

# SBVCONC: Construção de Compiladores

## Aula 11: Geração de Código

Bacharelado em Ciência da Computação  
Prof. Dr. David Buzatto

# Geração de Código

- A geração de código depende não somente do código fonte da linguagem, mas também da máquina alvo, tornando difícil desenvolver princípios gerais;
- **Regra Primordial da Geração de Código:** *“O código objeto resultante tem que ser semanticamente equivalente ao programa fonte”*;
- Além de erros de E/S, os erros encontrados durante a geração de código representam sempre erros internos e nunca devem ocorrer;
  - Ocasionalmente usaremos o mecanismo de asserções do Java para ter certeza de que tudo está consistente.

# Geração de Código para a CVM

- Inicialmente nos concentraremos na geração de código para o subconjunto CPRL/0, ou seja, não lidaremos com arrays e subprogramas;
- Usar a CVM como máquina alvo simplificará alguns aspectos da geração de código que, em máquinas reais, precisariam ser lidados, como E/S e o uso eficiente dos registradores de propósito geral;
- Gerar código em *assembly* ao invés de código de máquina diretamente também simplifica a geração de código. Por exemplo, o *assembler* mantém controle do endereço de cada instrução de máquina, mapeia *labels* (rótulos) à endereços da memória e gerencia os detalhes das instruções de desvio.

# Método emit()

- A geração de código é feita pelo método `emit()` nas classes da AST;
- De forma parecida com o método `checkConstraints()`, a maioria das classes da AST delegam toda ou alguma parte da geração de código para seus componentes dentro da árvore;
- **Exemplo:** `emit()` para a classe `StatementPart`

```
for ( Statement stmt : statements ) {  
    stmt.emit();  
}
```

# Emitindo Código Objeto

- A classe AST define diversos métodos que de fato escrevem o código *assembly* no arquivo alvo:

```
protected void emitLabel( String label )  
protected void emit( String instruction )  
...
```

- Visto que todas as classes da AST são subclasses, diretas ou indiretas, da classe AST, todas elas herdam esses métodos de geração de código;
- Todos os métodos `emit()` envolvidos na geração de código precisam chamar um ou vários desses métodos, ou então invocar um método que, por sua vez, chama um o vários desses métodos, de modo a escrever o código *assembly* durante a geração de código.



# Rótulos (*Labels*)

- Um rótulo é um nome dado a localização da memória. O compilador usa esses rótulos para realizar desvios no fluxo de execução, tanto para frente quanto para trás;
- **Exemplos:**
  - Uma instrução de laço precisa desviar para trás até o início do laço;
  - Uma instrução `if` com sua respectiva parte `else` precisa desviar para a parte do `else` se a condição for falsa. Se a condição for verdadeira, ela precisará executar suas instruções e então desviar para depois da parte do `else`;
- Os desvios (*branches*), ou pulos (*jumps*), são relativos. É o *assembler* que computa o deslocamento (*offset*):
  - **Exemplo:** `BR L5` pode ser traduzido para “`branch -12`” (12 bytes para trás).

# Implementando os Rótulos (*Labels*) no Compilador

- O rótulos são implementas dentro da classe AST;
- Método chave:

```
/**  
 * Retorna um novo valor para o número de um rótulo. Esse método deve  
 * ser invocado apenas uma vez para cada rótulo antes da geração de código.  
 */  
protected String getNewLabel()
```

- Durante a geração de código, o compilador mantém o controle dos números dos rótulos de modo que um novo rótulo será retornado toda vez que o método for invocado;
- Os rótulos são strings na forma "L1", "L2", "L3", ...

# Emitindo Código para Uma Instrução de *Loop*

- A classe `LoopStmt` da AST usa dois rótulos:

```
private String L1;    // rótulo para o início do laço  
private String L2;    // rótulo para o fim do laço
```

- Esses rótulos são inicializados dentro do construtor:

```
L1 = getNewLabel();  
L2 = getNewLabel();
```

- O valor que é de fato atribuído aos rótulos ao se invocar `getNewLabel()` não importa. O que importa é que esses valores são únicos e podem ser usados como alvos para os desvios;
- Observação: `L1` e `L2` são nomes locais para os rótulos. Os valores de `L1` e `L2` podem, e normalmente serão, diferentes, por exemplo, `"L12"` e `"L13"`.



