CS323: Compilers
Spring 2023

Week 6: Semantic Processing (contd..)

Intermediate Representation

- Compilers need to synthesize code based on the 'interpretation' of the syntactic structure

- Code can be generated with the help of AST or

  can directly do it in semantic actions (recall: SDTs augment grammar rules with program fragments. Program fragments contain semantic actions.)

- Generated code can be directly executed on the machine or an intermediate form such as 3-address code can be produced.

3 Address Code (3AC)

- **What is it?** sequence of elementary program instructions
  - Linear in structure (no hierarchy) unlike AST
  - Format:
    ```
    op A, B, C    //means C = A op B.
    //op: ADDI, MULI, SUBF, DIVF, GOTO, STOREF etc.
    ```
  - E.g.

program text   **3-address code**

program text	3-address code	
INT x; FLOAT y, z; z:=x+y;	ADDF x y T1 STOREF T1 z	

INT a, b, c, d;
d = a-b/c;
	DIVI b c T1
	SUBI a T1 T2
	STOREI T2 d
		Comments:
		d = a-b/c; is broken into:
		t1 = b/c;
		t2 = a–t1;
		d = t2;

3 Address Code (3AC)

- **Why is it needed?** To perform *significant* optimizations such as:
    - common sub-expression elimination
    - statically analyze possible values that a variable can take etc.

    How?

    Break the long sequence of instructions into "basic blocks" and operate on/analyze a graph of basic blocks

3 Address Code (3AC)

- **How is it generated?** Choices available:
  1. Do a post-order walk of AST
  - Generate/Emit code as a string/data_object (seen later) when you visit a node
  - Pass the code to the parent node

    Parent generates code for self after the code for children is generated. The generated code is appended to code passed by children and passed up the tree

    ```
    data_object generate_code() {
        //preprocessing code
        data_object lcode=left.generate_code();
        data_object rcode=right.generate_code();
        return generate_self(lcode, rcode);
    }
    ```

  2. Can generate directly in semantic routines or after building AST

3 Address Code (3AC)

- Generating 3AC directly in semantic routines.

$\Downarrow$

```
INT x;
x:=3*4+5+6+7;
```

```
MULI 3 4 T1
ADDI T1 5 T2
ADDI T2 6 T3
ADDI T3 7 T4
STOREI T4 x
```

Comments:
x = 3*4+5+6+7 is broken into:
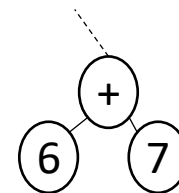t1 = 3*4;
t2 = 5+t1;
t3 = 6+t2;
t4 = 7+t3;
x = t4

- Walk the AST in post-order and infer at an internal node (labelled op) that it computes a constant expression

$\Downarrow$

```
INT x;
x:=3*4+5+6+7;
```

```
STOREI 30 x
```

Comments:

L-values and R-values

- Need to distinguish between meaning of identifiers appearing on RHS and LHS of an assignment statement

```
i := 5;        //RHS specifies data that is computed/read.
i := i + 1;    LHS specifies address where data is stored.
```

- **L-values**: addresses which can be loaded from or stored into
- **R-values**: data often loaded from address
    - Expressions produce R-values
- Assignment statements: `L-value := R-value;`

```
a := a;
```

a refers to memory location named a. We are storing into that memory location (L-value)

a refers to data stored in the memory location named a. We are loading from that memory location to produce R-value

Temporaries

- Earlier saw the use of temporaries e.g.

```
INT x;
FLOAT y, z;
z:=x+y;
```
```
ADDF x y T1
STOREF T1 z
```

- Think of them as unlimited pool of registers with memory to be allocated later

- Optionally declare them in 3AC. Name should be unique and should not appear in program text

```
INT x
FLOAT y z T1
ADDF x y T1
STOREF T1 z
```

- Temporary can hold L-value or R-value

Temporaries and L-value

- Yes, a temporary can hold L-value. Consider:

    ```
    a := &b;   //& is address-of operator. R-value
    of a is set to L-value of b.
    //expression on the RHS produces data that is
    an address of a memory location.
    ```

    Recall: L-Value = address which can be loaded
    from or stored into, R-Value = data (often)
    loaded from addresses.

Take L-value of b, don't load from it, treat it as an R-value and store the resulting data in a temporary

Dereference operator

- Consider:

```
*a := b;   //* is dereference operator. R-value
of a is set to R-value of b.
//expression on the LHS produces data that is
an address of a memory location.
```

a appearing on LHS is loaded from to produce R-value. That R-value is treated as an address that can be stored into.

Take R-value of a, treat it as an L-value (address of a memory location) and then store RHS data

*Summary: pointer operations & and * mess with meaning of L-value and R-values*

Observations

- Identifiers appearing on LHS are (normally) treated as L-values. Appearing on RHS are treated as R-values.

    - So, when you are visiting an `id` node in an AST, you cannot generate code (load-from or store-into) until you have seen how that identifier is used. => until you visit the parent.

- Temporaries are needed to store result of current expression

- a `data_object` should store:
    - Code
    - L-value or R-Value or constant
    - Temporary storing the result of the expression

Simple cases

- Generating code for constants/literals

  - Store constant in temporary

  - Optional: pass up flag specifying this is a constant

- Generating code for identifiers

  - Generated code depends on whether identifier is used as L-value or R-value

    - Is this an address? Or data?

  - One solution: just pass identifier up to next level

    - Mark it as an L-value (it's not yet data!)

    - Generate code once we see how variable is used

Generating code for expressions

- Create a new temporary for result of expression

- Examine data-objects from subtrees

- If temporaries are L-values, load data from them into new temporaries

  - Generate code to perform operation

  - In project, no need to explicitly load (variables can be operands)

- If temporaries are constant, can perform operation immediately

  - No need to perform code generation!

- Store result in new temporary

  - Is this an L-value or an R-value?

- Return code for entire expression

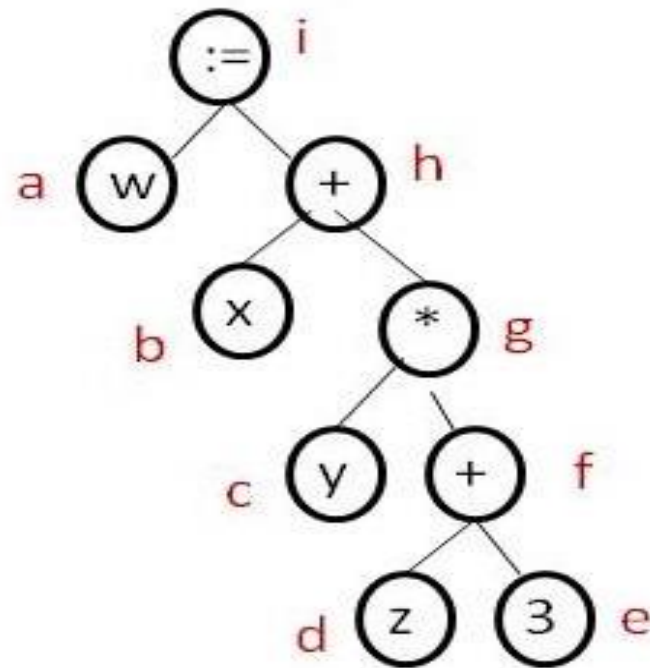# Example - assignment statement
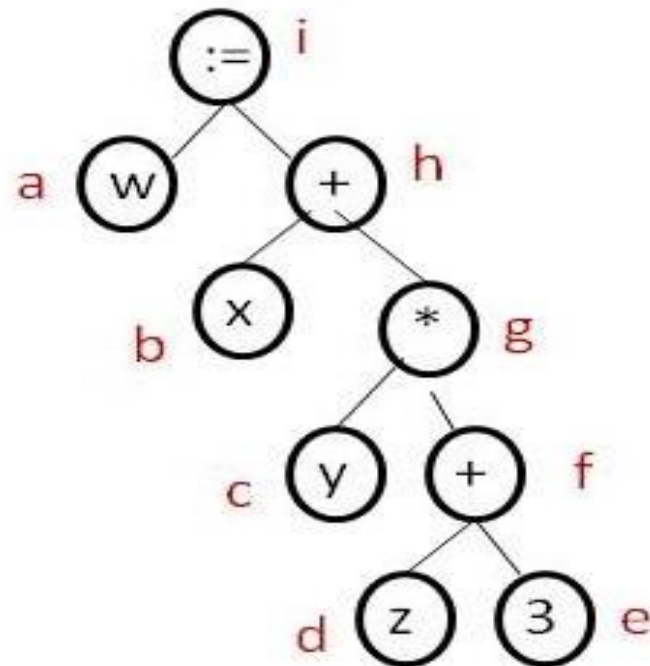
AST for ⟹

