# IR Code Generation (Part I)

*Input* — AST representation of a source language
*Output* — Three-address code or IR Tree code
*Approach* — Syntax-directed translation

*IR Language Components (generic verbose form):*

- *Expressions*

```
E -> E1 op E2         // arith op
E -> '-' E1
E -> E1 relop E2      // boolean op
E -> E1 logicop E2    //
E -> '!' E1
E -> 'newArray' E1    // new (integer) array of size E1
E -> E1 '[' E2 ']'    // array element
```

- *Statements*

```
S -> E1 ':=' E2 ';'
S -> 'if' '(' E ')' 'then' S1 'else' S2
S -> 'while' '(' E ')' S1
S -> 'print' E ';'
S -> 'return' E ';'
```

# Arithmetic Expressions

- *Three-Address Code:*
  — Introduces a new temp for each operation. (Two attributes: $E.s$ holds the sequence of statements evaluating $E$; and $E.t$ represents the temp that holds the value of $E$.)

```
E -> E1 op E2
```

```
t = new Temp();
E.s := [ E1.s; E_2.s; t := E1.t op E2.t; ]
E.t := t;
```

```
E -> '-' E1
```

```
t = new Temp();
E.s := [ E1.s; t := - E1.t; ]
E.t := t;
```

*Example:* `b * -c + b * d`

```
t1 := -c;
t2 := b * t1
t3 := b * d
t4 := t2 + t3
```

- *IR Trees:*
  — Simply embed the trees for the subexressions inside the outer expression tree. (The single attribute $E.tr$ holds the IR tree generated for $E$.)

```
E -> E1 op E2    E.tr := (BINOP op E1.tr E2.tr)
E -> - E1        E.tr := (UNOP - E1.tr)
```

# Boolean Expressions

Boolean expressions cannot be translated the same way as arithmetic expressions, since relational and logical operations are handled differently than arithmetic operations at lower levels — relational operations trigger conditional flags instead of producing value results; logical operations are realized (only) through control flow transfers.

For instance, `a<5 || b>2;` cannot be simply translated into

```
t1 = a < 5;
t2 = b > 2;
t3 = t1 || t2;
```

since relational operations can only be used in conditional jump statements.

There are two general approaches:

- *Value-Representation:*
  - Encode `true` and `false` numerically into 1 and 0
  - Map Boolean expressions into conditional jump statements
- *Flow-of-Control Representation*:
  - Use positions in generated code to represent Boolean values.

# Value Representation Approach

The value of a Boolean expression is represented by either 1 or 0.

*Example:* `a<5 || b>2`

- *Three-Address Code:*

```
    t1 := 1;
    if (a < 5) goto L1;
    t1 := 0;
L1:
    t2 := 1;
    if (b > 2) goto L2;
    t2 := 0;
L2:
    t3 := 1;
    if (t1 == 1) goto L3;
    if (t2 == 1) goto L3;
    t3 := 0;
L3:
```

- *IR Tree:*

```
(ESEQ [ [MOVE t3 (CONST 1)]
        [CJUMP == (ESEQ [ [MOVE t1 (CONST 1)]
                          [CJUMP < (NAME a) (CONST 5) L1]
                          [MOVE t1 (CONST 0)]
                          [LABEL L1] ] t1)
                  (CONST 1) L3]
        [CJUMP == (ESEQ [ [MOVE t2 (CONST 1)]
                          [CJUMP > (NAME b) (CONST 2) L2]
                          [MOVE t2 (CONST 0)]
                          [LABEL L2] ] t2)
                  (CONST 1) L3]
        [MOVE t3 (CONST 0)]
        [LABEL L3] ] t)
```

# Better Handling for Logical Operations

Many architectures provide hardware support for bit-wise logical operations, such as *and*, *or*, *xor not*, and etc.

And we know that

```
1 and 1 = 1; 1 and 0 = 0; 0 and 1 = 0; 0 and 0 = 0;
1 or  1 = 1; 1 or  0 = 1; 0 or  1 = 1; 0 or  0 = 0;
```

Taking advantage of this, when using value-representation for Boolean expressions, logical operations can be simply translated into arithmetic operations with corresponding bit-wise operators. For instance, the expression `a<5 || b>2` can be translated to

- *Three-Address Code:*

```
      t1 := 1;
      if (a < 5) goto L1;
      t1 := 0;
  L1:
      t2 := 1;
      if (b > 2) goto L2;
      t2 := 0;
  L2:
      t3 := t1 or t2;
```

- *IR Tree:*

```
(BINOP || (ESEQ [ [MOVE t1 (CONST 1)]
                  [CJUMP < (NAME a) (CONST 5) L1]
                  [MOVE t1 (CONST 0)]
                  [LABEL L1] ] t1)
          (ESEQ [ [MOVE t2 (CONST 1)]
                  [CJUMP > (NAME b) (CONST 2) L2]
                  [MOVE t2 (CONST 0)]
                  [LABEL L2] ] t2))
```