# Instruções de Desvio (*Branch*) da CVM

- A CVM possui sete instruções de desvio:

## ***Compare/Branch Opcodes***

**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*

- Em conjunto com a instrução **CMP** (*compare*), essas instruções de desvio são usadas para implementar a lógica do controle de fluxo dentro de um programa ou subprograma.

# Emitindo Código para Um Desvio Incondicional

- Um desvio incondicional na CVM tem a forma:

`BR Ln`

Onde  $L_n$  é o rótulo da instrução que é o alvo do desvio;

- O *assembler* converte `BR Ln` para um desvio com deslocamento relativo à instrução alvo;
- Emitindo um desvio incondicional:

```
emit( "BR " + L2 );
```

# Emitindo Código para Instruções de Desvio Baseando-se em Valores Booleanos

- Em várias situações, o código gerado para uma expressão booleana é seguido imediatamente por uma instrução de desvio;
- Considere como exemplo uma expressão relacional usada como parte da condição de um `while` em um laço:

```
while x <= y loop ...
```

Nesse caso, queremos gerar código similar ao seguinte (assuma que L1 é o rótulo para a instrução que segue o laço):

```
... // emite o código para deixar os  
    // valores de x e y no topo da pilha  
CMP  
BG L1  
...
```

# Emitindo Código para Instruções de Desvio Baseando-se em Valores Booleanos

- Considere um segundo exemplo usando a mesma expressão relacional, só que agora como parte de uma instrução “exit-when”:

```
exit when x <= y;
```

Nesse caso, queremos gerar código similar ao seguinte (assuma que L1 é o rótulo para a instrução que segue o laço):

```
... // emite o código para deixar os  
    // valores de x e y no topo da pilha  
CMP  
BLE L1
```

- Note que no primeiro exemplo, queríamos gerar um desvio se a expressão relacional fosse falsa. Neste exemplo, queremos gerar um desvio se a expressão relacional for verdadeira.

# Emitindo Código para Instruções de Desvio Baseando-se em Valores Booleanos

- Além da versão padrão do método `emit()`, que deixa o valor de uma expressão no topo da pilha, introduzimos o método `emitBranch()` para expressões que emitem código para produzir um valor na pilha, além de código que executa algum desvio baseado naquele valor;

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

- Como apontado nos exemplos anteriores, algumas vezes queremos emitir código para executar um desvio caso a expressão seja avaliada em verdadeiro ou em falso. O parâmetro booleano `condition` no método `emitBranch()` é usado para especificar qual opção queremos usar.



# Emitindo Código para Instruções de Desvio Baseando-se em Valores Booleanos

- O método `emitBranch()` é definido na classe `Expression` e é sobrescrito na classe `RelationalExpression`;
- A implementação padrão na classe `Expression` funciona apropriadamente para constantes e valores nomeados do tipo `Boolean` e para expressões `"not"`.

# emitBranch() para Expressões Relacionais

```
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 );
    } 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.");
    }
}
```

# Métodos Auxiliares para Emissão de Instruções de Carga (*Load*) e Armazenamento (*Store*)

- A classe AST fornece dois métodos auxiliares para emissão de instruções de carga e de armazenamento para vários tipos:

```
/**
 * Emite a instrução LOAD apropriada baseando-se no tipo.
 */
protected void emitLoadInst( Type t ) throws IOException

/**
 * Emite a instrução STORE apropriada baseando-se no tipo.
 */
protected void emitStoreInst( Type t ) throws IOException
```

# Métodos Auxiliares para Emissão de Instruções de Carga (*Load*) e Armazenamento (*Store*)

- O método `emitLoadInst( Type t )` emite a instrução `LOAD` apropriada baseando-se no tamanho de um tipo (quantidade de bytes):

## *Load Opcodes*

**LOADB:** *load byte*  
**LOAD2B:** *load 2 bytes*  
**LOADW:** *load word (4 bytes)*  
**LOAD:** *load n bytes*

- Similarmente, o método `emitStoreInst( Type t )` emite a instrução `STORE` apropriada baseando-se no tamanho de um tipo:

## *Store Opcodes*

**STOREB:** *store byte*  
**STORE2B:** *store 2 bytes*  
**STOREW:** *store word (4 bytes)*  
**STORE:** *store n bytes*

- Todas as instruções de carga e armazenamento obtém (dão um *pop*) no endereço alvo do topo da pilha.