`w:=x+y*(z+3);`

Visit Node a:
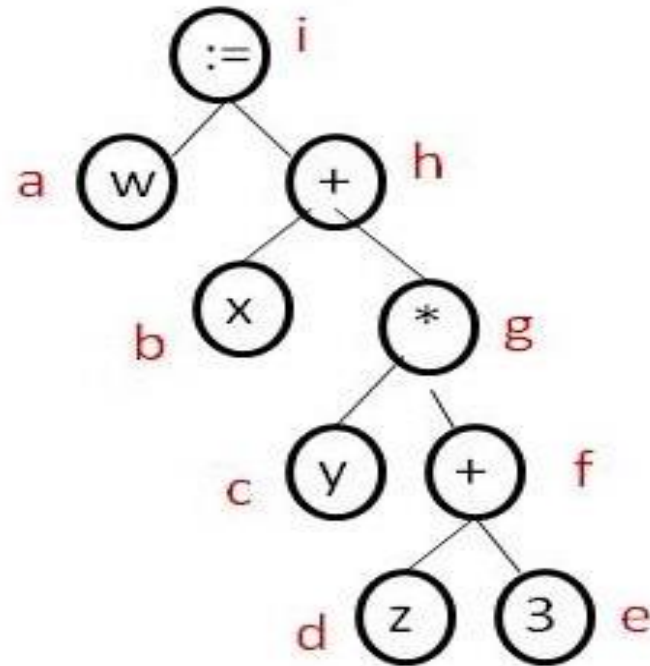 Temp: w
  Type: l-value
  Code: --

Example - assignment statement

AST for ⟹

`w:=x+y*(z+3);`

Visit Node b:
 Temp: x
 Type: l-value
 Code: --

Example - assignment statement

AST for $\Longrightarrow$

`w:=x+y*(z+3);`

Visit Node c:
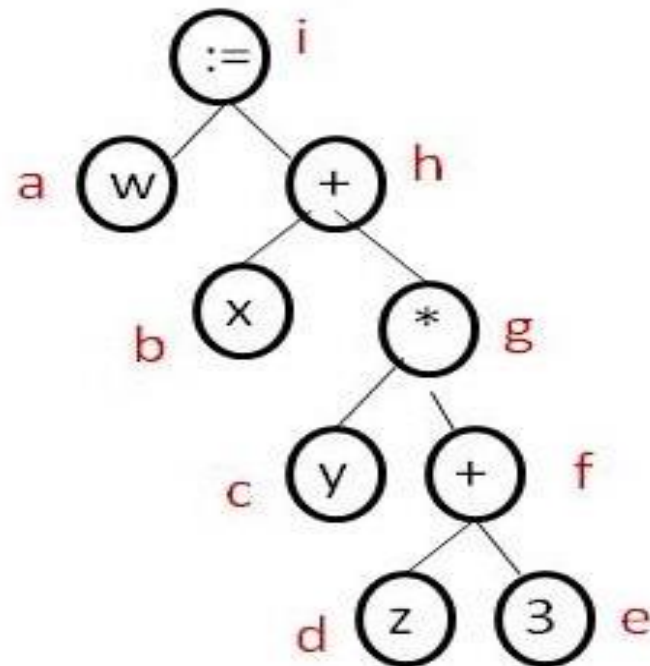   Temp: y
    Type: l-value
    Code: --

Example - assignment statement

AST for ⟹

`w:=x+y*(z+3);`

Visit Node d:
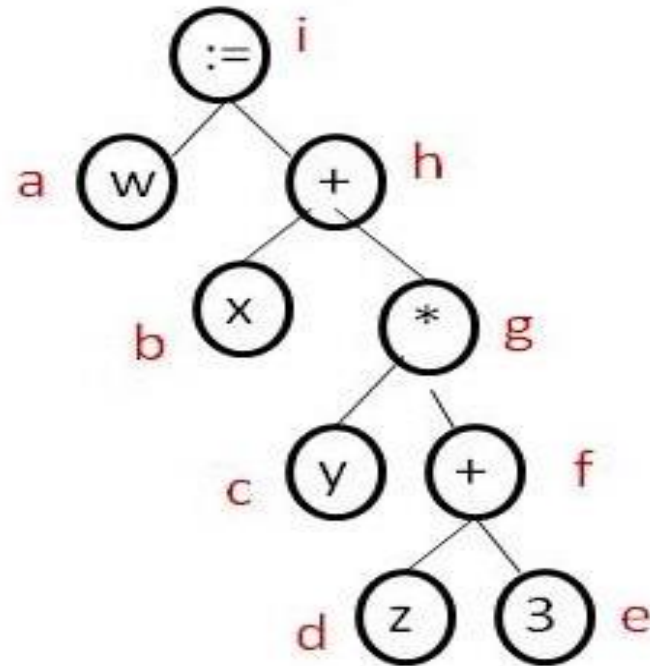   Temp: z
    Type: l-value
    Code: --

Example - assignment statement

AST for $\Longrightarrow$

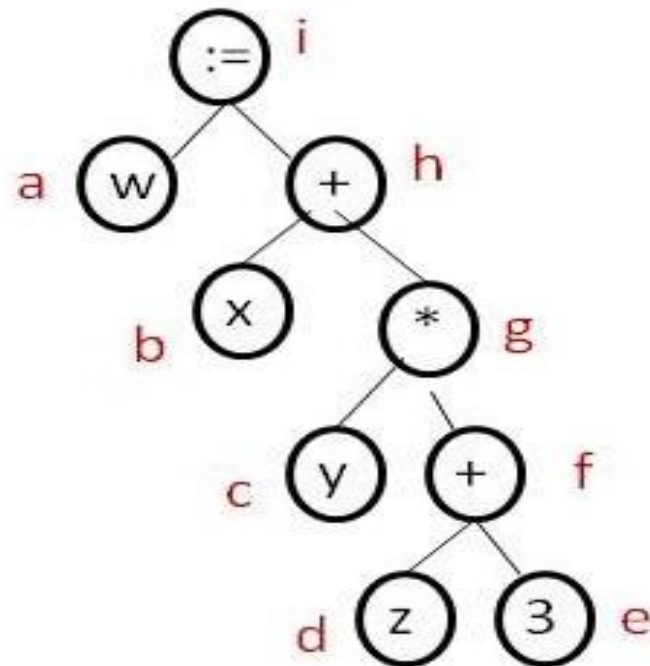`w:=x+y*(z+3);`

Visit Node e:
    Temp: 3
    Type: constant
    Code: --

Example - assignment statement

AST for ⟹

`w:=x+y*(z+3);`

Visit Node f:
    Temp: T1
    Type: R-value
    Code:

        LD z T2

        ADD T2 3 T1

Example - assignment statement
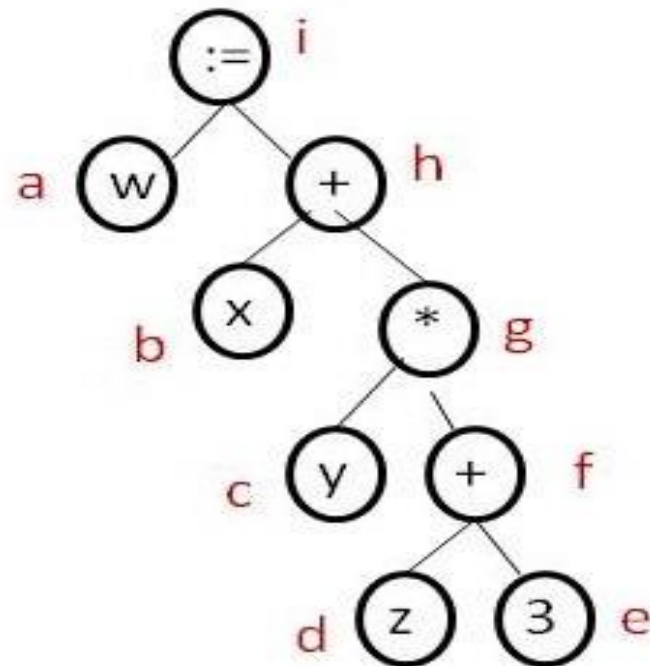
AST for $\Longrightarrow$

`w:=x+y*(z+3);`

Visit Node g:

    Temp: T3

    Type: R-value

    Code:

        LD y T4

        LD z T2

        ADD T2 3 T1

        MUL T4 T1 T3

Example - assignment statement

AST for

`w:=x+y*(z+3);`

Visit Node h:

    Temp: T5

    Type: R-value

    Code:

```
LD x T6
LD y T4
LD z T2

ADD T2 3 T1

MUL T4 T1 T3
ADD T6 T3 T5
```
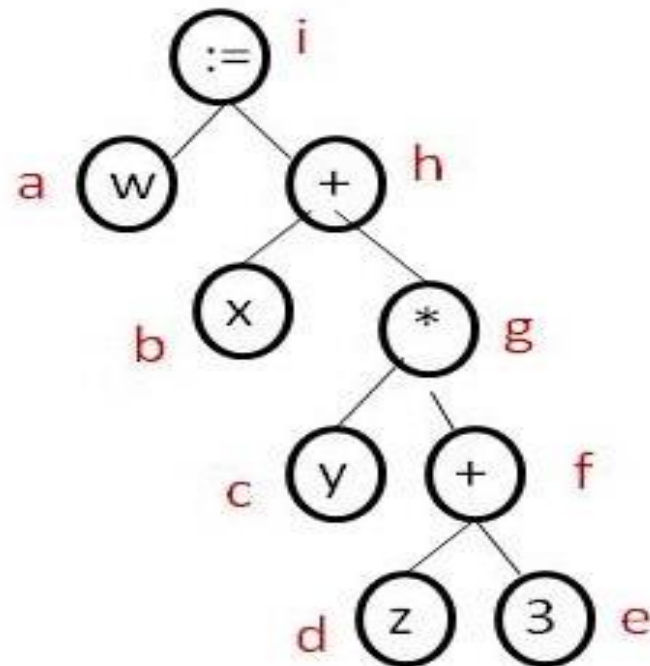
Example - assignment statement

