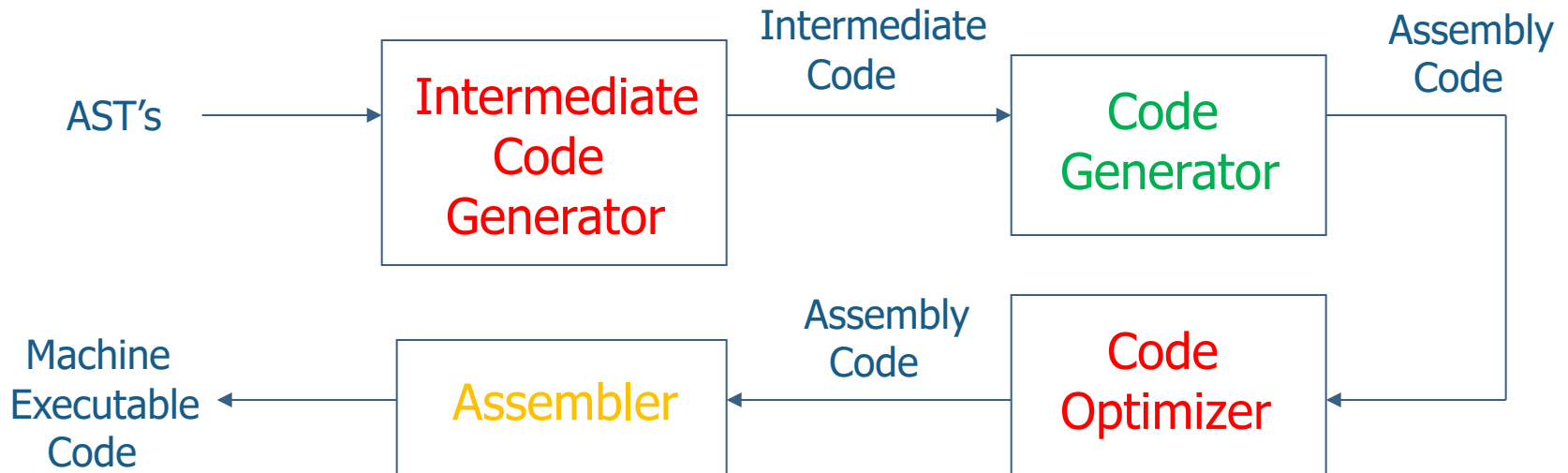


# Intermediate Code Generation

CIS\*4650 Compilers  
(Winter 2025)

# Review

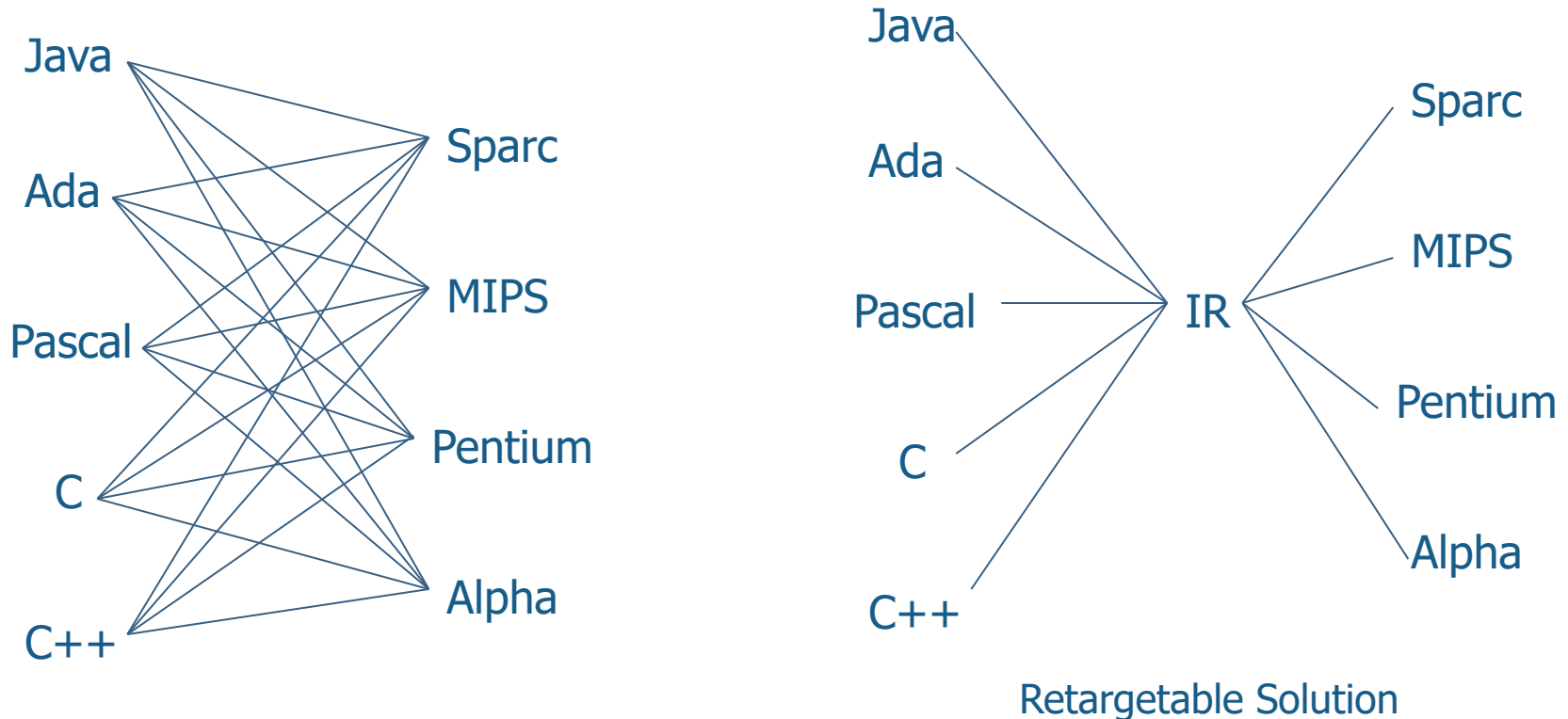
- Code generation: generate executable code for a target machine



Note that for Checkpoint Three, we will only implement "Code Generator" and run the assembly code on the "TM Simulator". "Intermediate Code Generator" and "Code Optimizer" will be skipped for simplifications.