# Método 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 );  
    }  
  
}
```



# Computando Endereços Relativos

- Visto que todo o endereçamento é calculado de forma relativa à um registrador, precisamos computar o endereço relativo (*offset*/deslocamento) para cada variável além da quantidade de bytes de todas as variáveis;
- O método `setRelativeAddresses()` da classe `Program` da AST computa esses valores iterando sobre todas as declarações únicas de variáveis.

# Computando Endereços Relativos

```
private void setRelativeAddresses() {  
  
    // o endereço relativo inicial para um programa é igual a zero  
    int currentAddr = 0;  
  
    if ( declPart != null ) {  
        for ( InitialDecl decl : declPart.getInitialDecls() ) {  
            // configura o endereço relativo para cada declaração única de variável  
            if ( decl instanceof SingleVarDecl ) {  
                SingleVarDecl singleVarDecl = (SingleVarDecl) decl;  
                singleVarDecl.setRelAddr( currentAddr );  
                currentAddr = currentAddr + singleVarDecl.getSize();  
            }  
        }  
    }  
  
    // computa o tamanho/comprimento de todas as variáveis  
    varLength = currentAddr;  
  
}
```

# Geração de Código para Variáveis

- Para as variáveis (lado esquerdo das instruções de atribuição), a geração de código precisa deixar o endereço da variável no topo da pilha;
- A instrução LDGADDR (*load global address*) da CVM empilhará o endereço global de uma variável no topo da pilha. Para a CPRL/0, todas as variáveis podem usar essa instrução, visto que todas elas têm o escopo de programa (PROGRAM);
- Método `emit()` para a classe `Variable` (para CPRL/0):

```
public void emit() throws IOException {  
    emit( "LDGADDR " + decl.getRelAddr() );  
}
```

# Geração de Código para Variáveis

- Para a CPRL completa, precisaremos modificar o método `emit()` da classe `Variable` para lidar corretamente com:
  - Parâmetros;
  - Variáveis declaradas no escopo de subprograma (SUBPROGRAM);
  - Expressões indexadas para variáveis de arrays.

# Geração de Código para Expressões

- Para expressões, a geração de código precisa deixar o valor da expressão no topo da pilha;
- O tamanho (quantidade de bytes) do valor dependerá do tipo da variável:
  - 1 byte para Boolean
  - 2 bytes para Char
  - 4 bytes para Integer
  - Vários bytes para um literal de string:
    - 4 para o comprimento/tamanho da string;
    - 2 para cada caractere.



# Geração de Código para a Classe ConstValue

- Um objeto da classe ConstValue é ou um literal ou um identificador declarado como const;
- A classe ConstValue possui o método `getLiteralIntValue()` que retorna o valor da constante como inteiro;
- Podemos usar esse método em conjunto com a instrução do tipo “*load constant*” apropriada para gerar código para o valor da constante.

# Método emit() para a Classe 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 {
        ... // lança uma CodeGenException
    }

}
```

# Valores Nomeados

- ▶ Um valor nomeado é similar a uma variável, com a exceção que ele gera código diferente;
- ▶ Por exemplo, considere a instrução de atribuição abaixo:

```
x := y;
```

O identificador “x” representa a variável, enquanto o identificador “y” representa o valor nomeado;

- ▶ A classe `NamedValue` é definida como subclasse de `Variable`.