AST for ⟹

`w:=x+y*(z+3);`

Visit Node i:

    Temp: NA

    Type: NA

    Code:

        LD x T6
        LD y T4
        LD z T2

        ADD T2 3 T1

        MUL T4 T1 T3
        ADD T6 T3 T5
        ST T5 w
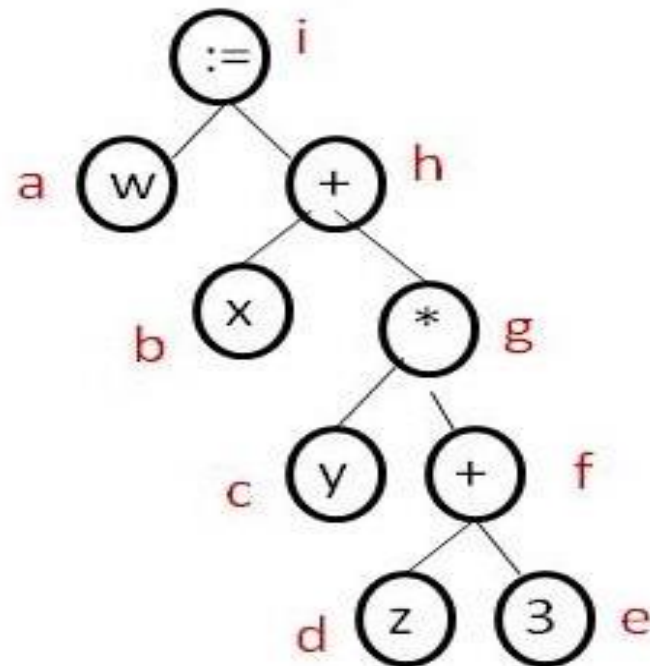
If statements

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

If statements
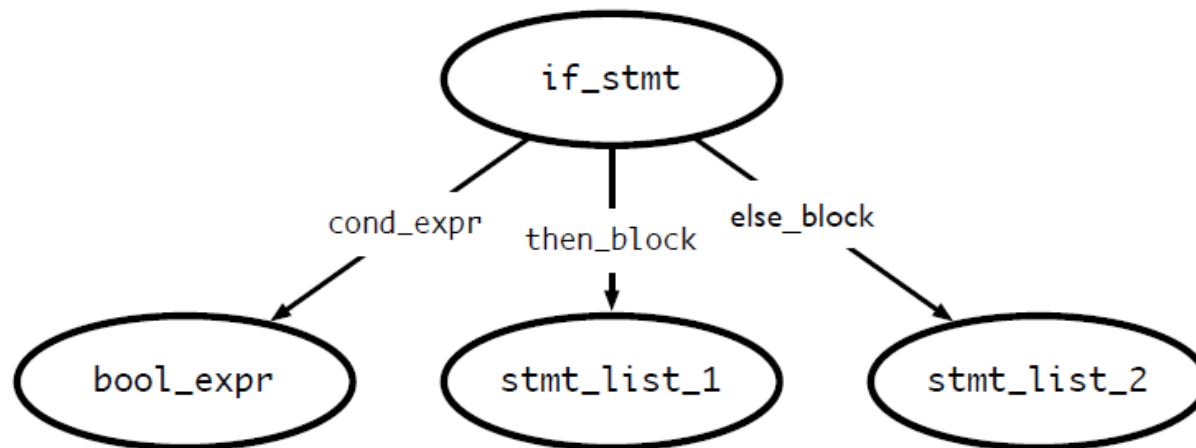
Generating code for ifs

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

```
<code for bool_expr_1>
j<!op> ELSE_1
<code for stmt_list_1>
jmp OUT_1
ELSE_1:
  <code for stmt_list_2>
OUT_1:
```

Slide courtesy: Milind Kulkarni

Notes on code generation

- The <op> in j<!op> is dependent on the type of comparison you are doing in <bool_expr>

- When you generate JUMP instructions, you should also generate the appropriate LABELs

- Remember: labels have to be unique!

Code-generation – if-statement

Program text	3AC
INT a, b;	

Code-generation – if-statement

Program text	3AC	
INT a, b;		Make entries in the symbol table

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;
a := 2;
```

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;
a := 2;
```
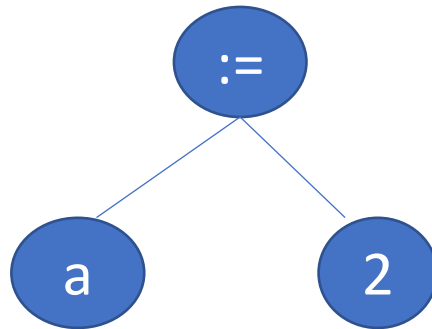

1. "a" is left-child, type=l-val. No code generated. *Return an object containing identifier details after verifying that "a" is present in the symbol table.*
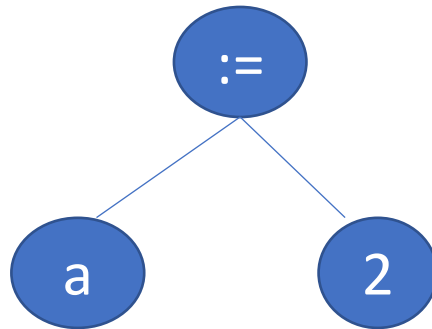
Code-generation – if-statement

Program text	3AC

`INT a, b;`
`a := 2;`

1. "a" is left-child, type=l-val. No code generated. Pass up the identifier.
2. "2" is right-child, type=const. No code generated.

Code-generation – if-statement
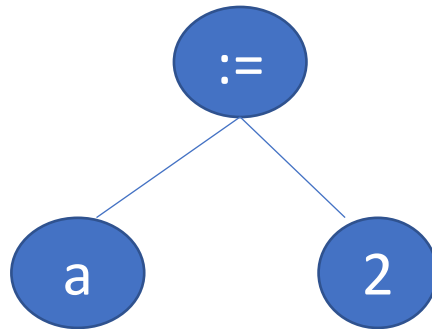
Program text   **3AC**

```
INT a, b;
a := 2;
```

```
        :=
       /  \
      a    2
```

1. "a" is left-child, type=l-val. No code generated. Pass up the identifier.
2. "2" is right-child, type=const. No code generated.
3. Create a temporary T1 to store the result of the expression

Code-generation – if-statement
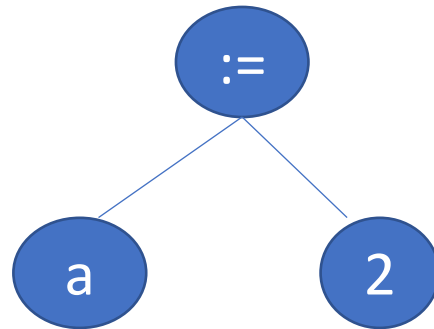
```
INT a, b;
a := 2;
```


1. "a" is left-child, type=l-val. No code generated. Pass up the identifier.
2. "2" is right-child, type=const. No code generated.
3. Current node stores the op ':='. A call to process_op stores the RHS data in LHS

Code-generation – if-statement

Program text	3AC
INT a, b; a := 2;	STOREI 2 T1 STOREI T1 a
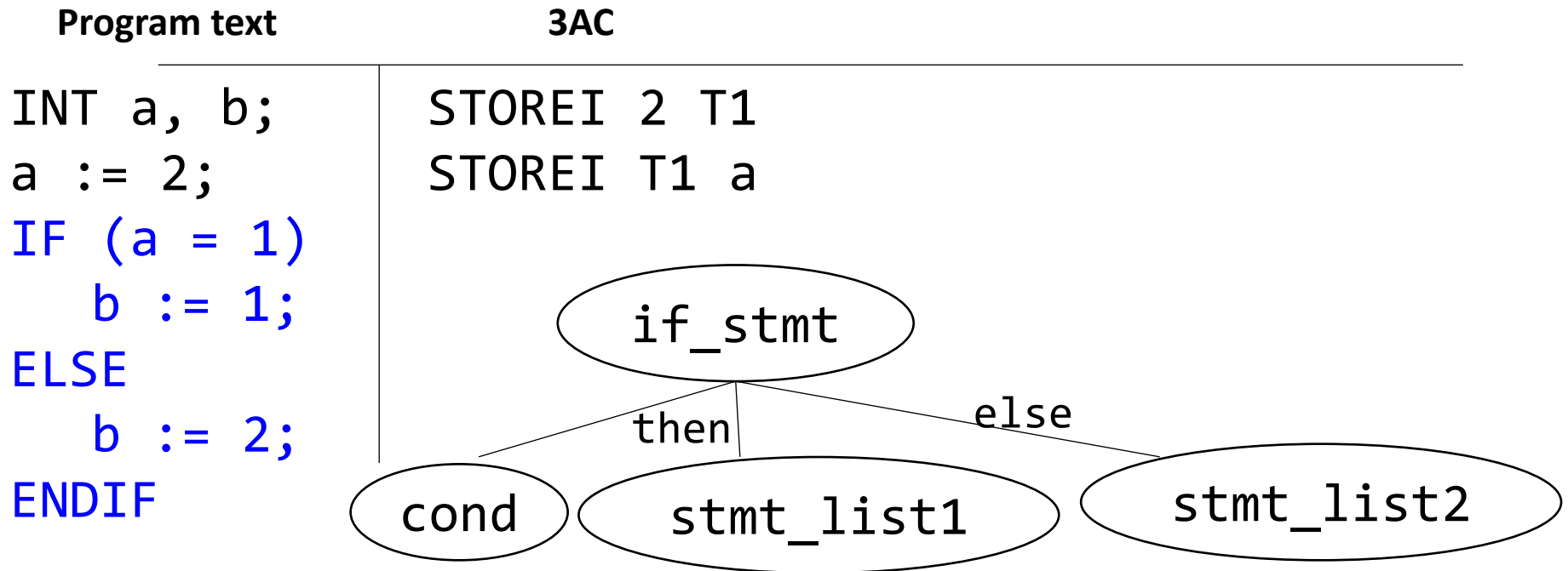
Code-generation – if-statement