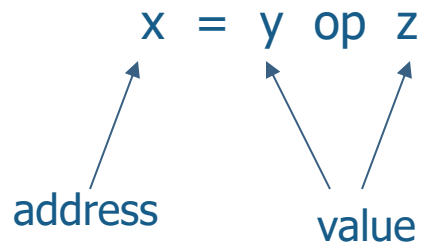
# Intermediate Representation (IR)

- Can be AST's, but linear sequences with jumps are more preferred



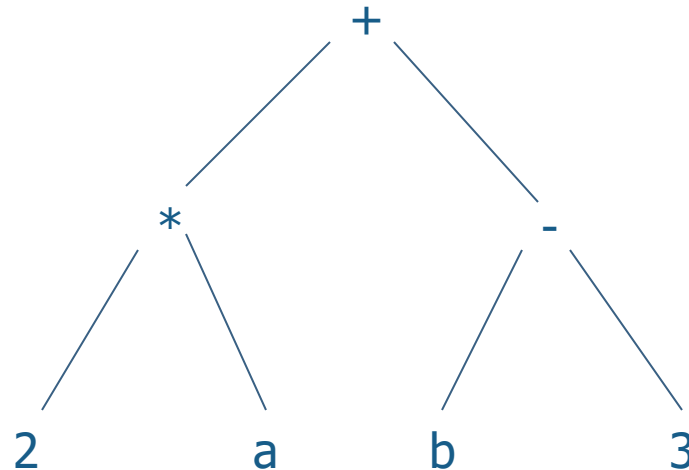
# Three-Address Code

- Basic form: maximum of three addresses
  - Compiler needs to generate names for temporary variables



e.g.,  $2 * a + (b - 3)$

$t1 = 2 * a$   
 $t2 = b - 3$   
 $x = t1 + t2$



# Bigger Example

{ Sample Tiny program for  
computing factorial }

```
read x;
if 0 < x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1;
  until x = 0;
  write fact
end
```

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

# Code Generation for Expressions

Grammar:     $\text{exp} \rightarrow \text{id} = \text{exp} \mid \text{aexp}$   
               $\text{aexp} \rightarrow \text{aexp} + \text{factor} \mid \text{factor}$   
               $\text{factor} \rightarrow ( \text{exp} ) \mid \text{num} \mid \text{id}$

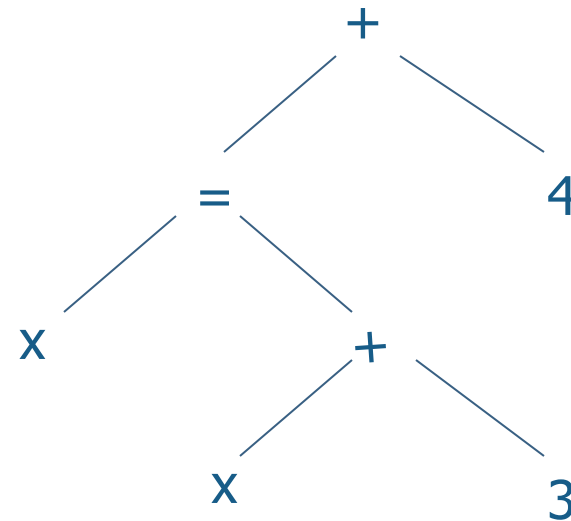
e.g.,  $(x = x + 3) + 4$

Intermediate Code:

$t1 = x + 3$

$x = t1$

$t2 = x + 4$



# Code Generation for Expressions

```
void genCode( Exp tree ) {      // newtemp() returns a new name such as t1, t2, etc.
    String codestr = "";
    if( tree != null ) {
        if( tree instanceof OpExp ) {
            genCode( tree.left );
            genCode( tree.right );
            tree.temp = newtemp();    // each node is added with a "temp" string
            codestr += tree.temp + " = " + tree.left.temp + " + " + tree.right.temp;
            emitCode( codestr );
        } else if( tree instanceof AssignExp ) {
            genCode( tree.rhs );
            tree.temp = tree.lhs.temp;
            codestr += tree.lhs.temp + " = " + tree.rhs.temp;
            emitCode( codestr );
        } else if( tree instanceof SimpleVar ) {
            // do nothing
        } else if( tree instanceof IntExp ) {
            // do nothing
        } else
            emitCode("Error");
    }
}
```

# Code Generation for Expressions

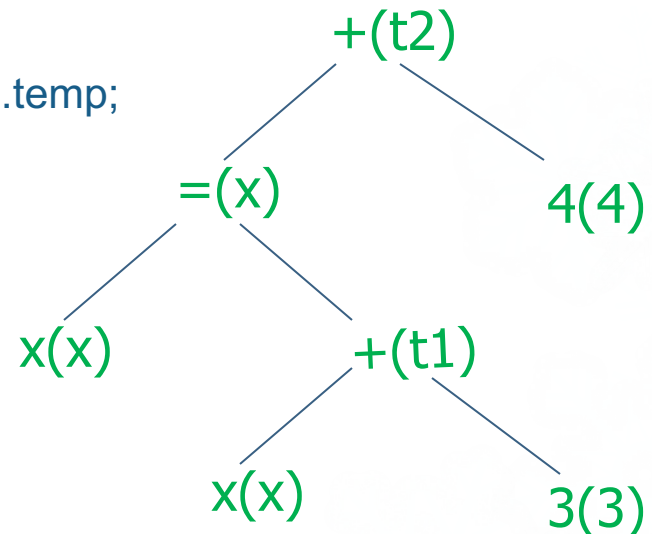
```
void genCode( Exp tree ) {      // newtemp() returns a new name such as t1, t2, etc.
    String codestr = "";
    if( tree != null ) {
        if( tree instanceof OpExp ) {
            genCode( tree.left );
            genCode( tree.right );
            tree.temp = newtemp();    // each node is added with a "temp" string
            codestr += tree.temp + " = " + tree.left.temp + " + " + tree.right.temp;
            emitCode( codestr );
        } else if( tree instanceof AssignExp ) {
            genCode( tree.rhs );
            tree.temp = tree.lhs.temp;
            codestr += tree.lhs.temp + " = " + tree.rhs.temp;
            emitCode( codestr );
        } else if( tree instanceof SimpleVar ) {
            // do nothing
        } else if( tree instanceof IntExp ) {
            // do nothing
        } else
            emitCode("Error");
    }
}
```

t1 = x + 3

x = t1

t2 = x + 4

e.g., (x = x + 3) + 4





# Array References

- Address vs. value of a variable:

$t1 = \&x + 10$

$*t1 = 2$

- Array references:

- address of  $a[i+1]$ :