# IR Gen — Value Representation

*Three-Address Code:*

```
E -> E1 relop E2
```

```
L = new Label();
t = new Temp();
E.s := [ E1.s; E2.s; t := 1;
         if (E1.t relop E2.t) goto L;
         t := 0; L: ]
E.t := t;
```

```
E -> E1 '||' E2
```

```
L = new Label();
t = new Temp();
E.s := [ E1.s; E2.s; t := 1;
         if (E1.t == 1) goto L;
         if (E2.t == 1) goto L;
         t := 0; L: ]
E.t := t;
```

```
E -> E1 '&&' E2
```

```
L = new Label();
t = new Temp();
E.s := [ E1.s; E2.s; t := 0;
         if (E1.t == 0) goto L;
         if (E2.t == 0) goto L;
         t := 1; L: ]
E.t := t;
```

```
E -> '!'E1
```

```
t = new Temp();
E.s := [ E1.s; t := 1 - E1.t; ]
E.t := t;
```

# IR Gen — Value Representation

*IR Tree:*

`E -> E1 relop E2`

```
L = new NAME();
t = new TEMP();
E.tr := (ESEQ [ [MOVE t (CONST 1)]
                [CJUMP relop E1.tr E2.tr L]
                [MOVE t (CONST 0)]
                [LABEL L] ] t)
```

`E -> E1 '||' E2`

```
L = new NAME();
t = new TEMP();
E.tr := (ESEQ [ [MOVE t (CONST 1)]
                [CJUMP == E1.tr (CONST 1) L]
                [CJUMP == E2.tr (CONST 1) L]
                [MOVE t (CONST 0)]
                [LABEL L] ] t)
```

`E - E1 '&&' E2`

```
L = new NAME();
t = new TEMP();
E.tr := (ESEQ [ [MOVE t (CONST 0)]
                [CJUMP == E1.tr (CONST 0) L]
                [CJUMP == E2.tr (CONST 0) L]
                [MOVE t (CONST 1)]
                [LABEL L] ] t)
```

`E - '!'E1`

```
t = new TEMP();
E.tr := (ESEQ [MOVE t (BINOP - (CONST 1) E1.tr)] t)
```

# Control-Flow Representation

In many cases, Boolean expressions in a source program are used to switch control flow, e.g.

```
if (a<5 || b>2) S1 else S2;
```

For these cases, the values of Boolean expressions are not needed in the end. So a better approach is to avoid using values all together.

So instead of adding instructions after value-representation code for `(a<5 || b>2)`:

```
    [code for (a<5 || b>2)] // result in t3
    if (t3 == 0) goto L5;
L4: [code for S1]  // then clause
    goto L6;
L5: [code for S2]  // else clause
L6:
```

we could generate the following more efficient code

```
    if (a < 5) goto L4;
    if (b <= 2) goto L5;
L4: [code for S1]  // then clause
    goto L6;
L5: [code for S2]  // else clause
L6:
```

In this version, there is no need to create all those temps for holding 0s and 1s.

# Control-Flow Representation (cont.)

One issue needs to be resolved:

> The two labels, `L4` and `L5`, are not available when the Boolean expression `(a<5 || b>2)` is being processed. How can the two conditional jump statements be generated, then?

*Answer: Use the idea of "back-patching":*

> Each block of code may contain jumps to unresolved labels; these labels will be *patched* when the environment of the block is processed.

*Example:* `if (a<5 || b>2) S1 else S2;`

- *Handling* `a<5`*:*

```
if (a < 5) goto <Lx>; // <Lx> needs to be patched
```

- *Handling* `b>2`*:*

```
if (b > 2) goto <Ly>; // <Ly> needs to be patched
```

- *Handling* `..||..`*:*

```
if (a < 5) goto <Lx>;   //
if (b <= 2) goto <Lz>;  // <Lz> needs to be patched
```

- *Handling* `if ..  S1 else S2`*:*

```
    if (a < 5) goto L4;   // <Lx> is patched to L4
    if (b <= 2) goto L5;  // <Lz> is patched to L5
L4: [code for S1]  // then clause
    ...
```

Note that in the approach, the logical operations are implemented by properly patching operand expressions' labels; no actual new code is generated.