Program text   **3AC**

```
INT a, b;
a := 2;
IF (a = 1)
    b := 1;
ELSE
    b := 2;
ENDIF
```

```
STOREI 2 T1
STOREI T1 a
```

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;        STOREI 2 T1
a := 2;          STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```


1. Generate code for (cond)

Code-generation – if-statement

Program text	3AC
`INT a, b;`	`STOREI 2 T1`
`a := 2;`	`STOREI T1 a`
`IF (a = 1)`	
`    b := 1;`	if_stmt
`ELSE`	
`    b := 2;`	then / else
`ENDIF`	cond stmt_list1 stmt_list2

1. Generate code for (cond)

```
STOREI 1 T2
NE a T2 label1
```
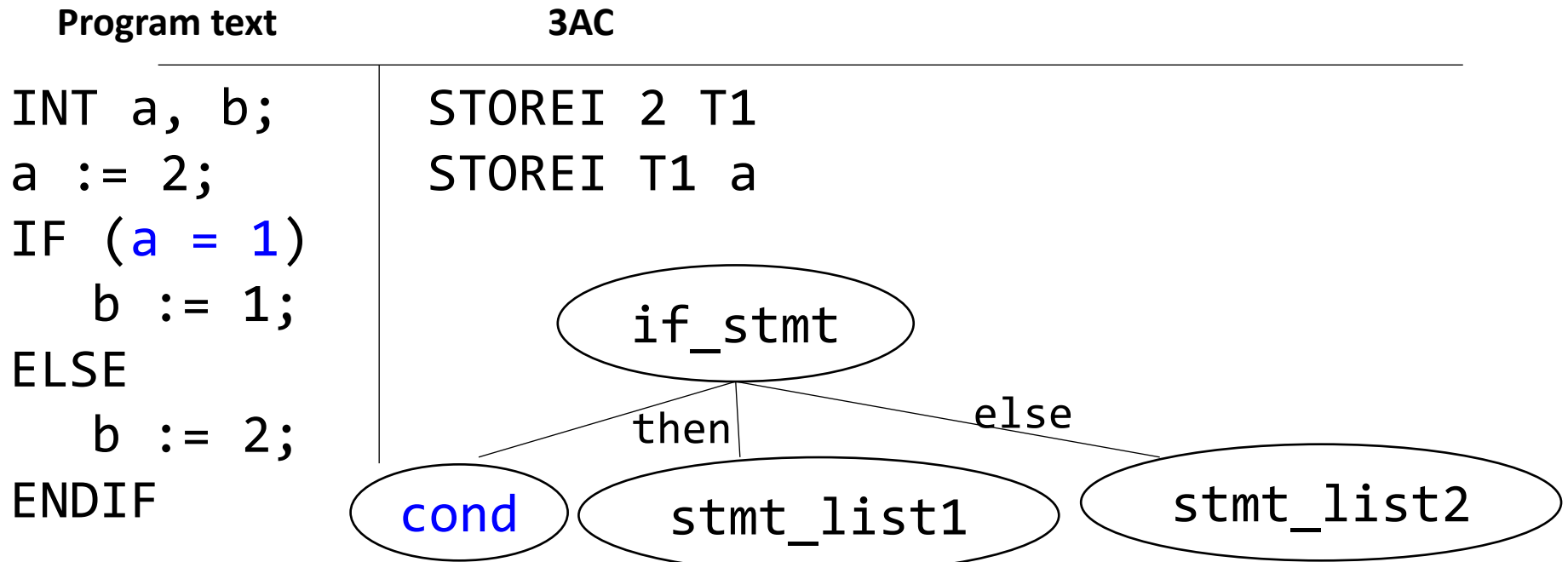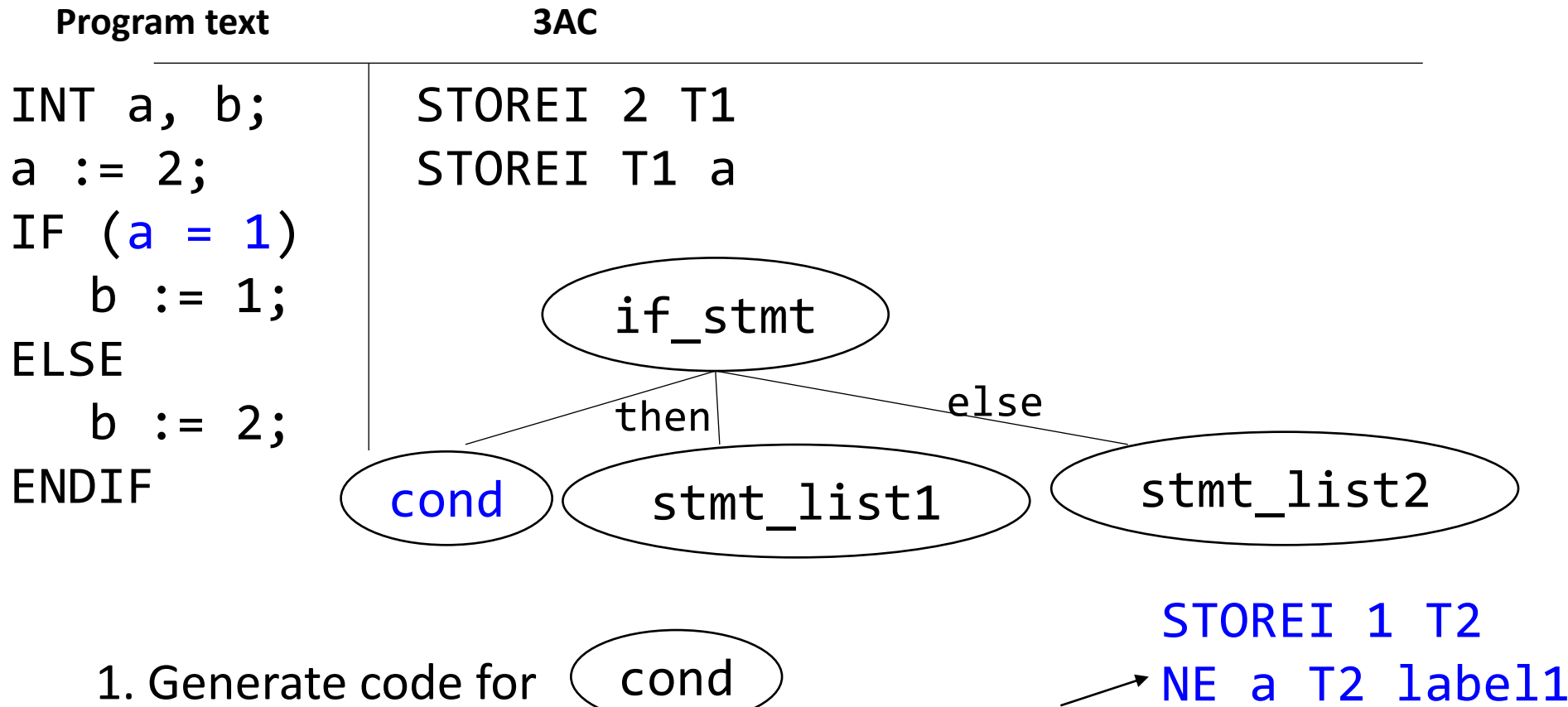
Note that to generate this instruction when cond node is visited, we need information about the label. This information can be passed on as a semantic record for the child node of the if construct. The record can be created by the IF construct (when the keyword IF is seen) and would be updated subsequently.