$\&a + (i+1)*\text{elem\_size}(a)$

- fetch the value of an element:  $t2 = a[t1]$

$t = \&a + t1*\text{elem\_size}(a) \quad t2 = *t$

- assign to the address of an element:  $a[t2] = t1$

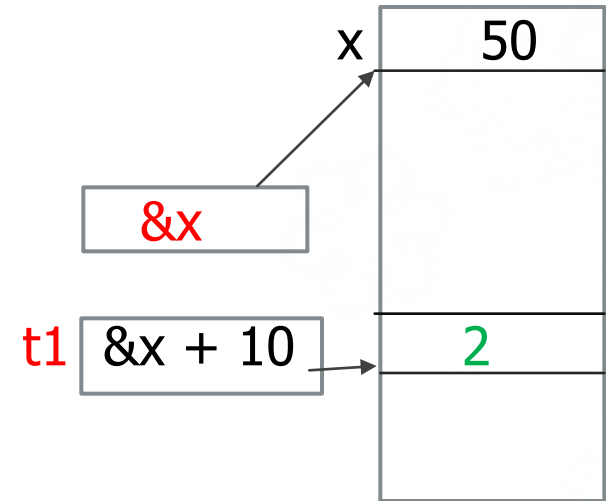
$t = \&a + t2*\text{elem\_size}(a) \quad *t = t1$

# Array References

➤ **Address** vs. **value** of a variable:

$t1 = \&x + 10$

$*t1 = 2$



➤ Array references:

- address of  $a[i+1]$ :

$\&a + (i+1)*\text{elem\_size}(a)$

- fetch the value of an element:  $t2 = a[t1]$

$t = \&a + t1*\text{elem\_size}(a)$        $t2 = *t$

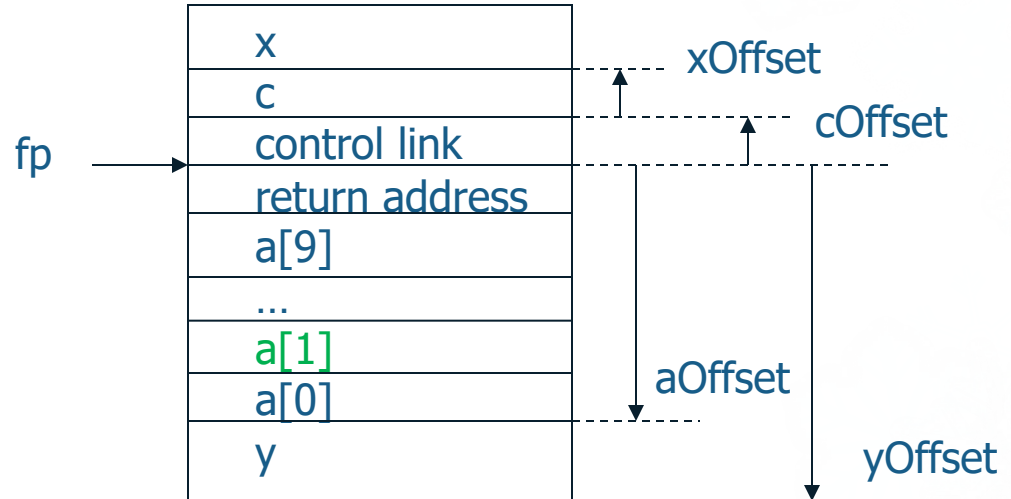
- assign to the address of an element:  $a[t2] = t1$

