Code Generation

Code Generation

- Code generation depends not only on the source language, but also very heavily on the target machine, making it harder to develop general principles.

- First Rule of Code Generation: The resulting object code must be semantically equivalent to the source program.

- Other than I/O errors, errors encountered during code generation represent internal errors and should never occur.
  - We occasionally use Java assertions to make sure that everything is consistent.

Code Generation for CVM

- We will concentrate initially on code generation for the CPRL/0 subset (i.e., no arrays or subprograms).

- Using CVM as the target machine simplifies some aspects of code generation that must be addressed on most "real" machines, such as I/O and the efficient use of general purpose registers.

- Generating assembly language rather than actual machine language also simplifies code generation. For example, the assembler keeps track of the address of each machine instruction, maps labels to machine addresses, and handles the details of branch instructions.

Method `emit()`

- Code generation is performed by the method `emit()` in the AST classes.

- Similar to the implementation of method `checkConstraints()`, most of the AST classes delegate some or all code generation to component classes within the tree.

- Example: `emit()` for class `StatementPart`
  ```
  for (Statement stmt : statements)
      stmt.emit();
  ```

Emitting Object Code

- Class `AST` defines several methods that actually write assembly language to the target file.
  ```
  protected void emitLabel(String label)
  protected void emit(String instruction)
  ...
  ```
- Since all AST classes are subclasses (either directly or indirectly) of class `AST`, then all AST classes inherit these code-generation methods.

- All `emit()` methods involved in code generation must call one or more of these methods, or call another method that calls one or more of these methods, to write out the assembly language during code generation.

Labels

- A label is simply a name for a location in memory. The compiler uses labels for branching, both forward and backward.

- Examples.
  - A loop statement needs to branch backward to the beginning of the loop.
  - An if statement with an else part needs to branch to the else part if the condition is false. If the condition is true, it needs to execute the then statements and then branch over the else part.

- Branches (a.k.a. jumps) are relative. The assembler computes the offset.
  - e. g., BR L5 could translate to branch -12 (backward 12 bytes)

Implementing Labels in the Compiler

- Labels are implemented within the class `AST`.

- Key method:
  ```
  /**
   * Returns a new value for a label number.  This method should
   * be called once for each label before code generation.
   */
  protected String getNewLabel()
  ```

- During code generation, the compiler keeps track of label numbers so that a new label is returned each time the method is called.

- Labels are strings of the form "L1", "L2", "L3", …

©SoftMoore Consulting   Slide 7

Emitting Code for a Loop Statement

- The AST class `LoopStmt` uses two labels
  ```
  private String L1;    // label for start of loop
  private String L2;    // label for end of loop
  ```

- These labels are initialized within the constructor
  ```
  L1 = getNewLabel();
  L2 = getNewLabel();
  ```

- The actual value assigned to the labels by calls to `getNewLabel()` does not matter. What matters is that the values are unique and can be used as targets for branches.

> Note: L1 and L2 are the local names for the labels. The actual string values of L1 and L2 could be different; e.g., "L12" and "L13".

©SoftMoore Consulting   Slide 8

CVM Branch Instructions

- CVM has seven branch instructions
  - `BR`   unconditional branch
  - `BNZ` branch if nonzero (branch if true)
  - `BZ`   branch if zero (branch if false)
  - `BG`   branch if greater
  - `BGE` branch if greater or equal
  - `BL`   branch if less
  - `BLE` branch if less or equal

- Together with the `CMP` (compare) instruction, these branch instructions are used to implement control flow logic within a program or subprogram.

©SoftMoore Consulting   Slide 9

Emitting Code for an Unconditional Branch

- An unconditional branch in CVM has the form
  ```
  BR Ln
  ```
  where L*n* is the label of the instruction that is the target of the branch.

- The assembler converts
  ```
  BR Ln
  ```
  to a branch to the relative offset of the target instruction.

- Emitting an unconditional branch:
  ```
  emit("BR " + L2);
  ```

©SoftMoore Consulting   Slide 10

Emitting Code for Branch Instructions Based on Boolean Values

- In many situations, the code generated for a Boolean expression is followed immediately by a branch instruction.

- Consider as one example a relational expression used as part of a while condition in a loop.
  ```
  while x <= y loop ...
  ```
  In this case, we want to generate code similar to the following:
  ```
  ... // emit code to leave the values of
      // x and y on the top of the stack
  CMP
  BG L1
  ...
  ```
  > Assume that L1 is a label for the instruction following the loop.