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
    b := 1;
ELSE
    b := 2;
ENDIF
```

if_stmt

then   else

cond   stmt_list1   stmt_list2

2. Generate code for   stmt_list1

Code-generation – if-statement

Program text	3AC
INT a, b;	STOREI 2 T1
a := 2;	STOREI T1 a
IF (a = 1)	
b := 1;	
ELSE	
b := 2;	
ENDIF	

if_stmt

then   else

cond   stmt_list1   stmt_list2

2. Generate code for   stmt_list1

STOREI 1 T3
STOREI T3 b

Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a

if_stmt

then

else

cond

stmt_list1

stmt_list2

3. Generate code for

stmt_list2

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;        STOREI 2 T1
a := 2;          STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

if_stmt

then   else

cond   stmt_list1   stmt_list2

3. Generate code for   stmt_list2

JUMP label2
LABEL label1
STOREI 2 T4
STOREI T4 b
JUMP label2

The statements shown in red can be part of the semantic routines that correspond to handling the else part.

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

if_stmt

then   else

cond   stmt_list1   stmt_list2
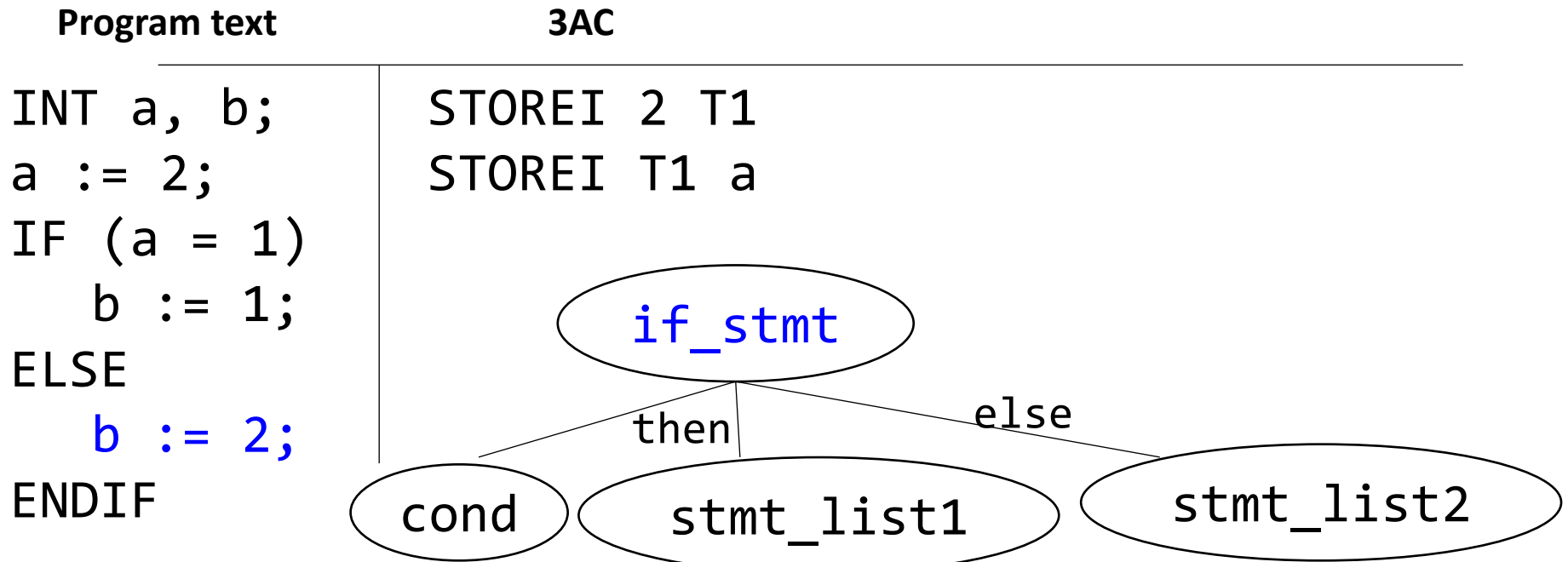
3. Generate code for   if_stmt

LABEL label2

Code-generation – if-statement

| Program text | 3AC |

```
INT a, b;     STOREI 2 T1    //a := 2
a := 2;       STOREI T1 a
IF (a = 1)    STOREI 1 T2    //a = 1?
   b := 1;    NE a T2 label1
ELSE          STOREI 1 T3    //b := 1
   b := 2;    STOREI T3 b
ENDIF         JUMP label2    //to out label
              LABEL label1   //else label begins here
              STOREI 2 T4    //b := 2
              STOREI T4 b
              JUMP label2    //jump to out label
              LABEL label2   //out label
```

Can also generate this code after seeing the token ENDIF (rather than as part of the routine that is executed when the whole production is matched)
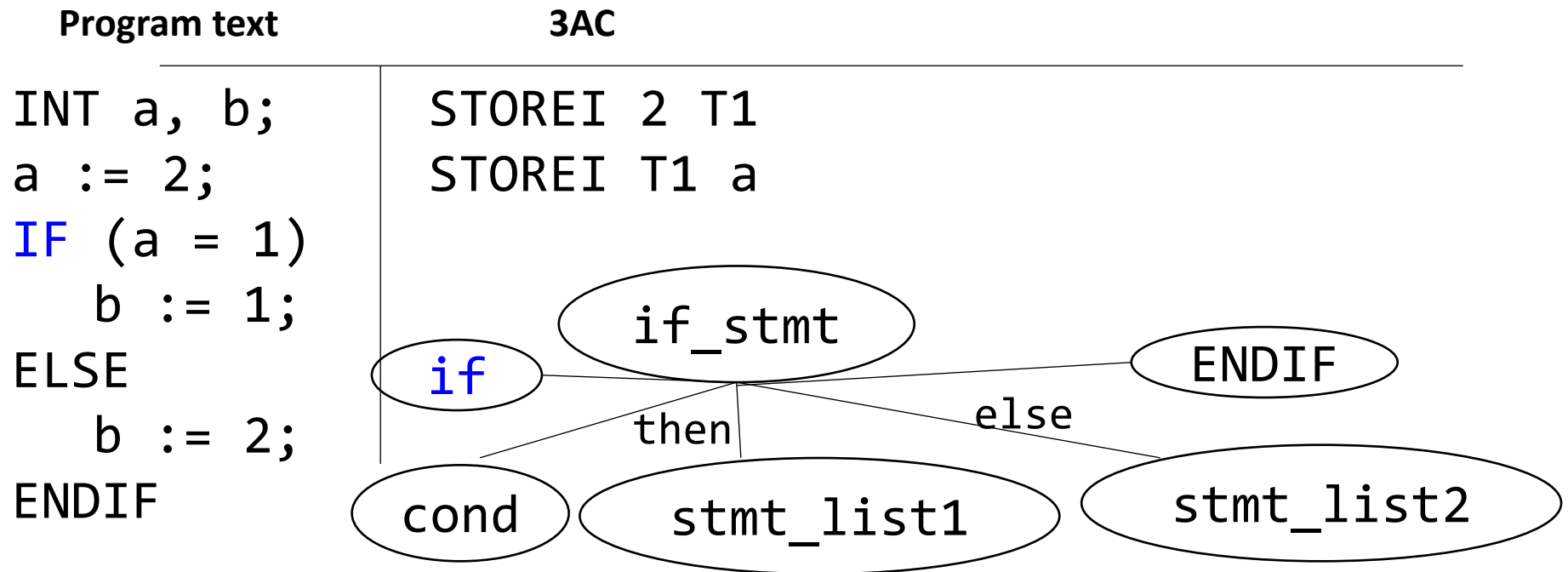
Jumps and Labels?

- Who will generate labels?

- When will the labels be generated?

- To what addresses will the labels be associated with?

How are targets of jumps decided?

If construct with semantic actions

- If_stmt->if #start_if <b_expr> #testif then <stmt_list> <else_part> endif; #gen_out_label

- else_part->else #gen_jump #gen_else_label <stmt_list>

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
    b := 1;
ELSE
    b := 2;
ENDIF
```

1 Generate out label and store it in semantic record of if_stmt (`label1`)

The #start_if routine is responsible for this

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

if_stmt

if — ENDIF

then

else

cond   stmt_list1   stmt_list2

2 Generate code for   cond

STOREI 1 T2
NE a T2 label1

cond

expr1   comp   expr2

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

```
STOREI 2 T1
STOREI T1 a
```