$t = \&a + t2*\text{elem\_size}(a)$        $*t = t1$

# Access to Names: LDA vs. LD

```
void f( int x, char c ) {  
    int a[10];  
    double y;  
    ...  
}
```

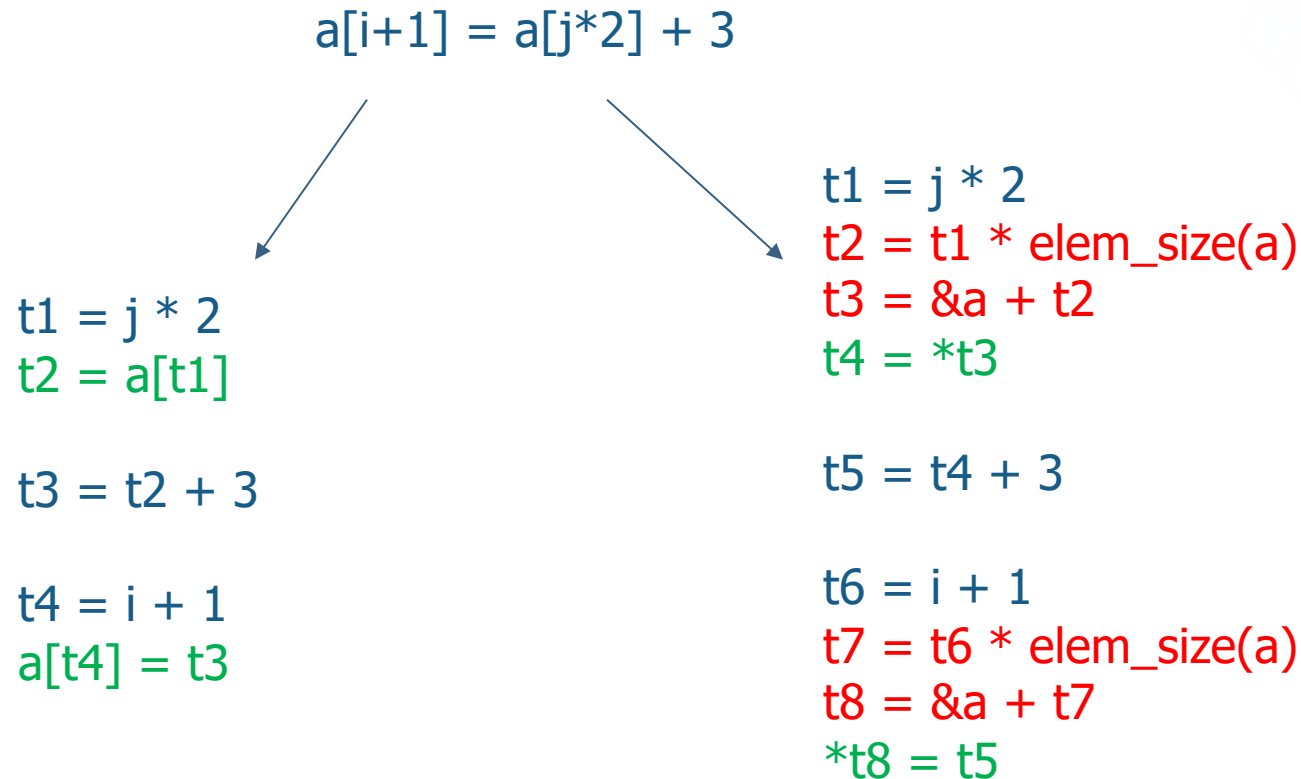
Name	Offset
x	+5
c	+4
a	-24
y	-32



$a[i]$  offset:  $(-24 + 2 * i)(fp)$

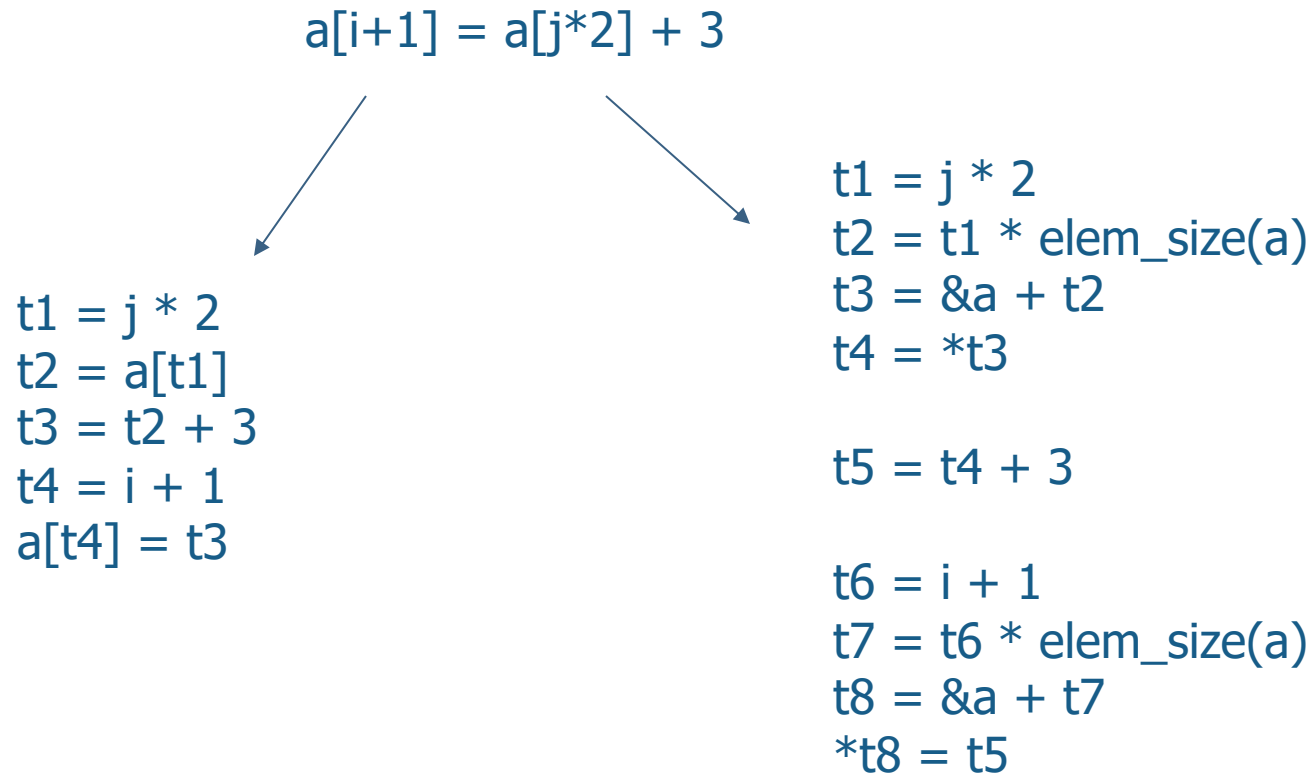
# Array References

➤ Different levels of details:



# Array References

- Different levels of details:



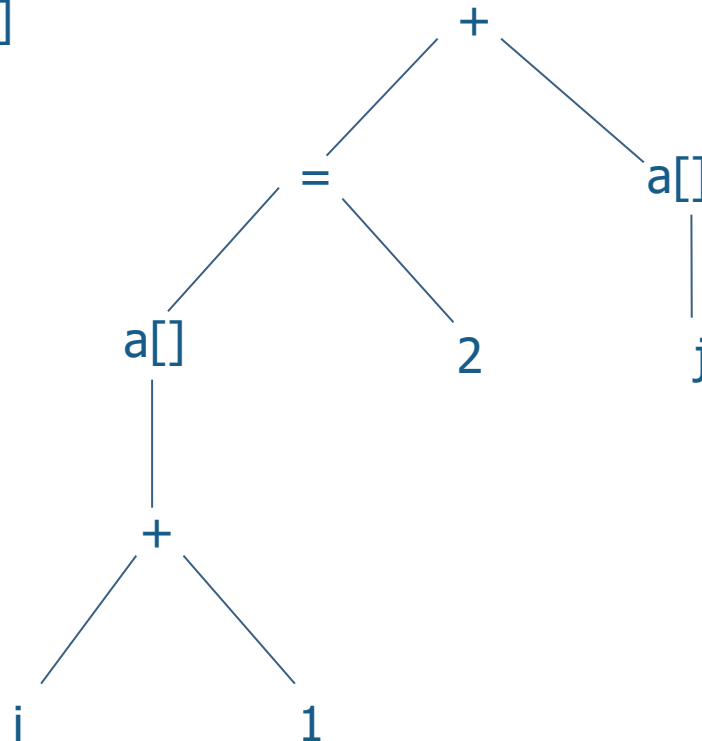
# Code Generation for Arrays

Grammar:     $\text{exp} \rightarrow \text{subs} = \text{exp} \mid \text{aexp}$   
               $\text{aexp} \rightarrow \text{aexp} + \text{factor} \mid \text{factor}$   
               $\text{factor} \rightarrow ( \text{exp} ) \mid \text{num} \mid \text{subs}$   
               $\text{subs} \rightarrow \text{id} \mid \text{id} [ \text{exp} ]$

e.g.,  $(a[i+1] = 2) + a[j]$

Intermediate code:

```
t1 = i + 1
t2 = t1 * elem_size(a)
t3 = &a + t2
*t3 = 2
t4 = j * elem_size(a)
t5 = &a + t4
t6 = *t3 + *t5
```



# Code Generation for Arrays

Grammar:     $\text{exp} \rightarrow \text{subs} = \text{exp} \mid \text{aexp}$   
               $\text{aexp} \rightarrow \text{aexp} + \text{factor} \mid \text{factor}$   
               $\text{factor} \rightarrow ( \text{exp} ) \mid \text{num} \mid \text{subs}$   
               $\text{subs} \rightarrow \text{id} \mid \text{id} [ \text{exp} ]$

e.g.,  $(a[i+1] = 2) + a[j]$

Intermediate code:

$t1 = i + 1$

$t2 = t1 * \text{elem\_size}(a)$

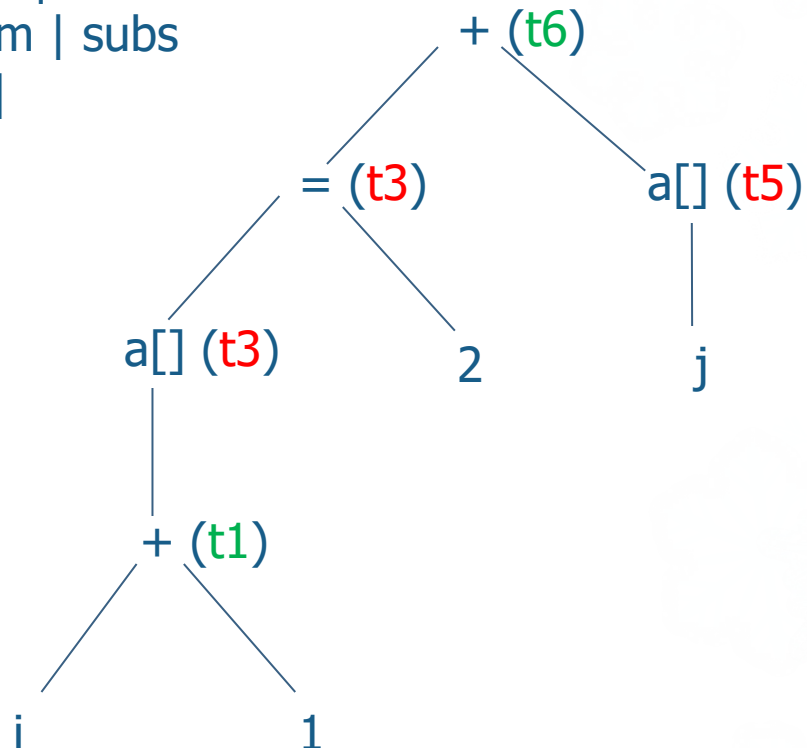
$t3 = \&a + t2$

$*t3 = 2$

$t4 = j * \text{elem\_size}(a)$

$t5 = \&a + t4$

$t6 = *t3 + *t5$



# Code Generation for Arrays

```
void genCode( Exp tree ) {  
    String codestr = "";  
    if( tree != null ) {  
        if( tree instanceof OpExp ) {  
            // refer to the related fragment  
        } else if( tree instanceof AssignExp ) {  
            // refer to the related fragment  
        } else if( tree instanceof IndexVar ) {  
            // refer to the related fragment  
        } else if( tree instanceof SimpleVar ) {  
            // do nothing  
        } else if( tree instanceof IntExp ) {  
            // do nothing  
        } else  
            emitCode( "Error" );  
    }  
}
```

```
// code fragment for IndexVar  
genCode( tree.index );  
String temp = newtemp();  
codestr += temp + " = " +  
    tree.index.temp + " * elem_size(" +  
    tree.name + ")";  
emitCode( codestr );  
String temp2 = newtemp();  
codestr += temp2 + " = &" +  
    tree.name + " + " + temp;  
emitCode( codestr );  
tree.temp = temp2;  
tree.isAddr = true;
```



# Code Generation for Arrays

```
// code fragment for OpExp
genCode( tree.left );
genCode( tree.right );
tree.temp = newtemp();
codestr += tree.temp + " = ";
if( tree.left.isAddr )
    codestr += "*" ;
codestr += tree.left.temp + " + ";
if( tree.right.isAddr )
    codestr += "*";
codestr += tree.right.temp;
emitCode( codestr );
```

```
// code fragment for AssignExp
genCode( tree.lhs );
genCode( tree.rhs );
tree.temp = tree.lhs.temp;
tree.isAddr = tree.lhs.isAddr;
if( tree.isAddr )
    codestr += "*" ;
codestr += tree.temp + " = ";
if( tree.rhs.isAddr )
    codestr += "*";
codestr += tree.rhs.temp;
emitCode( codestr );
```

# Code Generation for Arrays

```
void genCode( Exp tree ) {  
    String codestr = "";  
    if( tree != null ) {  
        if( tree instanceof OpExp ) {  
            // refer to the related fragment  
        } else if( tree instanceof AssignExp ) {  
            // refer to the related fragment  
        } else if( tree instanceof IndexVar ) {  
            // refer to the related fragment  
        } else if( tree instanceof SimpleVar ) {  
            // do nothing  
        } else if( tree instanceof IntExp ) {  
            // do nothing  
        } else  
            emitCode( "Error" );  
    }  
}
```

```
t1 = i + 1
```

```
t2 = t1 * elem_size(a)
```

```
t3 = &a + t2
```

// code fragment for IndexVar

```
genCode( tree.index );
```

```
String temp = newtemp();
```

```
codestr += temp + " = " +
```

```
    tree.index.temp + " * elem_size(" +  
    tree.name + ")";
```

```
emitCode( codestr );
```

```
String temp2 = newtemp();
```

```
codestr += temp2 + " = &" +
```

```
    tree.name + " + " + temp;
```

```
emitCode( codestr );
```

```
tree.temp = temp2;
```

```
tree.isAddr = true;
```

# Code Generation for Arrays

$t1 = i + 1$

$t6 = *t3 + *t5$

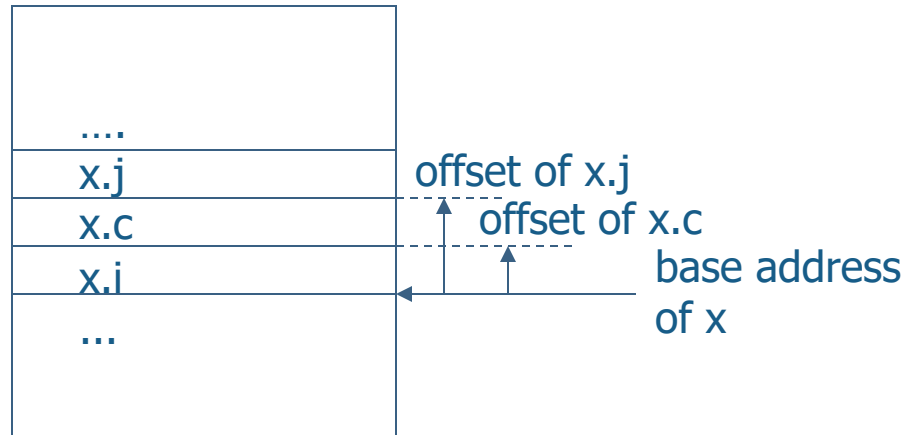
```
// code fragment for OpExp
genCode( tree.left );
genCode( tree.right );
tree.temp = newtemp();
codestr += tree.temp + " = ";
if( tree.left.isAddr )
    codestr += "*";
codestr += tree.left.temp + " + ";
if( tree.right.isAddr )
    codestr += "*";
codestr += tree.right.temp;
emitCode( codestr );
```

