Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue & Thu 12:00-13:15 @ ASRC 230, Lab: Fri 10:00-11:15/11:30-12:45 @ FAB 170

Lab 4: Stack IR Code Generation

Learning Objectives Upon successful completion, students will be able to:

- Use syntax-directed translation scheme to implement a Stack IR code generator for a simple AST language representing common expressions and statements.

Preparation

Download and unzip the file `lab4.zip`. You'll see a `lab4` directory with the following contents:

`ast/Ast0.java, ast/<other>.java` — the source AST representation and its parser code

`SC0Gen0.java` — a starter version of the stack IR code-generator

`SC0Interp.jar` — an interpreter for the stack IR language

`tst/` — a set of tests

`Makefile` — for compiling your program

`gen, run` — scripts for testing programs

Overview

You have practiced IR code generation in Lab 2, in which you implemented an IR0 generator for the AST0 language. This lab is similar to Lab 2 in nature. However, there are several differences. (1) The target language is different. We are going to generate stack IR code this time. (2) The target language's representation is different. In Lab 2, IR0 is represented by a collection of Java classes. For this lab, due to the simplicity of SC0's instructions, there is no need to use structured representation. The instructions are simply represented by Strings. (Hence a SC0 program is just a list of Strings.) (3) In Lab 2, you were given code-gen templates (*i.e.* attribute grammars) for all AST nodes. For this lab, you will need to develop some of these templates yourself. In the following, we show several examples of stack IR code-gen.

Arithmetic Binop

```
Exp -> (Binop AOP Exp1 Exp2)
AOP -> "+" |"-" | "*" | "/" | "&&" | "||"
```

Since there is no need to explicitly keep intermediate results, this case is very simple to handle: recursively generate operands' code, then add one extra instruction according to the operator. Here is an example:

```
AST0: (Binop "+" 2 3)      SC0: CONST 2    # operand 1
                                CONST 3    # operand 2
                                ADD        # instruction for Binop
```

The code-gen attribute grammar is

```
Exp.c = Exp1.c      # operand 1
      + Exp2.c      # operand 2
      + "<AOP>"     # instruction for Binop
<AOP> = ADD | SUB | MUL | DIV | AND | OR
```

Relational Binop

```
Exp -> (Binop ROP Exp1 Exp2)
ROP -> "==" | "!=" | "<" | "<=" | ">" | ">="
```

The SC0 language does not have direct instructions for relational operations. For such a binop, the corresponding SC0 code includes two sections, one for the true case and one for the false case; in each case, a 0 or 1 value is produced as the result. The code also includes two jumps. Since there is no labels in the stack language, the jumps are based on relative displacements. Here is an example:

```
AST0: (Binop ">" 2 3)      SC0: CONST 2    # operand 1
                                CONST 3    # operand 2
                                IFGT +3    # compare operands, if true jump to true
                                CONST 0    # false case: result is 0
                                GOTO +2    # jump to end
                                CONST 1    # true case: result is 1
```

Note that regardless of the relational operator, the two jumps' displacements stay the same. With this information, we can develop the following code-gen attribute grammar:

```
Exp.c = Exp1.c          # operand 1
      + Exp2.c          # operand 2
      + "<CJUMP> +3"    # compare operands, if true jump to true
      + "CONST 0"       # false case: result is 0
      + "GOTO +2"       # jump to end
      + "CONST 1"       # true case: result is 1
<CJUMP> = IFEQ | IFNE | IFLT | IFLE | IFGT | IFGE
```

If Statement

```
Stmt -> If Exp Stmt1 [Else Stmt2]
```

This is a more complex case. Let's look at an example:

```
AST0: If (Binop ">" 2 3)  SC0:  CONST 2    # Exp's code
       Assign x 10               CONST 3    # |
       Assign x 20               IFGT +3    # |
                                 CONST 0    # |
                                 GOTO +2    # |
                                 CONST 1    # |
                                 IFZ +4     # if false, jump to false
                                 CONST 10   # Stmt1's code
                                 STORE 0    # |
                                 GOTO +3    # jump to end
                                 CONST 20   # Stmt2's code
                                 STORE 0    # |
```

The SC0 code resembles that of a relational binop: There are true and false branches, and two jumps. However, the jump displacements are not constants in this case. For the first jump (`IFZ`), the displacement is the size of `Stmt1`'s code plus 2; For the second jump (`GOTO`), the displacement is the size of `Stmt2`'s code plus 1. The code generator has to figure out those two sizes before it can generate the jumps. The situation is further complicated by the fact that `Stmt2` is an optional component. If it does not exists, there would be no additional instructions after `Stmt1`'s code, and the first jump's displacement needs to be reduced by 1.