if_stmt

if   ENDIF
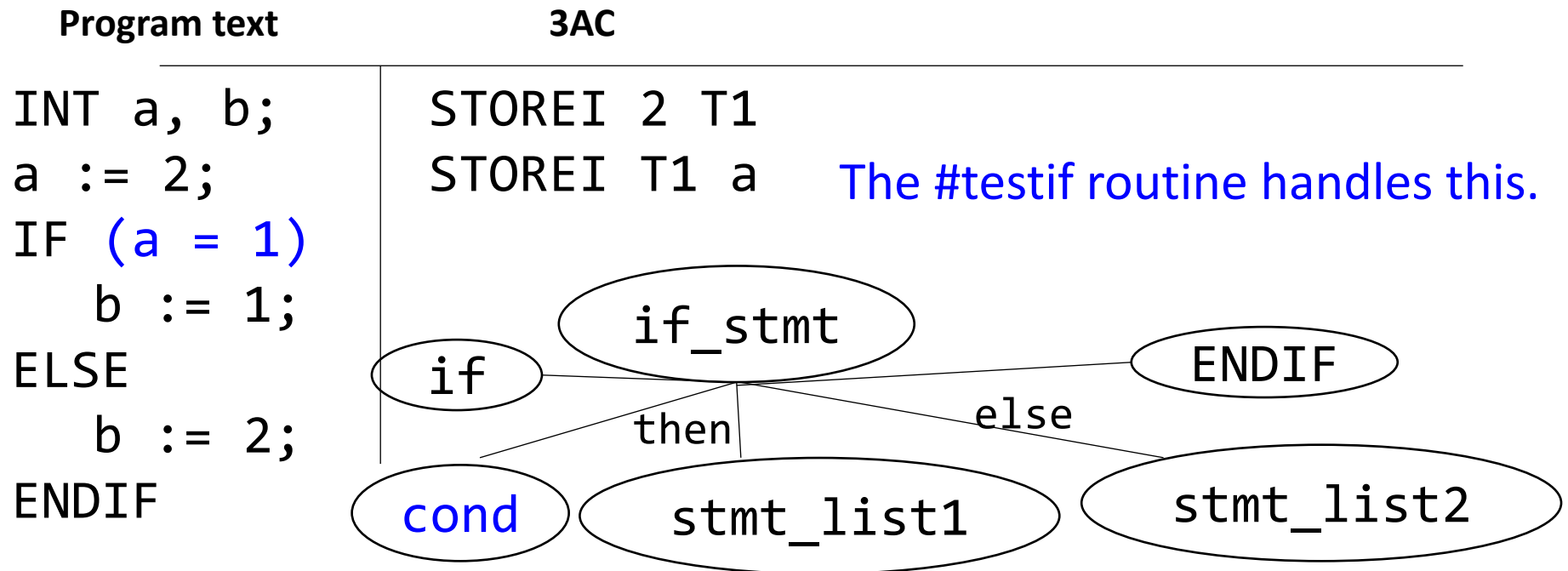
then   else

cond   stmt_list1   stmt_list2
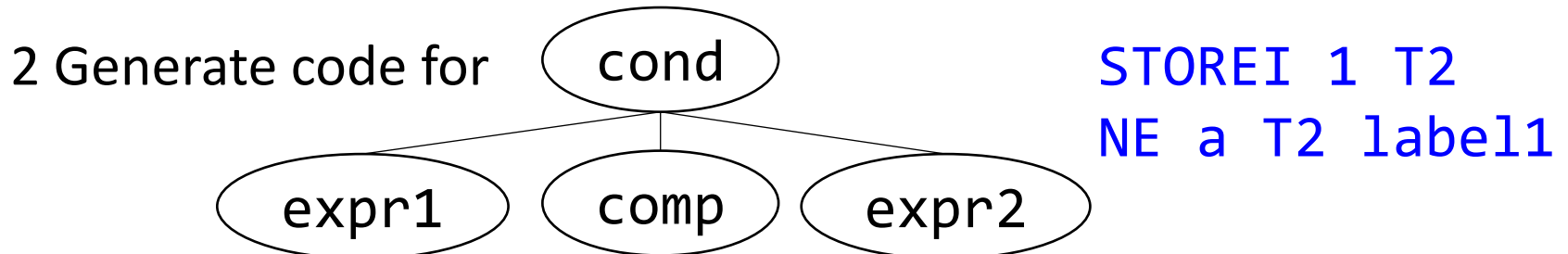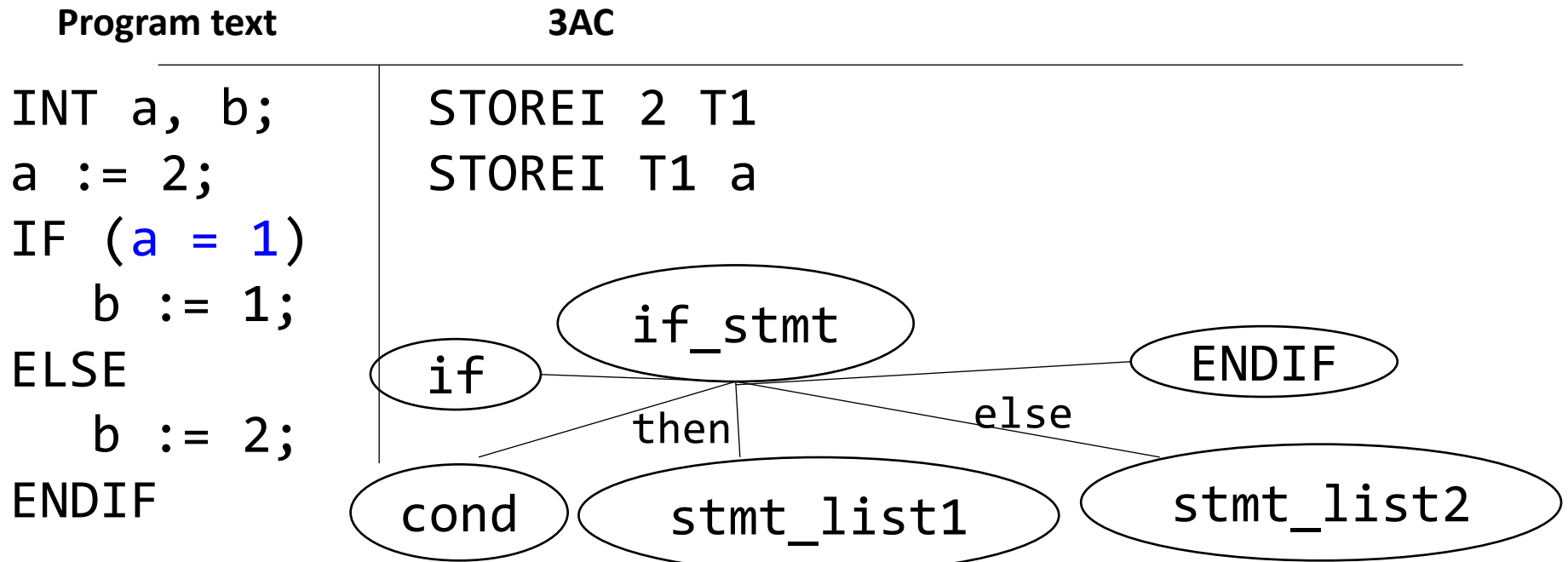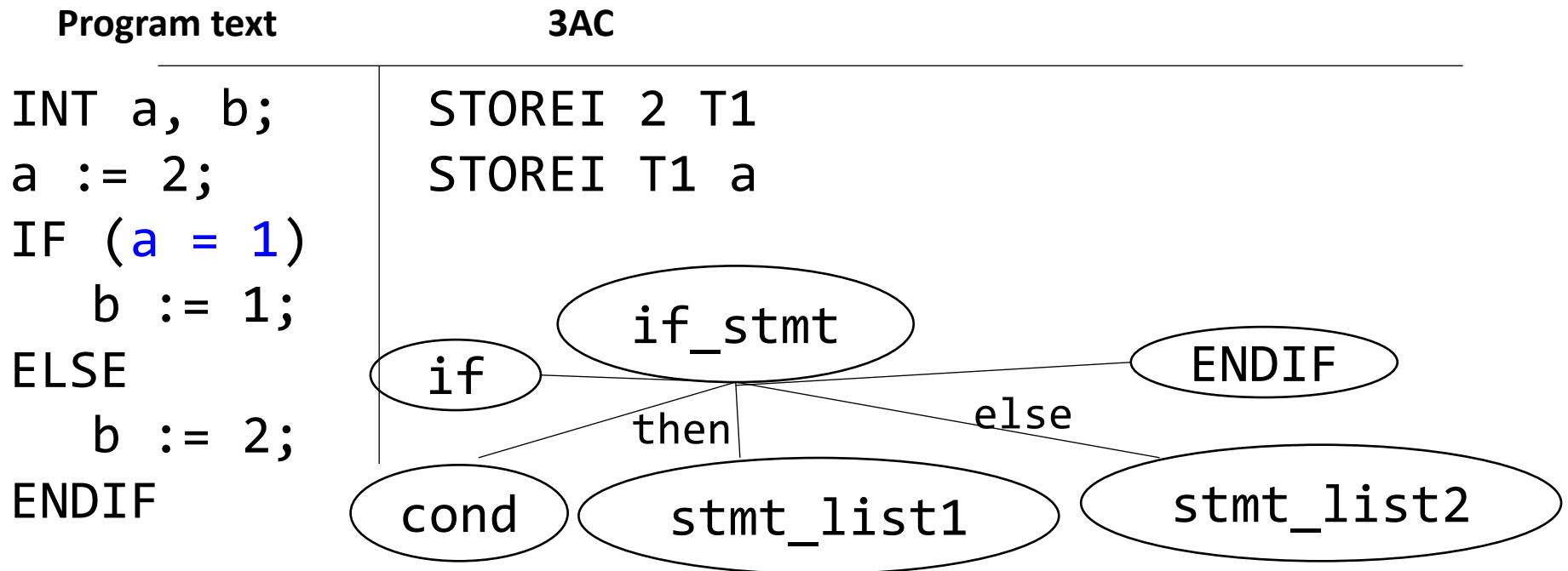
2. Store the result of calling `process_op`, where op is "=", in the node cond (bool_expr1=false)

```
STOREI 1 T2
NE a T2 label1
```

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;        STOREI 2 T1
a := 2;          STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

The #testif routine handles this.