# Back-Patching Jump Labels

*Three-Address Code:*

Add two attributes to expression `E`:

`E.true` — position to jump to when `E` evals to true;
`E.false` — position to jump to when `E` evals to false.

```
E -> E1 relop E2
```

```
E.s := [ E1.s; E2.s;
         if (E1.t relop E2.t) goto E.true; E.false: ]
```

```
E -> E1 '||' E2
```

```
E1.true := E.true;
E1.false := new Label();
E2.true := E.true;
E2.false := E.false;
E.s := [ E1.s; E1.false: E2.s; ]
```

```
E -> E1 '&&' E2
```

```
E1.true := new Label();
E1.false := E.false;
E2.true := E.true;
E2.false := E.false;
E.s := [ E1.s; E1.true: E2.s; ]
```

```
E -> '!'  E1
```

```
E1.true := E.false;
E1.false := E.true;
E.s := E1.s;
```

# Back-Patching Jump Labels (cont.)

*IR Tree:*

```
E -> E1 relop E2
```

```
E.tr := (ESEQ [CJUMP relop E1.tr E2.tr E.true] null)
```

```
E -> E1 || E2
```

```
E1.true := E.true;
E1.false := new NAME();
E2.true := E.true;
E2.false := E.false;
E.tr := (ESEQ [ stmt(E1.tr);
                LABEL(E1.false); stmt(E2.tr); ]
              null)
```

```
E -> E1 '&&' E2
```

```
E1.true := new NANE();
E1.false := E.false;
E2.true := E.true;
E2.false := E.false;
E.tr := (ESEQ [ stmt(E1.tr);
                LABEL(E1.true); stmt(E2.tr); ]
              null)
```

```
E -> '!'  E1
```

```
E1.true := E.false;
E1.false := E.true;
E.tr := E1.tr;
```

# Converting Back to Value

What if we have

```
boolean x = a<5 || b>2;
```

We still need to generate a value for the Boolean expression!

This can be implemented by patching the two labels `E.true` and `E.false` for the Boolean expression `E` with two assignment statements for assigning 1 and 0, respectively.

```
Boolean expression E

t = new Temp();
E.true := new Label();
E.false := new Label();
L := new Label();
E.s := [ E.true: t := 1; goto L;
         E.false: t := 0; L: ]
E.t := t;
```

# New Arrays

`E -> 'newArray' E1`

*Issues:*

- *Storage allocation* — We follow Java's array storage convention — the length of array is stored as the zeroth element of the array. So the storage for a 10-element array actually has 11 cells.
- *Cell initialization* — All array elements are automatically initialized to 0.

*Pseudo IR Code:*

```
-------------------------------------------------------------
  L: new Label;
  t1,t2,t3: new Temps;
  E.s := [ E1.s;
           t1 := (E1.t + 1) * wdSize;  // storage size
           t2 := malloc(t1);           // t2 points to cell 0
           t2[0] := E1.t;              // store array length
           t3 := t2 + (E1.t * wdSize); // t3 points to last cell
           L:
           t3[0] := 0;                 // init a cell to 0
           t3 := t3 - wdSize;          // move down a cell
           if (t3 > t2) goto L; ]      // loop back
  E.t := t2;
-------------------------------------------------------------
```

# Array Elements

`E -> E1 '[' E2 ']'`

*Issues:*

- *Calculating address* — the address of an array element can be calculated using the following formula:

  address of $a[i] = \text{base}(a) + (i+1) \times \text{wdSize}$.

- *Bounds-checking* — Java performs array index bounds-checking to make sure it is within bounds.

*Pseudo IR Code:*

```
------------------------------------------------------------
  L1,L2: new Label;
  t1,t2,t3,t4: new Temps;
  E.s := [ E1.s; E2.s;
           t1 := E1.t[0];
           if (E2.t < 0) goto L1;
           if (E2.t >= t1) goto L1;
           t2 := E2.t + 1;
           t3 := t2 * wdSize;
           t4 := E1.t[t3];
           goto L2;
           L1:
           param E1.t;
           param E2.t;
           call arrayError,2;
           L2: ]
  E.t := t4;
------------------------------------------------------------
```

# Statements

- *Assignments:*

```
S -> E1 ':=' E2 ';'
```

```
-----------------------------------------------
S.s := [ E1.s; E2.s; E1.t := E2.t; ]
-----------------------------------------------
```

- *If Statement:*

```
S -> 'if' '(' E ')' 'then' S1 'else' S2
```

```
-----------------------------------------------
L1,L2,L3: new Labels;
E.true := L1;
E.false := L2;
S.s := [ E.s; L1: S1.s; goto L3; L2: S2.s; L3: ]
-----------------------------------------------
```

- *While Statement:*

```
S -> 'while' '(' E ')' S1
```

```
-----------------------------------------------
L1,L2,L3: new Labels;
E.true := L2;
E.false := L3;
S.s := [ L1: E.s; L2: S1.s; goto L1; L3: ]
-----------------------------------------------
```

- *Print Statement:*

```
S -> 'print E ';'
```

```
-----------------------------------------------
S.s := [ E.s; param E.t; call prInt,1; ]
-----------------------------------------------
```