

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



NGUYÊN LÝ NGÔN NGỮ LẬP TRÌNH - Mở rộng (CO300C)

Báo cáo Bài tập lớn mở rộng

Giai đoạn 1 SINH MÃ JVM CHO PHÉP CỘNG HAI SỐ NGUYÊN DƯƠNG

GVHD: Nguyễn Hứa Phùng

Sinh viên: Nguyễn Thái Tân 2112256

Thành phố Hồ Chí Minh, 03/2024



Mục lục

1	Phần mở đầu	2
2	Giai đoạn 1: Code generation	3
2.1	Nhắc lại về Code generation	3
2.2	Hiện thực	3
2.2.1	Phân tích từ vựng và ngữ pháp	3
2.2.2	Hiện thực Code generation cho phép cộng	4
2.3	Kiểm thử	6
2.4	Nhận xét	8

1 Phần mở đầu

Các giai đoạn trong quá trình biên dịch có thể được nhóm thành hai nhóm chính: front-end và back-end. Đầu ra của front-end là mã trung gian, mã này có thể là AST. Đầu ra của back-end là mã máy. Trong phạm vi của môn học này, Java byte code (đoạn mã sử dụng để chạy trên Java Virtual Machine (JVM)) sẽ được sử dụng làm đầu ra của giai đoạn back-end. JVM là một stack-based và chấp nhận mô hình hướng đối tượng. Việc tạo mã stack-based sẽ dễ dàng hơn, tuy nhiên sẽ cần sự đánh đổi về hiệu suất.

Trong bài tập mở rộng này, chúng ta sẽ sinh mã cho một máy ảo register-based dựa trên kiến trúc MIPS (Microprocessor without Interlocked Pipelined Stages), cho ngôn ngữ *ZCode*.

Bài tập mở rộng này gồm 3 giai đoạn chính:

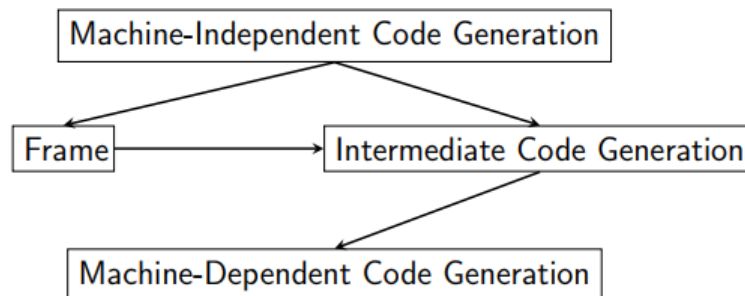
1. Dựa trên *initial code*, sinh mã JVM cho phép cộng hai số nguyên dương.
2. Thiết kế lại *initial code* để có thể sinh mã MIPS (thay vì sinh mã JVM) cho cùng input trong giai đoạn 1. Giai đoạn này cũng yêu cầu việc hiện thực gán các thanh ghi và quản lý các biến.
3. Dựa vào giai đoạn 2, hoàn thiện việc sinh mã MIPS cho một chương trình hoàn chỉnh được viết bằng ngôn ngữ *ZCode*.

2 Giai đoạn 1: Code generation

Trong giai đoạn 1, chúng ta sẽ làm quen với Code generation bằng cách hiện thực việc sinh mã JVM cho phép tính nhị phân, cụ thể là phép cộng cho 2 số nguyên dương.

Với mục đích làm quen với Code generation, chúng ta chỉ hiện thực trên một ngôn ngữ đơn giản. Cụ thể, mỗi chương trình chỉ gồm một hàm *main()*, trong thân hàm *main()* chỉ có tối đa một câu lệnh. Việc phân tích từ vựng và ngữ pháp của ngôn ngữ này đã được hiện thực sẵn trong *initial code*.

2.1 Nhắc lại về Code generation



Hình 1: Code generation design

Có 4 khối chính trong Code Generation

1. Machine-Dependent Code Generation: Thành phần phụ thuộc hoàn toàn vào máy mà chúng ta sinh mã ra. Trong trường hợp này là Jasmin.
2. Intermediate Code Generation: Thành phần phụ thuộc vào cả mã nguồn và mã máy. Được hiện thực trong class *Emitter*.
3. Frame: Các công cụ được sử dụng để quản lý thông tin được sử dụng để tạo mã cho một phương thức (như Labels, Local variable array, Operand stack). Được hiện thực trong class *Frame*.
4. Machine-Independent Code Generation: Dựa trên ngôn ngữ nguồn. Sử dụng các phương thức trong *Frame* và *Emitter*.