Here is the code-gen attribute grammar for the If statement:

```
Stmt.c = Exp.c
       + "IFZ +n1"      # n1 = Stmt1.c's size +2 if Stmt2 exists
       + Stmt1.c              | Stmt1.c's size +1 otherwise
      [+ "GOTO +n2"]    # n2 = Stmt2.c's size +1
      [+ Stmt2.c]
```

Handling Variables

In stack code, variables are stored in a variable array, and are accessed through their indices. Here is an example:

```
AST0: Assign x 1          SC0: CONST 1    #
      Assign y x               STORE 0    # store x (x's index is 0)
                               LOAD 0     # load x
                               STORE 1    # store y (y's index is 1)
```

In `SC0Gen.java`, the array for storing variable is defined as

```
static ArrayList<String> vars = new ArrayList<String>();
```

When a variable is first assigned a value, the code-gen enters it into the array by calling `vars.add(name)`. After that, to access the variable, the code-gen calls `vars.indexOf(name)` to get its index. (We assume variables in the source program are properly initialized, a variable's uses always come after its definition(s).)

Your Task

1. Follow the above examples, try to develop code-gen attribute grammars for other AST nodes.
2. Use the attribute grammars as guidelines, try to complete the code-gen program, `SC0Gen.java`.

Grammars of AST0 and SC0

The grammars of AST0 and SC0 are included below for your reference.

"AST0 Grammar"

```
Program -> {Stmt}
Stmt    -> "{" {Stmt} "}"
        |  "Assign" Exp Exp
        |  "If" Exp Stmt ["Else" Stmt]
        |  "While" Exp Stmt
        |  "Print" Exp

Exp     -> "(" "Binop" BOP Exp Exp ")"
        |  "(" "Unop" UOP Exp ")"
        |  <Id>
        |  <IntLit>
        |  <BoolLit>

BOP     -> "+" |"-" | "*" | "/" | "&&" | "||"
        |  "==" | "!=" | "<" | "<=" | ">" | ">="
UOP     -> "-" | "!"

<Id>      = [A-Za-z][A-Za-z0-9]*
<IntLit>  = [0-9]+
<BoolLit> = true|false
```

"SC0 Grammar"

```
Program -> {Inst <EOL>}

Inst    -> Inst0               // Inst with no explicit operand
         | Inst1 <IntLit>      // Inst with one operand

Inst0   -> "ADD" | "SUB" | "MUL" | "DIV" | "AND" | "OR" | "SWAP"
         | "NEG" | "PRINT"

Inst1   -> "CONST" | "LOAD" | "STORE" | "GOTO" | "IFZ" | "IFNZ"
         | "IFEQ" | "IFNE" | "IFLT" | "IFLE" | "IFGT" | "IFGE"
```

SC0's instructions are shown below.

Instruction	Semantics	Stack Top (before *vs* after)
`CONST n`	load constant `n` to stack	→ `n`
`LOAD n`	load `var[n]` to stack	→ `val`
`STORE n`	store `val` to `var[n]`	`val` →
`ADD`	`val1 + val2`	`val1,val2` → `result`
`SUB`	`val1 - val2`	`val1,val2` → `result`
`MUL`	`val1 * val2`	`val1,val2` → `result`
`DIV`	`val1 / val2`	`val1,val2` → `result`
`AND`	`val1 & val2`	`val1,val2` → `result`
`OR`	`val1 | val2`	`val1,val2` → `result`
`SWAP`	swap top two stack elements	`val1, val2` → `val2, val1`
`NEG`	`- val`	`val` → `result`
`GOTO n`	`pc = pc + n`	
`IFZ n`	`if (val == 0) pc = pc + n`	`val` →
`IFNZ n`	`if (val != 0) pc = pc + n`	`val` →
`IFEQ n`	`if (val1 == val2) pc = pc + n`	`val1,val2` →
`IFNE n`	`if (val1 != val2) pc = pc + n`	`val1,val2` →
`IFLT n`	`if (val1 < val2) pc = pc + n`	`val1,val2` →
`IFLE n`	`if (val1 <= val2) pc = pc + n`	`val1,val2` →
`IFGT n`	`if (val1 > val2) pc = pc + n`	`val1,val2` →
`IFGE n`	`if (val1 >= val2) pc = pc + n`	`val1,val2` →
`PRINT`	print `val`	`val` →

Note: For the jump instructions, the operand `n` represents the *relative* displacement from the the current instruction position. `n` can be either positive or negative.