĩa là nếu một node u và một node v có thể sống cùng một thời điểm, thì chúng được kết nối với nhau trong đồ thị.
3. Trả về danh sách kề được xây dựng.

```
class Chaitin:
    def __init__(self, graph, k):
        self.graph = graph
        self.k = k

    def color_graph(self):
        color_map = {}

        # Assign initial colors to all nodes
        for node in self.graph.nodes():
            color_map[node] = None

        # Iteratively select nodes to color
        for node in self.graph.nodes():
            available_colors = set(range(self.k))

            # Check the colors of neighboring nodes
            for neighbor in self.graph.neighbors(node):
                if neighbor in color_map:
                    color = color_map[neighbor]
                    if color in available_colors:
                        available_colors.remove(color)

            # Assign the lowest available color to the node
            if len(available_colors) > 0:
                color_map[node] = min(available_colors)
            else:
                color_map[node] = self.k
                self.k += 1

        return color_map
```

**Giải thích:** Hiện thực lại giải thuật Graph Coloring theo các bước ở **4.3.2**.

### 4.4.2 Thêm các hàm cần thiết trong file Emitter.py

```
    def registerAllocation(self):
        graph = Graph(self.liveRanges)

        chaitin = Chaitin(graph, 3)

        return chaitin.color_graph()
```

**Giải thích:** Thực hiện việc cấp phát thanh ghi với giải thuật Chaitin, truyền vào graph là directory chứa thời gian sống của các thanh ghi ảo, 3 là giá trị khởi tạo của k. Trả về là 1 directory chứa các cặp key-value tương ứng với các giá trị thanh ghi ảo - vật lý.

```
    def mapping(self):
        color_graph = self.registerAllocation()
        for i in range(len(self.buff)):
            instr = self.buff[i]
            if "$vR" in instr:
                space = len(instr) - instr[::-1].find(" ") - 1
                paramList = instr[space + 1:-1].split(',')
                regs = list(filter(lambda x: "$vR" in x, paramList))

                for reg in regs:
                    newReg = self.registers[color_graph[reg] + 8]
                    self.buff[i] = self.buff[i].replace(reg, newReg)
```

**Giải thích:** Phương thức này thực hiện việc ánh xạ các thanh ghi ảo sang các thanh ghi vật lý sau khi thực hiện việc cấp phát thanh ghi.

### 4.4.3 Kết quả

**TestCase:**

```
    def test_simple(self):
        input = """void main() {putInt((1+2) + (2+4) + (5+6));}"""
        expect = "20"
        self.assertTrue(TestCodeGen.test(input, expect, 505))
```

**Kết quả sau quá trình sinh mã MIPs với các thanh ghi ảo:**

```
        li $vR1,1
        li $vR2,2
        add $vR3,$vR1,$vR2
        li $vR4,2
        li $vR5,4
        add $vR6,$vR4,$vR5
        add $vR7,$vR3,$vR6
        li $vR8,5
        li $vR9,6
        add $vR10,$vR8,$vR9
        add $vR11,$vR7,$vR10
        move $a0,$vR11
        li $v0,1
        syscall
```

**Thời gian sống của các thanh ghi ảo:**

```
{'$vR1': [0, 1], '$vR2': [1, 1], '$vR3': [2, 5],
'$vR4': [3, 4], '$vR5': [4, 4], '$vR6': [5, 5],
'$vR7': [6, 9], '$vR8': [7, 8], '$vR9': [8, 8],
'$vR10': [9, 9], '$vR11': [10, 11], '$a0': [11, 11]}
```

**Sau khi cấp phát thanh ghi:**

```
{'$vR1': 0, '$vR2': 1, '$vR3': 0, '$vR4': 1,
'$vR5': 2, '$vR6': 1, '$vR7': 0, '$vR8': 1,
'$vR9': 2, '$vR10': 1, '$vR11': 0, '$a0': 1}
```

**Kết quả sau khi mapping:**

```
        li $t0,1
        li $t1,2
        add $t0,$t0,$t1
        li $t1,2
        li $t2,4
        add $t1,$t1,$t2
        add $t0,$t0,$t1
        li $t1,5
        li $t2,6
        add $t1,$t1,$t2
        add $t0,$t0,$t1
        move $a0,$t0
        li $v0,1
        syscall
```

# 5 Task 3:

# Tài liệu

[1] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)

[2] GeeksforGeeks: Register allocation algorithms in compiler design (2023418), https://www.geeksforgeeks.org/register-allocation-algorithms-in-compiler-design/

[3] Nguyen, P.H., Tran, D.B.N.: Introduction to programming languages and compilers (March 2022), slides presented at Principal Programing Language course, Ho Chi Minh University of Technology

[4] Parrish, A.: Pure interpretation: executing python without compiling it. ACM SIGPLAN Notices **49**(3), 125–134 (2014)

[5] Wikipedia: Programming languages — wikipedia, the free encyclopedia (2023), https://en.wikipedia.org/wiki/Programming_language, [Online; accessed 28-March-2023]

[6] Zhang, Y., Wang, Y., Wang, Y.: Design and implementation of a hybrid compiler-interpreter system. In: Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. pp. 139–148. ACM (2005)