2. Create a label for the next else part (label 2). Generate statement: `JUMP0 T2 label2`
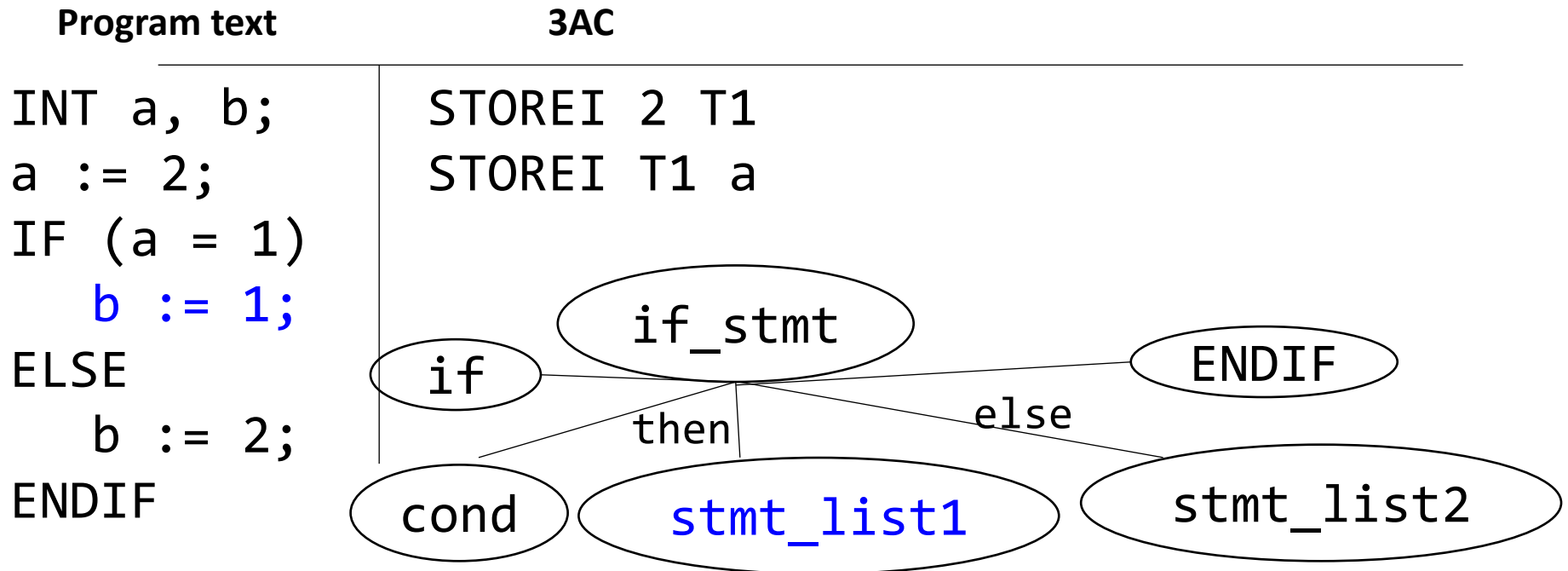
Code-generation – if-statement

Program text	3AC
`INT a, b;`	`STOREI 2 T1`
`a := 2;`	`STOREI T1 a`
`IF (a = 1)`	
`    b := 1;`	
`ELSE`	
`    b := 2;`	
`ENDIF`	

3. Generate code for `stmt_list1`
`STOREI 1 T3`
`STOREI T3 b`

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

Routine #gen_jump handles this

if_stmt

if

ENDIF

then

else

cond

stmt_list1

stmt_list2
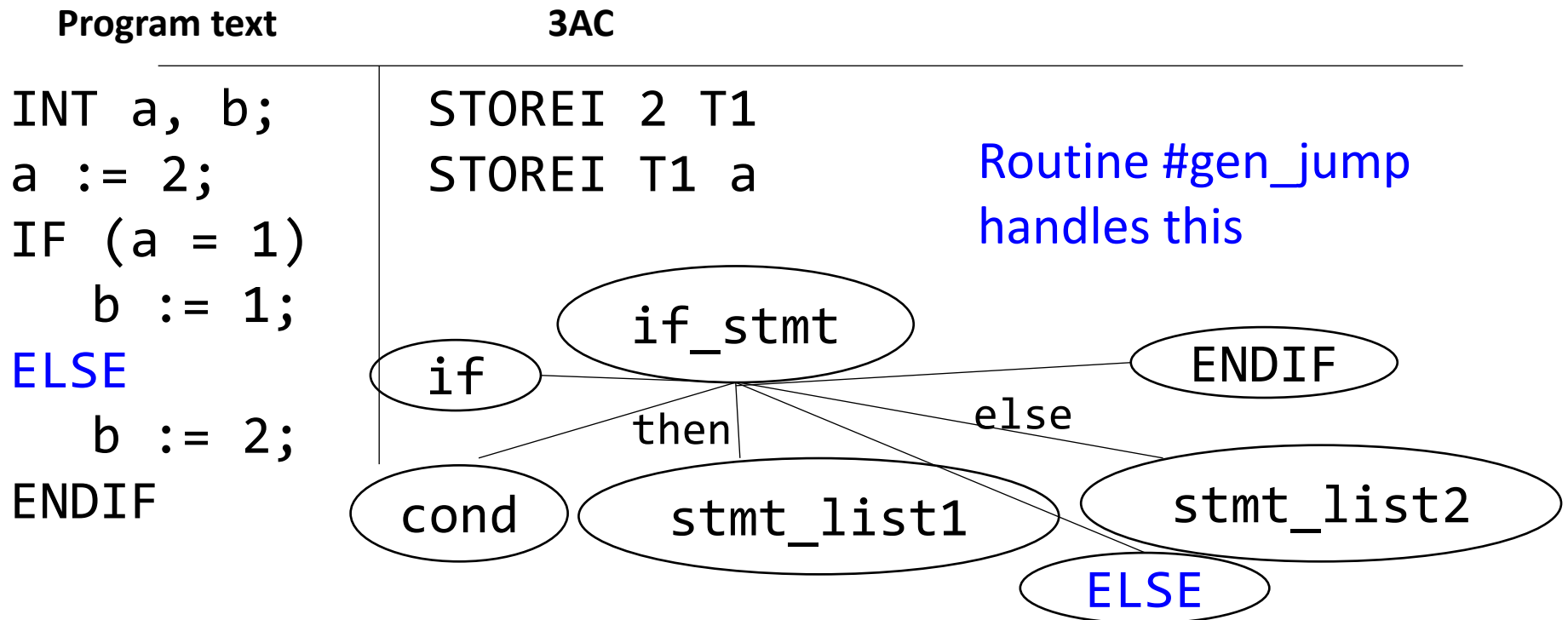
ELSE

4. Generate unconditional jump to out label (label1).
`JUMP label1`

Code-generation – if-statement

INT a, b;
a := 2;
IF (a = 1)
    b := 1;
ELSE
    b := 2;
ENDIF

```
STOREI 2 T1
STOREI T1 a
```

Routine
#gen_else_label
handles this
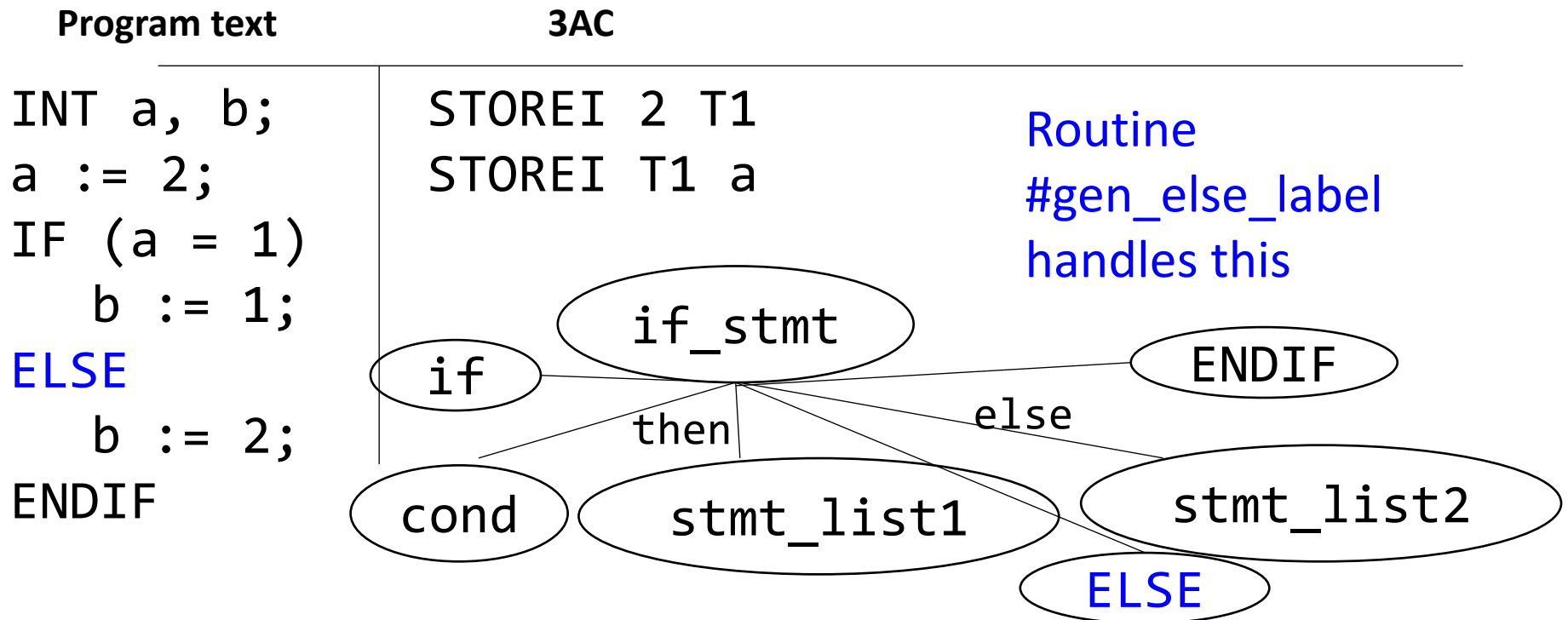
5. Associate else part label (`label2`) with address of next instruction i.e. generate a statement: `LABEL label2`

