Implementação do gerador de código

```
public class Instruction {
   public byte  op;              // op-code (0 .. 15)
   public byte  r;               // register field (0 .. 15)
   public byte  n;               // length field (0 .. 255)
   public short d;               // operand field (–32767 .. +32767)

   public static final byte   // op-codes (Table C.2)
         LOADop   =  0,  LOADAop  =  1,
         LOADIop  =  2,  LOADLop  =  3,
         STOREop  =  4,  STOREIop =  5,

         CALLop   =  6,  CALLIop  =  7,
         RETURNop =  8,
         PUSHop   = 10,  POPop    = 11,
         JUMPop   = 12,  JUMPIop  = 13,
         JUMPIFop = 14,  HALTop   = 15;

   public static final byte   // register numbers (Table C.1)
         CBr =  0,  CTr =  1,  PBr =  2,  PTr =  3,
         SBr =  4,  STr =  5,  HBr =  6,  HTr =  7,
         LBr =  8,  L1r =  9,  L2r = 10,  L3r = 11,
         L4r = 12,  L5r = 13,  L6r = 14,  CPr = 15;

   public Instruction (byte op, byte r, byte n,
                       short d)
   { ... }
}
```

```
private Instruction[] code = new Instruction[1024];
private short nextInstrAddr = 0;    // address of next instruction
                                    // to be stored in code
```

```
private void emit (byte op, byte n, byte r, short d) {
    // Append an instruction with fields op, n, r, d to the object program.
    code[nextInstrAddr++] =
            new Instruction(op, n, r, d);
}
```

Phrase class	Visitor/encoding method	Behavior of visitor/encoding method
Program	`visitProgram`	Generate code as specified by '*run P*'.
Command	`visit...Command`	Generate code as specified by '*execute C*'.
Expression	`visit...Expression`	Generate code as specified by '*evaluate E*'.
V-name	`visit...Vname`	Return an entity description for the given value-or-variable-name (explained in Section 7.3.)
Declaration	`visit...Declaration`	Generate code as specified by '*elaborate D*'.
Type-denoter	`visit...TypeDenoter`	Return the size of the given type.

```
public Object visitProgram (Program prog, Object arg);
    // Generate code as specified by 'run prog'.

public Object visit...Command
                (...Command com, Object arg);
    // Generate code as specified by 'execute com'.

public Object visit...Expression
                (...Expression expr, Object arg);
    // Generate code as specified by 'evaluate expr'.

public Object visit...Declaration
                (...Declaration decl, Object arg);
    // Generate code as specified by 'elaborate decl'.
```

```
private void encodeFetch (Vname vname);
    // Generate code as specified by 'fetch vname'.

private void encodeAssign (Vname vname);
    // Generate code as specified by 'assign vname'.
```

```
public Object visitProgram                    run ⟦C⟧ =
             (Program prog,
              Object arg) {
   prog.C.visit(this, arg);                      execute C
   emit(Instruction.HALTop, 0, 0, 0);            HALT
}
```

```
public Object visitAssignCommand                    execute [[V := E]] =
                (AssignCommand com,
                 Object arg) {
    com.E.visit(this, arg);                          evaluate E
    encodeAssign(com.V);                             assign V
    return null;
}
public Object visitCallCommand                      execute [[I ( E )]] =
                (CallCommand com,
                 Object arg) {
    com.E.visit(this, arg);                          evaluate E
    short p = address of primitive routine
                named com.I;
    emit(Instruction.CALLop,                         CALL p
         Instruction.SBr,
         Instruction.PBr, p);
    return null;
}
```

```
public Object visitSequentialCommand          execute ⟦C1 ; C2⟧ =
                (SequentialCommand com,
                 Object arg) {
    com.C1.visit(this, arg);                    execute C1
    com.C2.visit(this, arg);                    execute C2
    return null;
}
public Object visitLetCommand                 execute ⟦let D
                (LetCommand com,                          in C⟧ =
                 Object arg) {
    com.D.visit(this, arg);                     elaborate D
    com.C.visit(this, arg);                     execute C
    short s = amount of storage allocated by D;
    if (s > 0)                                  if s > 0
        emit(Instruction.POPop, 0, 0, s);         POP(0) s
    return null;
}
```

```
public Object visitIntegerExpression
                (IntegerExpression expr,
                 Object arg) {
    short v = valuation(expr.IL.spelling);
    emit(Instruction.LOADLop, 0, 0, v);
    return null;
}
```