©SoftMoore Consulting   Slide 11

Emitting Code for Branch Instructions Based on Boolean Values (continued)

- Consider as a second example the same relational expression used as part of an exit-when statement.
  ```
  exit when x <= y;
  ```
  In this case, we want to generate code similar to the following:
  ```
  ... // emit code to leave the values of
      // x and y on the top of the stack
  CMP
  BLE L1
  ```
  > Assume that L1 is a label for the instruction following the loop.

- Note that in the first example we wanted to generate a branch if the relational expression was false, and in this example we wanted to generate a branch if the relational expression was true.

©SoftMoore Consulting   Slide 12

Emitting Code for Branch Instructions
Based on Boolean Values (continued)

- In addition to the standard `emit()` method, which leaves the value of an expression on the top of the stack, we introduce a method `emitBranch()` for expressions that emits code to produce a value on the stack plus code that branches based on that value.

  ```
  public void emitBranch(boolean condition, String label)
      throws CodeGenException, IOException
  ```

- As pointed out in the previous examples, sometimes we want to emit code to branch if the expression evaluates to true, and sometimes we want to emit code to branch if the expression evaluates to false. The boolean parameter `condition` in method `emitBranch()` specifies which option we want to use.

©SoftMoore Consulting   Slide 13

Emitting Code for Branch Instructions
Based on Boolean Values (continued)

- The `emitBranch()` method is defined in class `Expression` and overridden in class `RelationalExpression`.
  - The default implementation in class `Expression` works correctly for Boolean constants, Boolean named values, and "not" expressions.

©SoftMoore Consulting   Slide 14

Example: `emitBranch()` for
Relational Expressions

```
public void emitBranch(boolean condition, String label)
    throws CodeGenException, IOException
  {
    Token operator = getOperator();

    emitOperands();
    emit("CMP");

    Symbol operatorSym = operator.getSymbol();

    if (operatorSym == Symbol.equals)
        emit(condition ? "BZ "  + label : "BNZ " + label);
    else if (operatorSym == Symbol.notEqual)
        emit(condition ? "BNZ " + label : "BZ "  +  label);
```

(continued on next slide)

©SoftMoore Consulting   Slide 15

Example: `emitBranch()` for
Relational Expressions (continued)

```
    else if (operatorSym == Symbol.lessThan)
        emit(condition ? "BL "  + label : "BGE " + label);
    else if (operatorSym == Symbol.lessOrEqual)
        emit(condition ? "BLE " + label : "BG "  + label);
    else if (operatorSym == Symbol.greaterThan)
        emit(condition ? "BG "  + label : "BLE " + label);
    else if (operatorSym == Symbol.greaterOrEqual)
        emit(condition ? "BGE " + label : "BL "  + label);
    else
        throw new CodeGenException(operator.getPosition(),
                "Invalid relational operator.");
}
```

©SoftMoore Consulting   Slide 16

Helper Methods for Emitting
Load and Store Instructions

Class `AST` provides two helper methods for emitting load and store instructions for various types.

```
/**
 * Emits the appropriate LOAD instruction based on the type.
 */
public void emitLoadInst(Type t) throws IOException


/**
 * Emits the appropriate STORE instruction based on the type.
 */
public void emitStoreInst(Type t) throws IOException
```

©SoftMoore Consulting   Slide 17

Helper Methods for Emitting
Load and Store Instructions (continued)

- Method `emitLoadInst(Type t)` emits the appropriate LOAD instruction based on the size (number of bytes) of a type; e.g.,
  - LOADB (load byte)
  - LOADW (load 4 bytes)
  - LOAD2B (load 2 bytes)
  - LOAD (load n bytes)
- Similarly, method `emitStoreInst(Type t)` emits the appropriate STORE instruction based on the size of a type; e.g.,
  - STOREB (store byte)
  - STOREW (store 4 bytes)
  - STORE2B (store 2 bytes)
  - STORE (store n bytes)

  All load and store instructions retrieve (pop) the target address from the top of the stack.

©SoftMoore Consulting   Slide 18

Method emitLoadInst()

```
protected void emitLoadInst(Type t) throws IOException
  {
    int numBytes = t.getSize();

    if (numBytes == Constants.BYTES_PER_WORD)
        emit("LOADW");
    else if (numBytes == 2)
        emit("LOAD2B");
    else if (numBytes == 1)
        emit("LOADB");
    else
        emit("LOAD " + numBytes);
  }
```

Computing Relative Addresses

- Since all addressing is performed relative to a register, we will need to compute the relative address (offset) for each variable plus the total number of bytes of all variables.
- Method setRelativeAddresses() in the AST class Program computes these values by looping over all single variable declarations.