$*t3 = 2$

```
// code fragment for AssignExp
genCode( tree.lhs );
genCode( tree.rhs );
tree.temp = tree.lhs.temp;
tree.isAddr = tree.lhs.isAddr;
if( tree.isAddr )
    codestr += "*";
codestr += tree.temp + " = ";
if( tree.rhs.isAddr )
    codestr += "*";
codestr += tree.right.temp;
emitCode( codestr );
```

# Record References

```
typedef struct {  
    int i;  
    char c;  
    int j;  
} Record;  
...  
Record x;
```



e.g.,  $x.j = x.i$

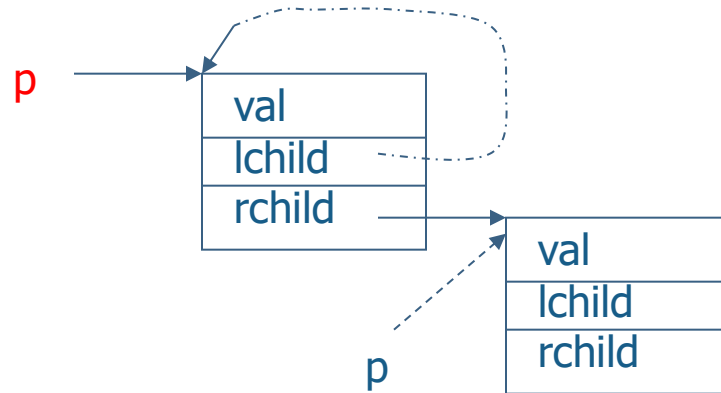
```
t1 = &x + field_offset(x, j)  
t2 = &x + field_offset(x, i)  
*t1 = *t2
```

# Pointer References

```
typedef struct treeNode {  
    int val;  
    struct treeNode *lchild, *rchild;  
} TreeNode;  
...  
TreeNode *p;
```

e.g., `p->lchild = p;`  
`p = p->rchild;`

```
t1 = p + field_offset(*p, lchild)  
*t1 = p;  
t2 = p + field_offset(*p, rchild)  
p = *t2
```

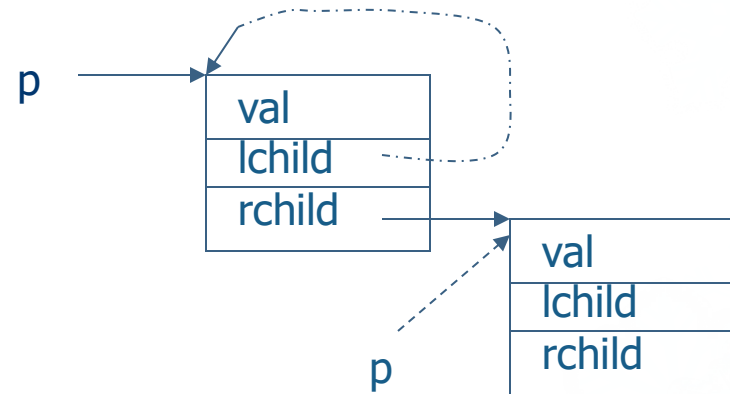


# Pointer References

```
typedef struct treeNode {  
    int val;  
    struct treeNode *lchild, *rchild;  
} TreeNode;  
...  
TreeNode *p;
```

e.g., `p->lchild = p; // (*p).lchild = p;`  
`p = p->rchild;`

```
t1 = p + field_offset(*p, lchild)  
*t1 = p;  
t2 = p + field_offset(*p, rchild)  
p = *t2
```

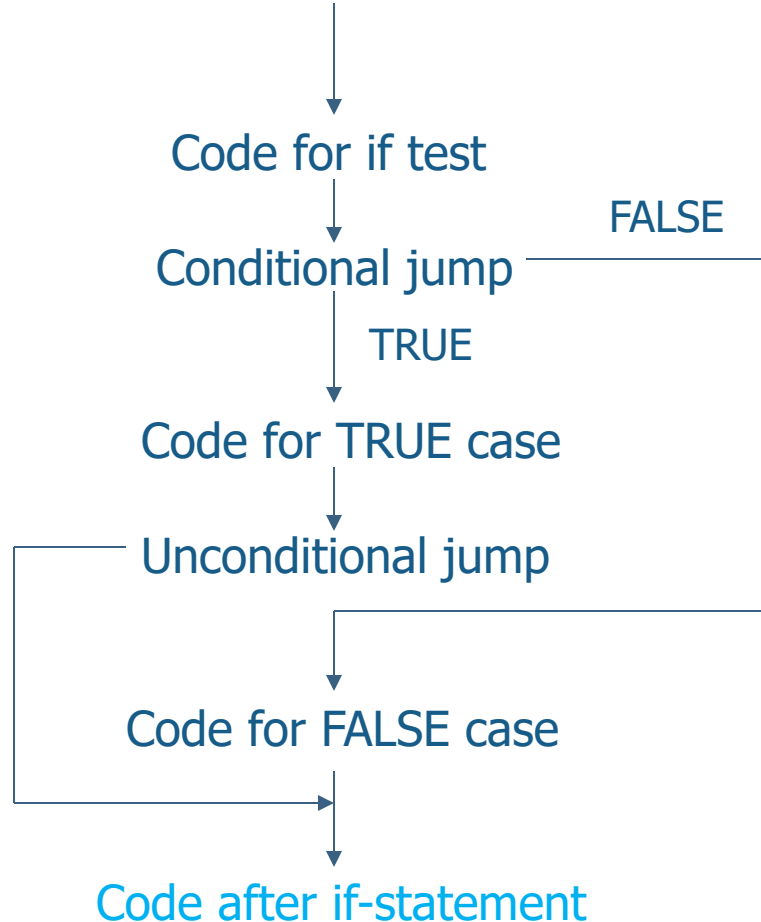


e.g., `x.j = x.i`

```
t1 = &x + field_offset(x, j)  
t2 = &x + field_offset(x, i)  
*t1 = *t2
```