evaluate $[\![IL]\!]$ =

LOADL *v*

```
public Object visitVnameExpression
                (VnameExpression expr,
                 Object arg) {
    encodeFetch(expr.V);
    return null;
}
```

evaluate $[\![V]\!]$ =

fetch V

```
public Object visitUnaryExpression          evaluate ⟦O E⟧ =
                (UnaryExpression expr,
                 Object arg) {
    expr.E.visit(this, arg);                    evaluate E
    short p = address of primitive routine
                named expr.O;
    emit(Instruction.CALLop,                    CALL p
         Instruction.SBr,
         Instruction.PBr, p);
    return null;
}

public Object visitBinaryExpression         evaluate ⟦E1 O
                (BinaryExpression expr,                E2⟧ =
                 Object arg) {
    expr.E1.visit(this, arg);                   evaluate E1
    expr.E2.visit(this, arg);                   evaluate E2
    short p = address of primitive routine
                named expr.O;
    emit(Instruction.CALLop,                    CALL p
         Instruction.SBr,
         Instruction.PBr, p);
    return null;
}
```

```
public final class Encoder implements Visitor {
    ... // Auxiliary methods, as above.
    ... // Visitor/encoding methods, as above.

    public void encode (Program prog) {
        prog.visit(this, null);
    }
}
```

For instance, in Example 7.3 we saw the translation of '`while i > 0 do i := i - 2`'. Here we show in detail how `visitWhileCommand` generates this object code:

(1) It saves the next instruction address (say 30) in `j`.

(2) It generates a `JUMP` instruction with a zero address field:

```
30: JUMP  0
```

(3) It saves the next instruction address (namely 31) in `g`.

(4) It translates the subcommand '`i := i - 2`' to object code:

```
31: LOAD  i
32: LOADL 2
33: CALL  sub
34: STORE i
```

(5) It takes the next instruction address (namely 35), and patches it into the address field of the instruction whose address was saved in `j` (namely 30):

```
30: JUMP  35
```

(6) It translates the expression '`i > 0`' to object code:

```
35: LOAD  i
36: LOADL 0
37: CALL  gt
```

(7) It generates a `JUMPIF` instruction whose address field contains the address that was saved in `g` (namely 31):

```
38: JUMPIF(1) 31
```

```
public Object visitWhileCommand (
                (WhileCommand com,
                 Object arg) {
   short j = nextInstrAddr;
   emit(Instruction.JUMPop, 0,
        Instruction.CBr, 0);

   short g = nextInstrAddr;
   com.C.visit(this, arg);
   short h = nextInstrAddr;
   patch(j, h);
   com.E.visit(this, arg);
   emit(Instruction.JUMPIFop, 1,
        Instruction.CBr, g);
   return null;
}
```

execute ⟦while *E* do *C*⟧ =

j:
JUMP *h*

g:
execute C
h:

evaluate E
JUMPIF(1) *g*

```
public Object visitIfCommand
                (IfCommand com,
                 Object arg) {
    com.E.visit(this, arg);
    short i = nextInstrAddr;
    emit(Instruction.JUMPIFop, 0,
          Instruction.CBr, 0);
    com.C1.visit(this, arg);
    short j = nextInstrAddr;
    emit(Instruction.JUMPop, 0,
          Instruction.CBr, 0);
    short g = nextInstrAddr;
    patch(i, g);
    com.C2.visit(this, arg);
    short h = nextInstrAddr;
    patch(j, nextInstrAddr);
    return null;
}
```

execute ⟦`if` *E*
`then` C_1
`else` C_2⟧ =

evaluate E
i:
`JUMPIF(0)` *g*

execute C_1
j:
`JUMP` *h*

g:

execute C_2
h:

```
private void patch (short addr, short d) {
    // Store d in the operand field of the instruction at address addr.
    code[addr].d = d;
}
```

Observações

- No projeto, não iremos fazer backpatching;
- Refereciar os endereços de memória através de rótulos simbólicos;
- Desvio do fluxo de controle e chamada de procedimentos e funções.