Computing Relative Addresses
(continued)

```
private void setRelativeAddresses()
  {
    // initial relative address is 0 for a program
    int currentAddr = 0;

    if (declPart != null)
      {
        for (InitialDecl decl : declPart.getInitialDecls())
          {
            // set relative address for single variable declarations
            if (decl instanceof SingleVarDecl)
              {
                SingleVarDecl singleVarDecl = (SingleVarDecl) decl;
                singleVarDecl.setRelAddr(currentAddr);
                currentAddr = currentAddr + singleVarDecl.getSize();
              }
          }
      }                    (continued on next slide)
```

Computing Relative Addresses
(continued)

```
    // compute length of all variables
    varLength = currentAddr;
  }
```

Code Generation for Variables

- For variables (e.g., on the left side of an assignment statement), code generation must leave the address of the variable on the top of the stack.
- The CVM instruction LDGADDR (load global address) will push the (global) address for a variable onto the top of the stack. For CPRL/0, all variables can use this instruction since they all have PROGRAM scope.
- Method emit() for class Variable (for CPRL/0)
  ```
  public void emit() throws IOException
    {
      emit("LDGADDR " + decl.getRelAddr());
    }
  ```

Code Generation for Variables
(continued)

- For full CPRL, we will need to modify emit() for class Variable to correctly handle
  - parameters
  - variables declared at SUBPROGRAM scope level
  - index expressions for array variables

Code Generation for Expressions

- For expressions, code generation must leave the value of the expression on the top of the stack.
- The size (number of bytes) of the value will depend on the type of the variable.
  - 1 byte for a boolean
  - 2 bytes for a character
  - 4 bytes for an integer
  - several bytes for a string literal
    - 4 for the length of the string
    - 2 for each character plus

Code Generation for `ConstValue`

- An object of class `ConstValue` is either a literal or a declared `const` identifier.
- Class `ConstValue` has a method `getLiteralIntValue()` that returns the value of the constant as an integer.
- We can use this method together with the appropriate "load constant" instruction to generate code for the value of the constant.

Method `emit()` for Class `ConstValue`

```
@Override
public void emit() throws CodeGenException, IOException
  {
    Type exprType = getType();

    if (exprType == Type.Integer)
        emit("LDCINT " + getLiteralIntValue());
    else if (exprType == Type.Boolean)
        emit("LDCB " + getLiteralIntValue());
    else if (exprType == Type.Char)
        emit("LDCCH " + literal.getText());
    else if (exprType == Type.String)
        emit("LDCSTR " + literal.getText());
    else
        ...  // throw a CodeGenException
  }
```

Named Values

- A named value is similar to a variable except that it generates different code.
- For example, consider the assignment statement
  ```
  x := y;
  ```
  The identifier "x" represents a variable, and the identifier "y" represents a named value.
- Class `NamedValue` is defined as a subclass of `Variable`.

Code Generation for `NamedValue`

- Code generation for `NamedValue`
  - Calls `emit()` for its superclass `Variable`, which leaves the address of the variable on the top of the stack
  - Calls `emitLoadInst()`, which pops the address off the stack and then pushes the appropriate number of bytes onto the stack, starting at that memory address
- Method `emit()` for class `NamedValue`
  ```
  public void emit() throws CodeGenException, IOException
    {
      super.emit();    // leaves address on top of stack
      emitLoadInst(getType());
    }
  ```

Code Generation for Binary Expressions

- A binary expression contains an operator and two operands, each of which is an expression.
- Code generation for a binary expression usually follows the following pattern:
  - emit code for the left operand
  - emit code for the right operand
  - emit code to perform the operation
- Note that we are generating code that will evaluate the expression using a "postfix" (a.k.a. "reverse polish") notation approach.