Code-generation – if-statement
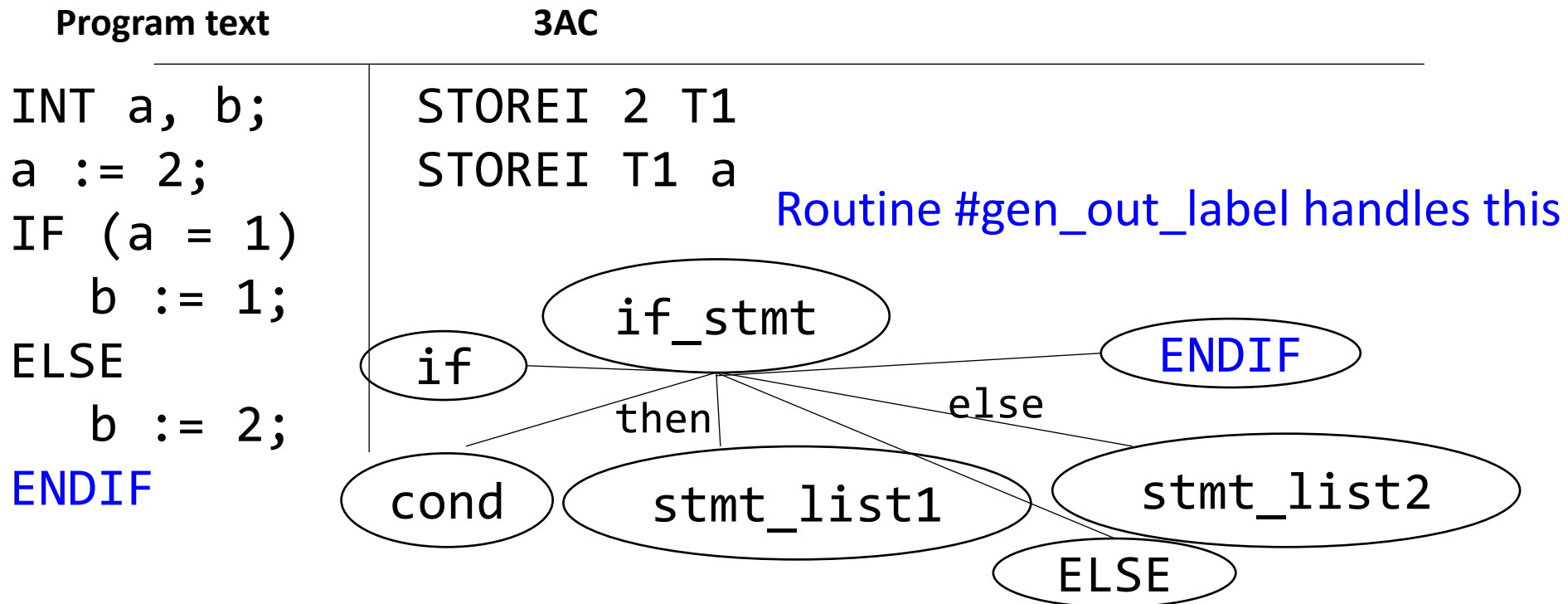
Program text   **3AC**

```
INT a, b;        STOREI 2 T1
a := 2;          STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```
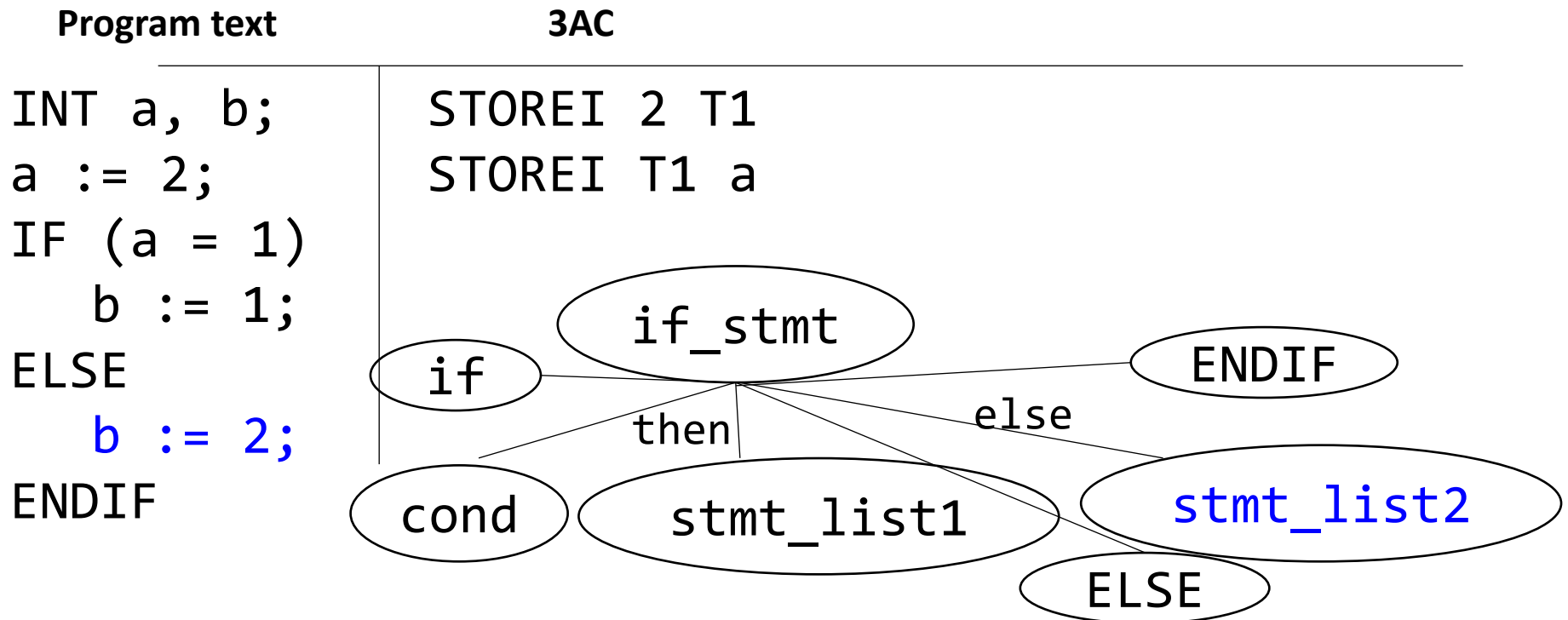

5. Generate code for `stmt_list2`
```
STOREI 2 T4
STOREI T4 b
```

Code-generation – if-statement

Program text   **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

Routine #gen_out_label handles this

5. Associate out label (label1) with address of next instruction
i.e. generate a statement: LABEL label1

Observations

- We added semantic actions with tokens IF, ELSE, ENDIF

- Generated code is equivalent but not exact
  - e.g. "NE a T2 label1" is replaced with an equivalent "JUMP0 bool_expr label1"

- Done in one pass ?

Will this approach work when generating machine code directly?

If construct with semantic actions

- If_stmt->if #start_if <b_expr> #testif then <stmt_list> <optional_elsif_part> <else_part> endif; #gen_out_label

- <optional_elsif_part>-> elsif #gen_jump #gen_else_label <b_expr> #testif then <stmts>

- Else_part->else #gen_jump #gen_else_label <stmt_list>

Exercise: augment the grammar rule to handle elsif blocks.

Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSIF (TRUE)
   b := 2;
ENDIF
	STOREI 2 T1 //a := 2
	STOREI T1 a
	STOREI 1 T2 //a = 1?
	NE a T2 label1
	STOREI 1 T3 //b := 1
	STOREI T3 b
	JUMP label2 //to out label
	LABEL label1 //elsif label
	STOREI 1 T4 //TRUE can be handled by checking 1 = 1?
	STOREI 1 T5
	NE T4 T5 label3 //jump to the next elsif label
	STOREI 2 T6 //b := 2
	STOREI T6 b
	JUMP label2 //jump to out label
	LABEL label3 //out label
	LABEL label2 //out label

do-while

- **do{S}while(B);** //S is executed at least once and again and again and again... while B remains true

do-while

- do{S}while(B); //S is executed at least once
  and again and again and again... while B remains true

do #beginloop {S} while(B) #testloop ; #endloop

```
LOOP:
    <stmt_list>
    <bool_expr>
    j<!op> OUT
    jmp LOOP
OUT:
```

#beginloop
    create labels LOOP and OUT
    generate LABEL LOOP

#testloop
    Check if the conditional statement B
    has the correct type (Boolean)

#endloop
    Generate JUMP0 OUT
    Generate JUMP LOOP

repeat-until

- repeat{S}until(B); //S is executed at least once and again and again and again... while B remains false

repeat-until

- repeat{S}until(B); //S is executed at least once and again and again and again... while B remains false

```
LOOP:
    <stmt_list>
    <bool_expr>
    j<!op> LOOP
OUT:
```