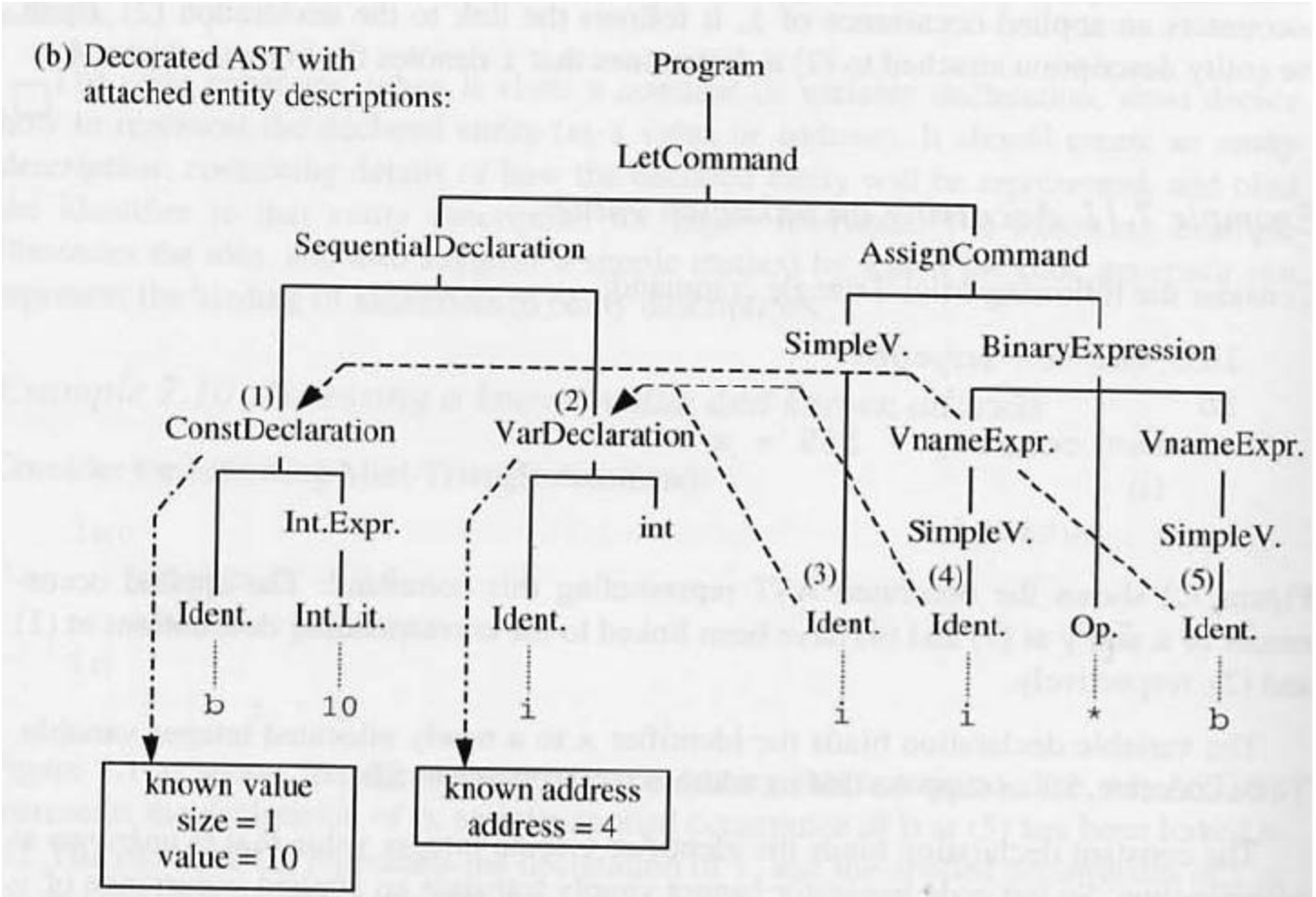

Figure 7.1 Entity descriptions for a known value and a known address.