Method emit() for Class AddingExpr

```
public void emit() throws CodeGenException, IOException
  {
    Expression leftOperand  = getLeftOperand();
    Expression rightOperand = getRightOperand();
    Symbol     operatorSym  = getOperator().getSymbol();

    leftOperand.emit();
    rightOperand.emit();

    ... // assert that the operator is plus or minus

    if (operatorSym == Symbol.plus)
        emit("ADD");
    else if (operatorSym == Symbol.minus)
        emit("SUB");
  }
```

Slide 31

Short Circuit Evaluation of Logical Expressions

- Given an expression of the form $expr_1$ and $expr_2$
  – The left operand ($expr_1$) is evaluated.
  – If $expr_1$ is false, then $expr_2$ **is not** evaluated and the truth value for the compound expression is considered to be false.
  – If $expr_1$ is true, then $expr_2$ **is** evaluated, and its value becomes the truth value for the compound expression.
- Given an expression of the form $expr_1$ or $expr_2$
  – The left operand ($expr_1$) is evaluated.
  – If $expr_1$ is true, then $expr_2$ **is not** evaluated and the truth value for the compound expression is considered to be true.
  – If $expr_1$ is false, then $expr_2$ **is** evaluated, and value becomes the truth value for the compound expression.

Slide 32

Generating Code for Logical Expressions

- In general, code generation needs to consider whether or not the language requires logical expressions to use short-circuit evaluation (a.k.a., early exit). Similar to most high-level languages, CPRL has such a requirement.
- Using a code generation approach similar that for AddingExpr will **not** result in short-circuit evaluation. For example, in generating code for an "and" expression, we can't simply emit code for left operand, emit code for the right operand, and then "and" them together.

Slide 33

CPRL Code Template for Logical and (with Short-Circuit Evaluation)

```
... // emit code for the left operand
    // (leaves boolean result on top of stack)
BNZ L1
LDCB 0       will be converted to LDCB0 by the optimizer
BR  L2
L1:
... // emit code for the right operand
    // (leaves boolean result on top of stack)

L2:
```

Note: When the instruction BNZ L1 is executed, the boolean value on the top of the stack is popped off. The instruction LDCB 0 is needed to restore the expression value 0 (false) to the top of the stack.

Slide 34

Code Generation for Statements

- Code generation for statements can be described by showing several representative examples of code templates or patterns.
- A code generation template
  – specifies some explicit instructions
  – delegates portions of the code generation to nested components
- Code generation templates for control structures will often use labels to designate destination addresses for branches.

Slide 35

Code Generation for AssignmentStmt

General Description
- Emit code for variable on left side of the assignment operator
  – leaves variable's **address** on top of stack
- Emit code for expression on right side of the assignment operator
  – leaves expression **value** on top of stack
- Emit appropriate store instruction based on the expression's type
  – removes value and address, copies value to address
  – example store instructions are STOREB, STORE2B, STOREW, etc.

Slide 36

Code Generation for `AssignmentStmt`
(continued)

- Grammar Rule
  ```
  variable ":=" expression ";" .
  ```
- Code generation template for type Integer
  ```
  ...  // emit code for variable
  ...  // emit code for expression
  STOREW
  ```
- Code generation template for type Boolean
  ```
  ...  // emit code for variable
  ...  // emit code for expression
  STOREB
  ```

Method emit() for
Class `AssignmentStmt`

```
public void emit() throws CodeGenException, IOException
{
  var.emit();
  expr.emit();

  emitStoreInst(expr.getType());
}
```

Code Generation for a List of Statements

- Grammar Rule
  ```
  statements = ( statement )* .
  ```
- Code generation template
  ```
  for each statement in statements
      ...  // emit code for statement
  ```
- Example: method emit() in class `StatementPart`
  ```
  public void emit() throws CodeGenException, IOException
    {
      for (Statement stmt : statements)
          stmt.emit();
    }
  ```

Code Generation for `LoopStmt`

- Grammar Rule
  ```
  loopStmt = ( "while" booleanExpr )?
             "loop" statements "end" "loop" ";" .
  ```
- Code generation template for loop without a `while` prefix:
  ```
  L1:
      ...  statements nested within the loop
           (usually contain an exit statement)
      BR L1
  L2:
  ```

Code Generation for `LoopStmt`
(continued)

- Code generation template for loop with a `while` prefix
  ```
  L1:
      ...  emit code to evaluate while expression
      ...  branch to L2 if value of expression is false
      ...  statements nested within the loop
      BR L1
  L2:
  ```

Method emit() for `LoopStmt`

```
@Override
public void emit() throws CodeGenException, IOException
  {
    // L1:
    emitLabel(L1);

    if (whileExpr != null)
        whileExpr.emitBranch(false, L2);

    for (Statement stmt : statements)
        stmt.emit();

    emit("BR " + L1);

    // L2:
    emitLabel(L2);
  }
```

Code Generation for ReadStmt

- **Grammar Rule**
  ```
  readStmt = "read" variable ";" .
  ```
- **Code generation template for a variable of type Integer**
  ```
  ... // emit code for variable
      // (leaves variable's address on top of stack)
  GETINT
  ```
- **Code generation template for a variable of type Character**
  ```
  ... // emit code for variable
      // (leaves variable's address on top of stack)
  GETCH
  ```
- Both of the above two templates are followed by code to store the value that was read into the variable.