# Geração de Código para a Classe NamedValue

- A geração de código para a classe NamedValue precisa:
  - Invocar emit() da sua superclasse (Variable), deixando o endereço da variável no topo da pilha;
  - Invocar emitLoadInst(), que desempilhará o endereço da pilha e então empilhará a quantidade apropriada de bytes no topo da pilha, iniciando naquele endereço de memória;
- Método emit() para a classe NamedValue:

```
public void emit() throws CodeGenException, IOException {  
    super.emit();    // deixa o endereço no topo da pilha  
    emitLoadInst( getType() );  
}
```

# Geração de Código para Expressões Unárias

- Uma expressão unária contém um operador e um operando, sendo que o operando é uma expressão;
- A geração de código para uma expressão unária segue, usualmente, o seguinte padrão:
  - Emitir código para o operando;
  - Emitir código para executar a operação.



# Geração de Código para Expressões Binárias

- Uma expressão binária contém um operador e dois operandos sendo que, cada um deles, é uma expressão;
- A geração de código para uma expressão binária segue, usualmente, o seguinte padrão:
  - Emitir código para o operando da esquerda;
  - Emitir código para o operando da direita;
  - Emitir código para executar a operação;
- Note que estamos gerando código que avaliará a expressão usando uma abordagem pós-fixada (*reverse polish notation/notação polonesa inversa/reversa*).

# Método emit() para a Classe AddingExpr

```
public void emit() throws CodeGenException, IOException {  
  
    Expression leftOperand = getLeftOperand();  
    Expression rightOperand = getRightOperand();  
    Symbol operatorSym = getOperator().getSymbol();  
  
    leftOperand.emit();  
    rightOperand.emit();  
  
    if ( operatorSym == Symbol.plus ) {  
        emit( "ADD" );  
    } else if ( operatorSym == Symbol.minus ) {  
        emit( "SUB" );  
    }  
  
}
```

# Avaliação de Curto-Circuito de Expressões Lógicas

- Dada uma expressão na forma  $\text{expr}_1$  **and**  $\text{expr}_2$ 
  - O operando da esquerda ( $\text{expr}_1$ ) é avaliado;
  - Se  $\text{expr}_1$  for falsa, então  $\text{expr}_2$  **não é** avaliada e o valor verdade para a expressão composta é considerado como falso;
  - Se  $\text{expr}_1$  for verdadeira, então  $\text{expr}_2$  **é** avaliada e seu valor se torna o valor verdade para a expressão composta;
- Dada uma expressão na forma  $\text{expr}_1$  **or**  $\text{expr}_2$ 
  - O operando da esquerda ( $\text{expr}_1$ ) é avaliado;
  - Se  $\text{expr}_1$  for verdadeira, então  $\text{expr}_2$  **não é** avaliada e o valor verdade para a expressão composta é considerado como verdadeiro;
  - Se  $\text{expr}_1$  for falsa, então  $\text{expr}_2$  **é** avaliada e seu valor se torna o valor verdade para a expressão composta.

# Geração de Código para Expressões Lógicas

- Em geral, a geração de código precisa considerar se a linguagem requer ou não que as expressões lógicas usem a avaliação em curto-circuito (também chamada de saída antecipada/*early exit*). De forma similar à maioria das linguagens de alto nível, a CPRL possui tal requisito;
- Ao se usar para a geração de código uma abordagem similar à usada na classe `AddngExpr`, o resultado da avaliação em curto-circuito **não será alcançado**. Por exemplo, ao gerar o código para uma expressão “and”, não poderemos simplesmente emitir o código para o operando da esquerda, depois para o operando da direita e, por fim, aplicar o operador “and” nos resultados obtidos.

# Modelo (*Template*) de Código da CPRL para and (e) Lógico com Avaliação de Curto-Circuito

```
... // emite código para o operando da esquerda
    // (deixa o resultado booleano no topo da pilha)
BNZ L1
LDCB 0 // será convertido para LDCB0 pelo otimizador
BR L2

L1:
... // emite código para o operador da direita
    // (deixa o resultado booleano no topo da pilha)

L2:
```

- Note que quando a instrução BNZ L1 é executada, o valor booleano no topo da pilha é desempilhado. A instrução LDCB 0 é necessária para restaurar o resultado 0 (falso) da expressão ao topo da pilha.



# Geração de Código para Instruções

- A geração de código para as instruções pode ser descrita ao se mostrar diversos exemplos representativos de modelos ou padrões de código;
- Modelo de geração de código:
  - Especifica algumas instruções explícitas;
  - Delega porções da geração de código aos componentes aninhados;
- Os modelos para a geração de código das estruturas de controle usarão, na maioria das vezes, rótulos para designar os endereços de destino para os desvios.