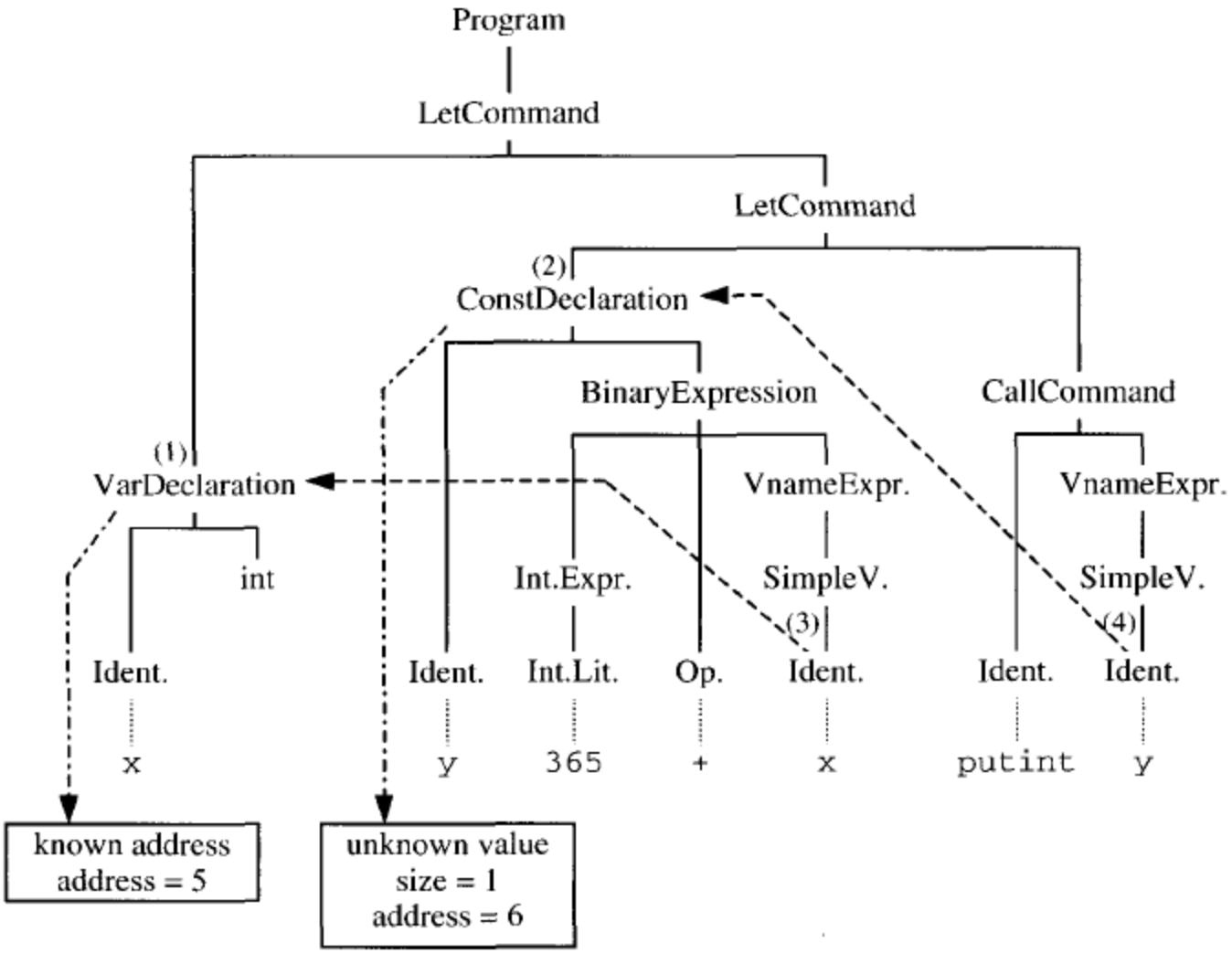

Figure 7.2 Entity descriptions for a known address and an unknown value.