2.2 Hiện thực

2.2.1 Phân tích từ vựng và ngữ pháp

Đầu tiên, ta quan tâm đến biểu thức phép cộng trong phân tích ngữ pháp. Trong toán học, phép cộng được thực hiện từ trái sang phải, vì vậy ta sẽ hiện thực *left-associated*, cụ thể như sau:

```
1 exp: exp ADDOP exp1 | exp1;  
2 exp1: funcall | INTLIT;  
3  
4 ADDOP: '+';
```

Như đã nói, ta chỉ quan tâm đến số nguyên dương, nên ta sẽ hiện thực *INTLIT*

```
1 INTLIT: [0-9]+;
```

Tiếp theo, ta sẽ tiến hành hiện thực AST. Trong *AST.py* đã có *class BinExpr(Expr)*, ta sẽ dùng class này để sinh cây cho phép cộng.

```
1 class BinExpr(Expr):  
2     def __init__(self, op: str, left: Expr, right: Expr):  
3         self.op = op  
4         self.left = left  
5         self.right = right
```

- *op*: Có kiểu String, đây chính là phép tính của biểu thức nhị phân. Ta sẽ lấy giá trị này từ *ctx.ADDOP().getText()* (cần *getText()* để chuyển về dạng string).
- *left*: Có kiểu Expr, là toán hạng bên trái của biểu thức nhị phân. Với cách hiện thực *left-associated* như trên, giá trị ở đây chính là *ctx.exp()*.
- *right*: Có kiểu Expr, là toán hạng bên phải của biểu thức nhị phân. Tương tự *left*, giá trị ở đây chính là *ctx.exp1()*.

Với những ý tưởng trên, ta hiện thực cụ thể như sau:

```
1 def visitExp(self, ctx: ZCodeParser.ExprContext):  
2     if ctx.ADDOP():  
3         exp = self.visit(ctx.exp())  
4         exp1 = self.visit(ctx.exp1())  
5         return BinExpr(ctx.ADDOP().getText(), exp, exp1)  
6     return self.visit(ctx.exp1())  
7  
8 def visitExp1(self, ctx: ZCodeParser.ExprContext):  
9     if (ctx.funcall()):  
10        return self.visit(ctx.funcall())  
11    else:  
12        return IntegerLit(int(ctx.INTLIT().getText()))
```

2.2.2 Hiện thực Code generation cho phép cộng

Nhiệm vụ chính của ta trong bài tập này là trong *CodeGenerator.py*, hiện thực hàm *def visitBinExpr(self, ast, c)*.

Trong *Emitter.py* đã cung cấp hàm *def emitADDOP(self, lexeme, in_, frame)*, dùng để sinh ra mã cho phép cộng. Cụ thể,

- *lexeme*: Có kiểu *String*, dùng để xác định là phép cộng hay phép trừ. Ta sẽ lấy giá trị này từ *ast.op*.
- *in_*: Có kiểu *Type*, dùng để xác định kiểu của số được truyền vào phép tính này là số nguyên hay số thực. Như đã đề cập, ta chỉ quan tâm đến số nguyên, vì thế ta chỉ cần truyền vào một đối tượng *IntegerType()*.
- *frame*: Có kiểu *Frame*. Ta chỉ cần truyền thẳng *c.frame* vào.
- Với input của bài tập này, vì ta chỉ quan tâm đến phép cộng hai số nguyên dương, nên *emitADDOP* sẽ sinh ra đoạn mã từ *jvm.emitIADD()*, có trong *MachineCode.py*, kết quả cuối cùng là *iadd*

```
1 def emitIADD(self):  
2     return JasminCode.INDENT + "iadd" + JasminCode.END
```

Tổng hợp những ý tưởng trên, ta hiện thực *def visitBinExpr(self, ast, c)* cụ thể như sau:

```
1 def visitBinExpr(self, ast, c):  
2     access = Access(c.frame, c.sym, False, True)  
3     lCode, lType = self.visit(ast.left, access)  
4     rCode, rType = self.visit(ast.right, access)  
5     return lCode + rCode + self.emit.emitADDOP(ast.op, IntegerType()  
        (), c.frame), IntegerType()
```

- Gọi đệ quy *visit* để đi sâu hơn vào cây AST. Kết quả khi truy cập vào phía bên trái của biểu thức nhị phân được lưu trữ trong *lCode* và *lType*, tương tự với phía bên phải được lưu trữ trong *rCode* và *rType*.
- Phương thức *visit* cho mỗi phía bên trái và bên phải đề cập bên trên đã được hiện thực trong *def visitIntegerLit(self, ast, o)* trong *initial code*. Phương thức này sinh mã cho *IntegerLit* bằng cách gọi hàm *emitPUSHICONST(ast.val, frame)* trong *Emitter.py*.

```
1 def visitIntegerLit(self, ast, o):  
2     #ast: IntLiteral  
3     #o: Any  
4  
5     ctxt = o  
6     frame = ctxt.frame  
7     return self.emit.emitPUSHICONST(ast.val, frame), IntegerType()
```

- *emitPUSHICONST* đã đề cập có thể được dùng để sinh mã cho cả số nguyên và chuỗi, tuy nhiên trong bài tập này ta chỉ quan tâm đến số nguyên. Như kiến thức đã học, với mỗi trường hợp cụ thể sẽ sinh mã khác nhau