# Geração de Código para a Classe AssignmentStmt

## ➤ Descrição geral:

- Emitir código para a variável do lado esquerdo do operador de atribuição:
  - Deixa o **endereço** da variável no topo da pilha;
- Emitir código para a expressão do lado direito do operador de atribuição:
  - Deixa o **valor** da expressão no topo da pilha;
- Emitir a instrução de armazenamento apropriada baseada no tipo da expressão:
  - Remove o valor e o endereço, copia o valor para o endereço;
  - Os exemplos de instrução de armazenamento são STOREB, STORE2B, STOREW etc.

# Geração de Código para a Classe AssignmentStmt

- Regra gramatical:

```
variable "[:=" expression "];" .
```

- Modelo de geração de código para o tipo Integer:

```
... // emite código para a variável  
... // emite código para a expressão  
STOREW
```

- Modelo de geração de código para o tipo Boolean:

```
... // emite código para a variável  
... // emite código para a expressão  
STOREB
```

# Método emit() para a Classe AssignmentStmt

```
public void emit() throws CodeGenException, IOException {  
  
    variable.emit();  
    expr.emit();  
    emitStoreInst( expr.getType() );  
  
}
```

# Geração de Código para uma Lista de Instruções

- Regra gramatical:

```
statements = ( statement )* .
```

- Modelo de geração de código:

```
for each statement in statements  
    ... // emite código para a instrução
```

- **Exemplo:** método emit() para a classe StatementPart:

```
public void emit() throws CodeGenException, IOException {  
    for ( Statement stmt : statements ) {  
        stmt.emit();  
    }  
}
```

# Geração de Código para a Classe LoopStmt

- Regra gramatical:

```
loopStmt = ( "while" booleanExpr )?  
           "loop" statements "end" "loop" ";" .
```

- Modelo de geração de código para um laço sem o prefixo while:

```
L1:  
    ... // instruções aninhadas dentro do laço  
        // (normalmente contém uma instrução exit)  
    BR L1  
L2:
```

- Modelo de geração de código para um laço com o prefixo while:

```
L1:  
    ... // emite código para avaliar a expressão while  
    ... // desvia para L2 se o valor da expressão for falso  
    ... // instruções aninhadas dentro do laço  
    BR L1  
L2:
```

# Método emit() para a Classe 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 );

}
```



# Geração de Código para a Classe ReadStmt

- Regra gramatical:

```
readStmt = "read" variable ";" .
```

- Modelo de geração de código para uma variável do tipo Integer:

```
... // emite código para a variável  
    // (deixa o endereço da variável no topo da pilha)  
GETINT
```

- Modelo de geração de código para uma variável do tipo Char:

```
... // emite código para a variável  
    // (deixa o endereço da variável no topo da pilha)  
GETCH
```

- Os dois modelos acima são seguidos pelo código que corresponde ao armazenamento do valor que foi lido e inserido na variável.

# Método emit() para a Classe ReadStmt

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

    variable.emit();

    if ( variable.getType() == Type.Integer ) {
        emit( "GETINT" );
    } else { // o tipo tem que ser Char
        emit( "GETCH" );
    }

    emitStoreInst( variable.getType() );

}
```

# Geração de Código para a Classe ExitStmt

- Regra gramatical:

```
exitStmt = "exit" ( "when" booleanExpr )? ";" .
```

- A instrução `exit` tem que obter o número do rótulo do fim (L2) da estrutura de laço que a engloba;
- Modelo de geração de código quando a instrução `exit` não tem um sufixo `when` para a expressão booleana:

```
BR L2
```

- Modelo de geração de código quando a instrução `exit` tem um sufixo `when` para a expressão booleana:

```
... // emite código que desviará à L2 se a  
    // expressão booleana de when for avaliada como verdadeiro
```

# Método emit() para a Classe ExitStmt

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

    String exitLabel = loopStmt.getExitLabel();

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

}
```

# Geração de Código para a Classe IfStmt

## ➤ Regra gramatical:

```
ifStmt = "if" booleanExpr "then" statements  
        ( "elseif" booleanExpr "then" statements ) *  
        ( "else" statements ) ? "end" "if" ";" .
```

## ➤ Modelo de geração de código para uma instrução if:

```
... // emite código que desviará à L1 se  
    // a expressão booleana for falsa  
... // emite código para as instruções do "then"  
BR L2  
L1:  
... // emite código para a parte elseif (pode ser vazia)  
... // emite código para as instruções do else (pode ser vazia)  
L2:
```

# Geração de Código para a Classe IfStmt

- Modelo de geração de código para a parte `elsif`, assumindo que que `L2` é o rótulo do fim da instrução `if`:

```
... // emite código que desviará à L1 se a expressão  
    // booleana do elsif for falsa
```

```
... // emite código para as instruções do elsif  
BR L2
```

L1:

- Note que o rótulo `L1` é local à parte `elsif`.



# Disassembler (Desmontador)

- Um *assembler* (montador) traduz da linguagem de montagem (*assembly*) para código de máquina;
- Um *disassembler* (desmontador) é um programa que traduz da linguagem de máquina (arquivo binário) de volta à linguagem *assembly* (arquivo de texto);
- O *disassembler* da CVM é fornecido na classe `edu.citadel.cvm.Disassembler`

# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string

# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string

# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string

# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** *Load constant integer*  
**LDCINT1:** *Load constant integer 1*  
**LDCSTR:** *Load constant string*  
**LDGADDR:** *Load global address*  
**LOADW:** *Load word*  
**STOREW:** *store word*

### Arithmetic Opcodes

**INC:** *increment*

### Compare/Branch Opcodes

**CMP:** *compare*  
**BG:** *branch if greater*  
**BR:** *branch*

### Program Opcodes

**HALT:** *halt*  
**PROGRAM:** *program*

### I/O Opcodes

**PUTEOL:** *put end-of-line*  
**PUTINT:** *put integer*  
**PUTSTR:** *put string*



# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin
    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string



# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string

# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string

# Exemplo de Geração de Código

```

var x : Integer;
const n := 5;

begin

    x := 1;

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

    writeln "x = ", x;

end.

```

## Exemplo Desmontado

```

0: PROGRAM 4
5: LDGADDR 0
10: LDCINT1
11: STOREW
12: LDGADDR 0
17: LOADW
18: LDCINT 5
23: CMP
24: BG 23
29: LDGADDR 0
34: LDGADDR 0

```

## Otimizado!

```

39: LOADW
40: INC
41: STOREW
42: BR -30
47: LDCSTR "x = "
60: PUTSTR
61: LDGADDR 0
66: LOADW
67: PUTINT
68: PUTEOL
69: HALT

```

Sem otimização, a instrução **LDCINT1** no endereço de memória **10** deveria ser **LDCINT 1** e a instrução **INC** no endereço de memória **40** deveria parecer com algo como:

```

LDCINT 1
ADD

```

### Load/Store Opcodes

**LDCINT:** Load constant integer  
**LDCINT1:** Load constant integer 1  
**LDCSTR:** Load constant string  
**LDGADDR:** Load global address  
**LOADW:** Load word  
**STOREW:** store word

### Arithmetic Opcodes

**INC:** increment

### Compare/Branch Opcodes

**CMP:** compare  
**BG:** branch if greater  
**BR:** branch

### Program Opcodes

**HALT:** halt  
**PROGRAM:** program

### I/O Opcodes

**PUTEOL:** put end-of-line  
**PUTINT:** put integer  
**PUTSTR:** put string

## Projeto 6: Implementação da Geração de Código do Compilador da CPRL

- Implemente todos os métodos `emit()` das classes da AST;
- Verifique no projeto o arquivo “Visão Geral das Classes do Projeto.txt” em que as classes que precisam ser modificadas estão listadas;
- As implementações que devem ser feitas estão explicadas em comentários dentro dos métodos `emit()` que precisam ser implementados;
- Há várias classes com a implementação pronta e que podem ser usadas como base para a implementação das que precisam ser complementadas.

# Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.