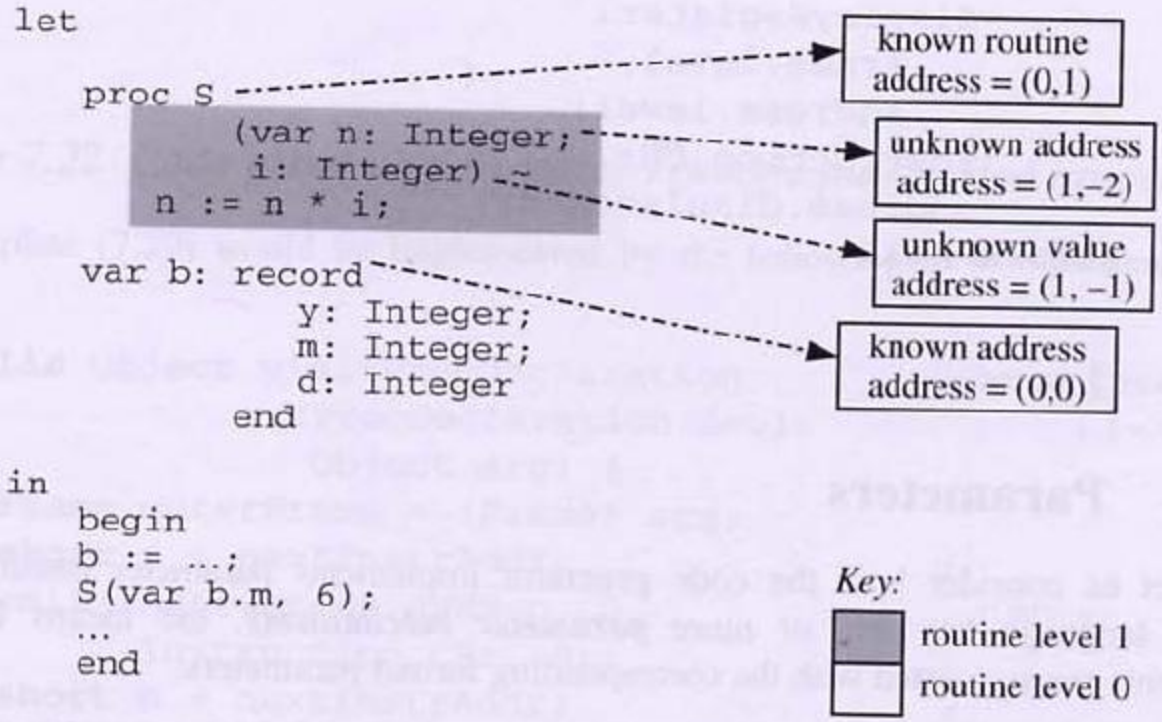

Figure 7.6 Entity descriptions for constant and variable parameters.

In declarations, identifiers may be bound to entities such as *values* and *addresses*. Each entity may be either *known* or *unknown* (at compile-time). All combinations are possible, and all actually occur in some languages:

- ***Known value:*** This describes a value bound in a constant declaration whose right side is a literal.
- ***Unknown value:*** This describes a value bound in a constant declaration whose right side must be evaluated at run-time, or an argument value bound to a constant parameter.
- ***Known address:*** This describes an address allocated and bound in a variable declaration.
- ***Unknown address:*** This describes an argument address bound to a variable parameter.

We can systematically deal with both known and unknown entities by the techniques illustrated in Examples 7.10 and 7.11. In general:

- If an identifier *I* is bound to a *known* entity, the code generator creates an entity description containing that known entity, and attaches that entity description to the declaration of *I*. It translates each applied occurrence of *I* to that known entity.
- If an identifier *I* is bound to an *unknown* entity, the code generator generates code to evaluate the unknown entity and store it at a known address, creates an entity description containing that known address, and attaches that entity description to the declaration of *I*. At each applied occurrence of *I*, the code generator generates code to fetch the unknown entity from the known address.

The auxiliary function *display-register*(*cl*, *l*) selects the display register that will enable code at routine level *cl* to address a variable declared at routine level *l*:

$$\textit{display-register}(cl, l) = \begin{cases} \text{SB} & \text{if } l = 0 \quad (7.20a) \\ \text{LB} & \text{if } l > 0 \text{ and } cl = l \quad (7.20b) \\ \text{L1} & \text{if } l > 0 \text{ and } cl = l + 1 \quad (7.20c) \\ \text{L2} & \text{if } l > 0 \text{ and } cl = l + 2 \quad (7.20d) \\ \ldots & \end{cases}$$

fetch $[\![I]\!]$ = (7.39)

(i) if I is bound to a known value:

`LOADL` v where v = value bound to I

(ii) if I is bound to an unknown value or known address:

`LOAD(`s`)` d`[`r`]` where s = *size*(type of I),
(l, d) = address bound to I,
cl = current routine level,
r = *display-register*(cl, l)

(iii) if I is bound to an unknown address:

`LOAD(1)` d`[`r`]`
`LOADI(`s`)` where s = *size*(type of I),
(l, d) = address bound to I,
cl = current routine level,
r = *display-register*(cl, l)

assign $[\![I]\!]$ = (7.40)

(i) if I is bound to a known address:

`STORE(`s`)` d`[`r`]` where s = *size*(type of I),
(l, d) = address bound to I,
cl = current routine level,
r = *display-register*(cl, l)

(ii) if I is bound to an unknown address:

`LOAD(1)` d`[`r`]`
`STOREI(`s`)` where s = *size*(type of I),
(l, d) = address bound to I,
cl = current routine level,
r = *display-register*(cl, l)

fetch-address $[\![I]\!]$ = (7.41)

(i) if I is bound to a known address:

`LOADA` $d[r]$ where (l, d) = address bound to I,
cl = current routine level,
r = *display-register*(cl, l)

(ii) if I is bound to an unknown address:

`LOAD(1)` $d[r]$ where (l, d) = address bound to I,
cl = current routine level,
r = *display-register*(cl, l)