1. Số nguyên trong $[0; 5]$: sinh bằng *jvm.emitICONST(i)*, kết quả là *iconst_*
2. Số nguyên trong $[6; 128]$: sinh bằng *jvm.emitBIPUSH(i)*, kết quả là *bipush*
3. Số nguyên trong $[128; 32768]$: sinh bằng *jvm.emitSIPUSH(i)*, kết quả là *sipush*

- Kết hợp mã được tạo: Theo kiến thức đã được học, để thực hiện phép tính giữa hai toán hạng, ta cần load chúng vào stack trước, sau đó mới thực hiện phép tính. Vì thế, khi trả về, ta cần kết hợp *lCode* và *rCode* trước, sau đó mới kết hợp với *emitADDOP*.
- Để return, ta cần thêm `IntegerType()`.

2.3 Kiểm thử

Đầu tiên, ta kiểm thử với một testcase đơn giản là $100 + 1$ để kiểm tra mã sinh ra có giống với mong muốn hay không.

```
1 def test_500(self):
2     input = """void main() {putInt(100 + 1);}"""
3     expect = "101"
4     self.assertTrue(TestCodeGen.test(input, expect, 500))
```

- Vì $100 \in [6; 128)$ nên mã sinh ra phải là *bipush 100*
- Vì $1 \in [0; 5]$ nên mã sinh ra phải là *iconst_1*
- Mã sinh ra cho phép cộng là *iadd*

Sau khi chạy chương trình, mã sinh ra là

```
1 .source ZCodeClass.java
2 .class public ZCodeClass
3 .super java.lang.Object
4
5 .method public static main([Ljava/lang/String;)V
6 .var 0 is args [Ljava/lang/String; from Label0 to Label1
7 Label0:
8     bipush 100
9     iconst_1
10    iadd
11    invokestatic io/putInt(I)V
12 Label1:
13    return
14 .limit stack 2
15 .limit locals 1
16 .end method
17
18 .method public <init>()V
19 .var 0 is this LZCodeClass; from Label0 to Label1
20 Label0:
21    aload_0
22    invokespecial java/lang/Object/<init>()V
23 Label1:
24    return
25 .limit stack 1
26 .limit locals 1
27 .end method
```

Ta chú ý vào dòng 8, 9, 10 trong đoạn mã trên. Kết quả đã như mong muốn.

Sau khi kiểm tra testcase đơn giản trên thành công, ta thiết kế nhiều testcase khác với số lượng cụ thể như sau.

STT	Chương trình được viết	Phép cộng	Số lượng
1	Cụ thể	Giữa hai số thuộc $[0; 5]$	5
2	Cụ thể	Giữa hai số thuộc $[6; 128)$	5
3	Cụ thể	Giữa hai số thuộc $[128; 32768)$	5
4	Cụ thể	Giữa nhiều số thuộc $[0; 32768)$	5
5	Bằng AST	Giữa hai số thuộc $[0; 5]$	5
6	Bằng AST	Giữa hai số thuộc $[6; 128)$	5
7	Bằng AST	Giữa hai số thuộc $[128; 32768)$	5
8	Bằng AST	Giữa nhiều số thuộc $[0; 32768)$	5

Bảng 1: Thiết kế testcase

Ví dụ về một số testcase:

```

1 def test_509(self):
2     input = """void main() {putInt(64+98);}"""
3     expect = "162"
4     self.assertTrue(TestCodeGen.test(input, expect, 509))

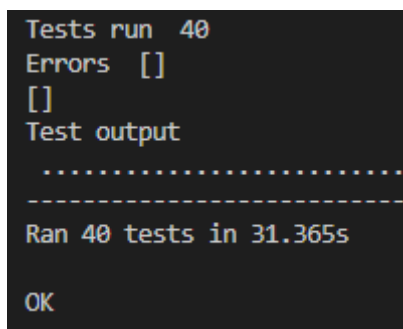
```

```

1 def test_521(self):
2     input = Program([
3         FuncDecl("main", VoidType(), [], None, BlockStmt([
4             FuncCall("putInt", [BinExpr("+", IntegerLit(1),
5                 IntegerLit(1))])))]))
6     expect = ""2""
7     self.assertTrue(TestCodeGen.test(input, expect, 521))

```

Sau khi chạy kiểm thử, kết quả chương trình đã đúng tất cả các testcase.



```

Tests run 40
Errors []
[]
Test output
.....
-----
Ran 40 tests in 31.365s

OK

```

Hình 2: Kết quả khi chạy kiểm thử



2.4 Nhận xét

- Với ý tưởng hiện thực trên, ta hoàn toàn có thể dễ dàng mở rộng ra đối với các phép tính khác bằng cách thay các phương thức *emitADDOP* thành *emitMULOP*, *emitDIV*,... và thay phép $+$ thành $*$, $/$,...
- Vẫn còn hạn chế khi chúng ta chỉ cộng được các số trong phạm vi $[0; 32768)$.