Slide 43

Method emit() for ReadStmt

```
@Override
public void emit() throws CodeGenException, IOException
  {
    var.emit();

    if (var.getType() == Type.Integer)
        emit("GETINT");
    else  // type must be Char
        emit("GETCH");

    emitStoreInst(var.getType());
  }
```

Slide 44

Code Generation for ExitStmt

- **Grammar Rule**
  ```
  exitStmt = "exit" ( "when" booleanExpr )? ";" .
  ```
- The exit statement must obtain the end label number, say L2, from its enclosing loop statement.
- **Code generation template when the exit statement does not have a when boolean expression suffix**
  ```
  BR L2
  ```
- **Code generation template when the exit statement has a when boolean expression suffix**
  ```
  ... // emit code that will branch to L2 if the
      // when boolean expression evaluates to true
  ```

Slide 45

Method emit() for ExitStmt

```
public void emit() throws CodeGenException, IOException
  {
    String exitLabel = loopStmt.getExitLabel();

    if (whenExpr != null)
        whenExpr.emitBranch(true, exitLabel);
    else
        emit("BR " + exitLabel);
  }
```

Slide 46

Code Generation for IfStmt

- **Grammar Rule**
  ```
  ifStmt = "if" booleanExpr "then" statements
          ( "elsif" booleanExpr "then" statements )*
          ( "else" statements )? "end" "if" ";" .
  ```
- **Code generation template for an if statement**
  ```
  ... // emit code that will branch to L1 if
      //    the boolean expression is false
  ... // emit code for then statements
  BR L2
  L1:
  ... // emit code for elsif parts     (may be empty)
  ... // emit code for else statements (may be empty)
  L2:
  ```

Slide 47

Code Generation for IfStmt
(continued)

- **Code generation template for an elsif part**
  (assumes L2 is the label for the end of the if statement)
  ```
  ... // emit code to branch to L1 if the elsif
      //    Boolean expression is false

  ... // emit code for elsif statements
  BR L2
  L1:
  ```
  > Note: Label L1 is local to the elsif part

Slide 48

Disassembler

- An assembler translates from assembly language to machine code.
- A disassembler is a program that translates from machine code (binary file) back to assembly language (text file).
- A disassembler for CVM has been provided.
  (see `edu.citadel.cvm.Disassembler`)

Code Generation Example: Source Code

```
var x : Integer;
const n := 5;

begin

    x := 1;

    while x <= n loop
        x := x + 1;
    end loop;

    writeln "x = ", x;

end.
```

Code Generation Example: Disassembled Machine Code

```
 0: PROGRAM 4          39: LOADW
 5: LDGADDR 0          40: INC      ← assumes optimization
10: LDCINT1 ← assumes  41: STOREW
11: STOREW  optimization 42: BR -30
12: LDGADDR 0          47: LDCSTR  "x = "
17: LOADW              60: PUTSTR
18: LDCINT 5           61: LDGADDR 0
23: CMP                66: LOADW
24: BG 23              67: PUTINT
29: LDGADDR 0          68: PUTEOL
34: LDGADDR 0          69: HALT
```

Without optimization, the `LDCINT1` instruction at memory address 10 would be `LDCINT 1`, and the `INC` instruction at memory address 40 would look like the following:
```
    LDCINT 1
    ADD
```

Code Generation Example: Annotated Disassembled Object Code

```
// reserve 4 bytes for x      39: LOADW
 0: PROGRAM 4                  40: INC
                              41: STOREW
// x := 1;
 5: LDGADDR 0                  // end loop;
10: LDCINT1                    42: BR -30
11: STOREW
                              // writeln "x = ", x
// while x <= n loop           47: LDCSTR  "x = "
12: LDGADDR 0                  60: PUTSTR
17: LOADW                      61: LDGADDR 0
18: LDCINT 5                   66: LOADW
23: CMP                        67: PUTINT
24: BG 23                      68: PUTEOL

// x := x + 1                  // end.
29: LDGADDR 0                  69: HALT
34: LDGADDR 0
```