# Code for If-statements

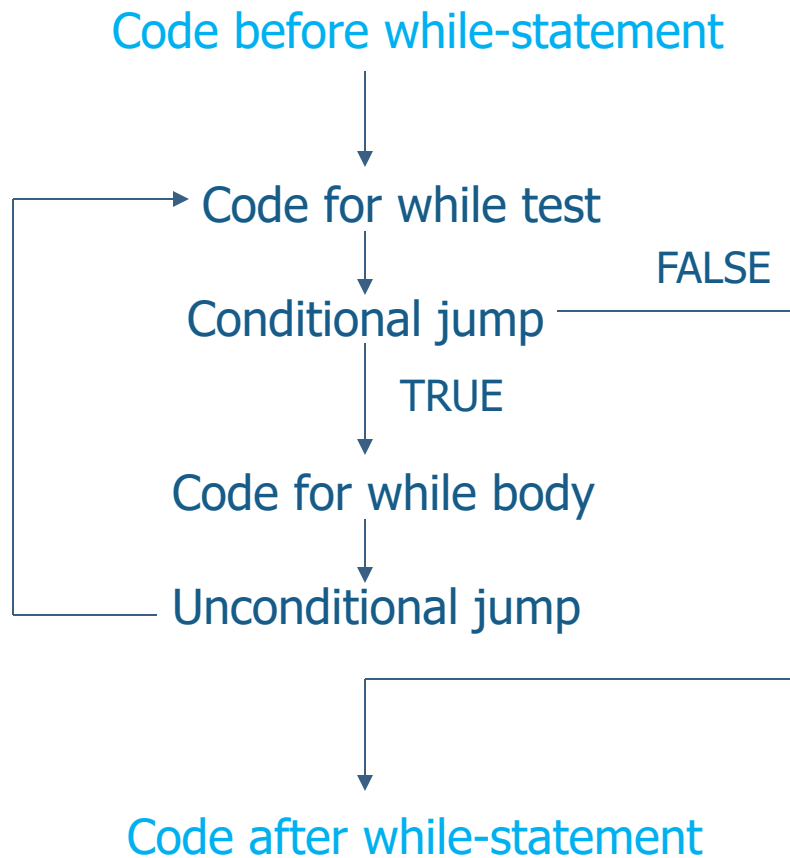
Code before if-statement



if (E) S1 else S2

```
<code to evaluate E to t1>  
if_false t1 goto L1  
<code for S1>  
goto L2  
label L1  
<code for S2>  
label L2
```

# Code for While-statements



while (E) S

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```



# Label Generation and Backpatching

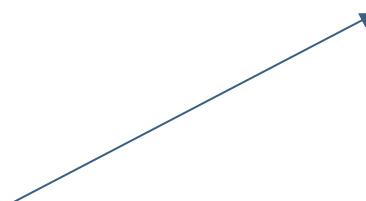
- Jumps to a label may need to be generated before the label definition
  - **Intermediate code**: generate a label for a forward jump and save it until the label location is known
  - **Executable code**: labels must be resolved to absolute or relative addresses
- Backpatching: leave a gap in the code for a forward jump or create a dummy jump to a fake location, and then go back to fix the location when the actual label is known
  - Keep the generated code in a buffer or a temporary file

# Code for Logical Expressions

- Short circuit:
  - If a is false then (a and b) is also false
  - If a is true then (a or b) is also true
- If-expressions: equivalent to if-statements except that they return values
  - a and b  $\equiv$  if a then b else false
  - a or b  $\equiv$  if a then true else b

`(x != 0) && (y == x)`

`if( x != 0 ) then (y == x)  
else false`



```
t1 = (x != 0)  
if_false t1 goto L1  
t2 = (y == x)  
goto L2  
label L1  
t2 = FALSE  
label L2
```

# Code Generation for Control Stmts

stmt -> if-stmt | while-stmt | **break** | **other**  
if-stmt -> **if** ( exp ) stmt | **if** ( exp ) stmt **else** stmt  
while-stmt -> **while** ( exp ) stmt  
exp -> **true** | **false**

if\_false true goto L1

e.g., if ( true ) while ( true ) if ( false ) break else other

label L2

if\_false true goto L3

if\_false false goto L4

goto L3

goto L5

label L4

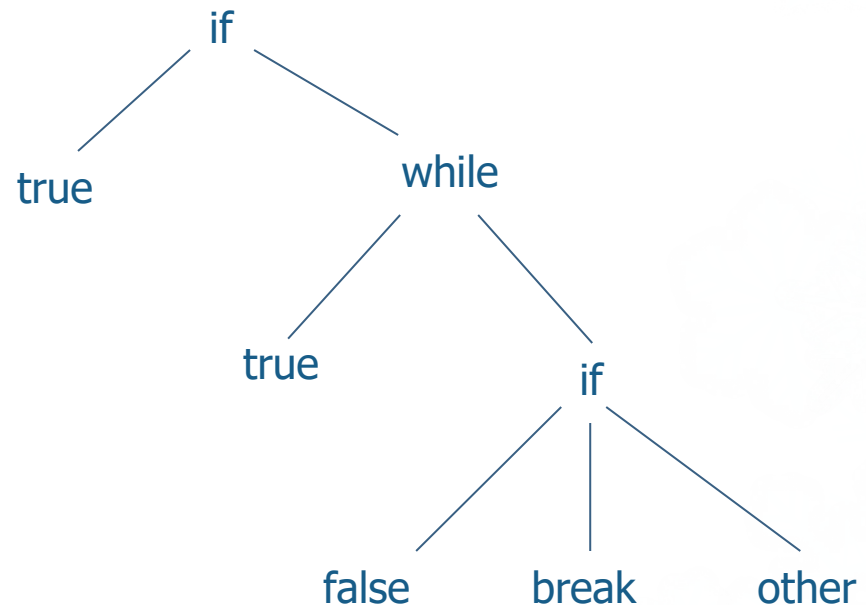
other

label L5

goto L2

label L3

label L1



# Code Generation for Control Stmts

```
void genCode( Exp tree, String label ) {  
    String codestr = "";  
    String lab1, lab2;  
    if( tree != null ) {  
        if( tree instanceof IntExp ) {  
            // do nothing  
        } else if( tree instanceof IfExp ) {  
            // refer to the related fragment  
        } else if( tree instanceof WhileExp ) {  
            // refer to the related fragment  
        } else if( tree instanceof BreakExp ) {  
            codestr += "goto " + label;  
            emitCode( codestr );  
        } else if( tree instanceof OtherExp ) {  
            emitCode( "Other" );  
        } else  
            emitCode("Error");  
    }  
}
```

```
// code fragment for WhileExp  
lab1 = genLabel();  
codestr += "label" + lab1;  
emitCode( codestr );  
genCode( tree.test, label );  
lab2 = genLabel();  
if( tree.test.value == 0 )  
    codestr += "if_false false goto " + lab2;  
else  
    codestr += "if_false true goto " + lab2;  
emitCode( codestr );  
genCode( tree.body, lab2 );  
codestr += "goto " + lab1;  
emitCode( codestr );  
codestr += "label " + lab2;  
emitCode( codestr );
```

# Code Generation for Control Stmts

```
// code fragment for IfExp
genCode( tree.test, label );
lab1 = genLabel();
if( tree.test.value == 0 )
    codestr += "if_false false goto " + lab1;
else
    codestr += "if_false true goto " + lab1;
emitCode( codestr );
genCode( tree.then, label );
if( tree.else != null ) {
    lab2 = genLabel();
    codestr += "goto " + lab2;
    emitCode( codestr );
}
```

```
// continued from left
codestr += "label " + lab1;
emitCode( codestr );
if( tree.else != null ) {
    genCode( tree.else, label );
    codestr += "label " + lab2;
    emitCode( codestr );
}
```

# Code Generation for Control Stmts

```
stmt -> if-stmt | while-stmt | break | other  
if-stmt -> if ( exp ) stmt | if ( exp ) stmt else stmt  
while-stmt -> while ( exp ) stmt  
exp -> true | false
```

if\_false true goto L1

e.g., if ( true ) while ( true ) if ( false ) break else other

label L2

if\_false true goto L3

if\_false false goto L4

goto L3

goto L5

label L4

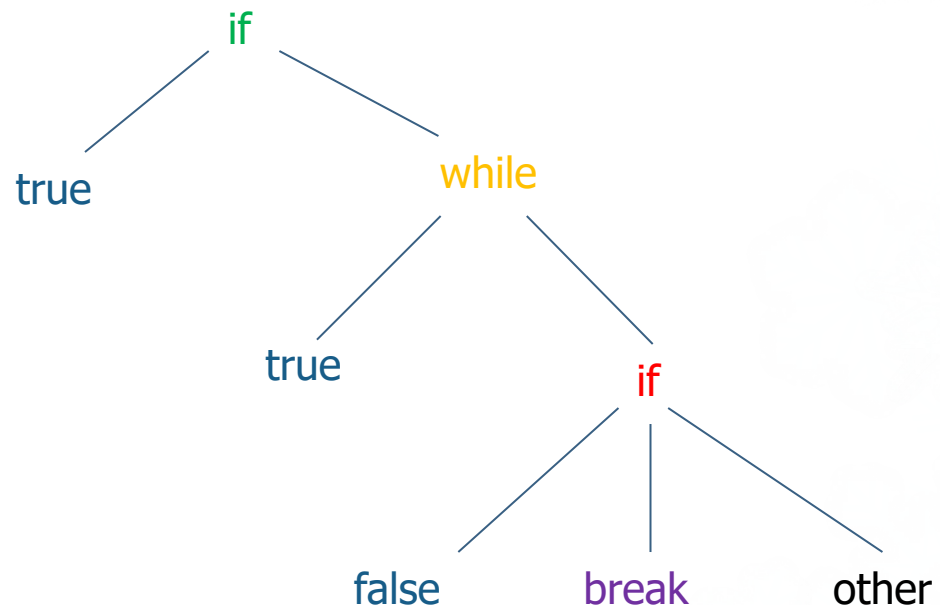
other

label L5

goto L2

label L3

label L1



# Code Generation for Control Stmts

```
void genCode( Exp tree, String label ) {
    String codestr = "";
    String lab1, lab2;
    if( tree != null ) {
        if( tree instanceof IntExp ) {
            // do nothing
        } else if( tree instanceof IfExp ) {
            // refer to the related fragment
        } else if( tree instanceof WhileExp ) {
            // refer to the related fragment
        } else if( tree instanceof BreakExp ) {
            codestr += "goto " + label;
            emitCode( codestr );
        } else if( tree instanceof OtherExp ) {
            emitCode( "Other" );
        } else
            emitCode("Error");
    }
}
```

// code fragment for WhileExp

```
lab1 = genLabel();
codestr += "label" + lab1;
emitCode( codestr );

genCode( tree.test, label );
lab2 = genLabel();
if( tree.test.value == 0 )
    codestr += "if_false false goto " + lab2;
else
    codestr += "if_false true goto " + lab2;
emitCode( codestr );

genCode( tree.body, lab2 );

codestr += "goto " + lab1;
emitCode( codestr );

codestr += "label " + lab2;
emitCode( codestr );
```

# Code Generation for Control Stmts

```
// code fragment for IfExp
genCode( tree.test, label );
lab1 = genLabel();
if( tree.test.value == 0 )
    codestr += "if_false false goto " + lab1;
else
    codestr += "if_false true goto " + lab1;
emitCode( codestr );

genCode( tree.then, label );

if( tree.else != null ) {
    lab2 = genLabel();
    codestr += "goto " + lab2;
    emitCode( codestr );
}
```

```
// continued from left
codestr += "label " + lab1;
emitCode( codestr );

if( tree.else != null ) {
    genCode( tree.else, label );

    codestr += "label " + lab2;
    emitCode( codestr );
}
```



# Function Definitions and Calls

- Function definition: create a function name, parameters, the return type, and the code
- Function call: create actual values for parameters (called arguments), perform a jump to the function code, and return to the caller
- The runtime environment is not known at the definition time, but the general record structure is clear
  - The runtime environment is built by the calling sequence (partially by the caller and partially by the callee)

# Intermediate Code for Functions

e.g., function definition:

```
int f( int x, int y ) {  
    return x + y + 1;  
}
```

Three-address code:

```
entry f  
t1 = x + y  
t2 = t1 + 1  
return t2
```

e.g., function call:

```
x = f( 2 + 3, 4 );
```

Three-address code:

```
begin_args  
t1 = 2 + 3  
arg t1  
arg 4  
x = call f
```

# Code Generation for Functions

fn f(x) = 2 + x  
fn g(x, y) = f(x) + y  
g(3, 4)

entry f  
t1 = 2 + x  
return t1

entry g  
begin\_args  
arg x  
t2 = call f  
t3 = t2 + y  
return t3

begin\_args  
arg 3  
arg 4  
call g

