CSc 553
Principles of Compilation

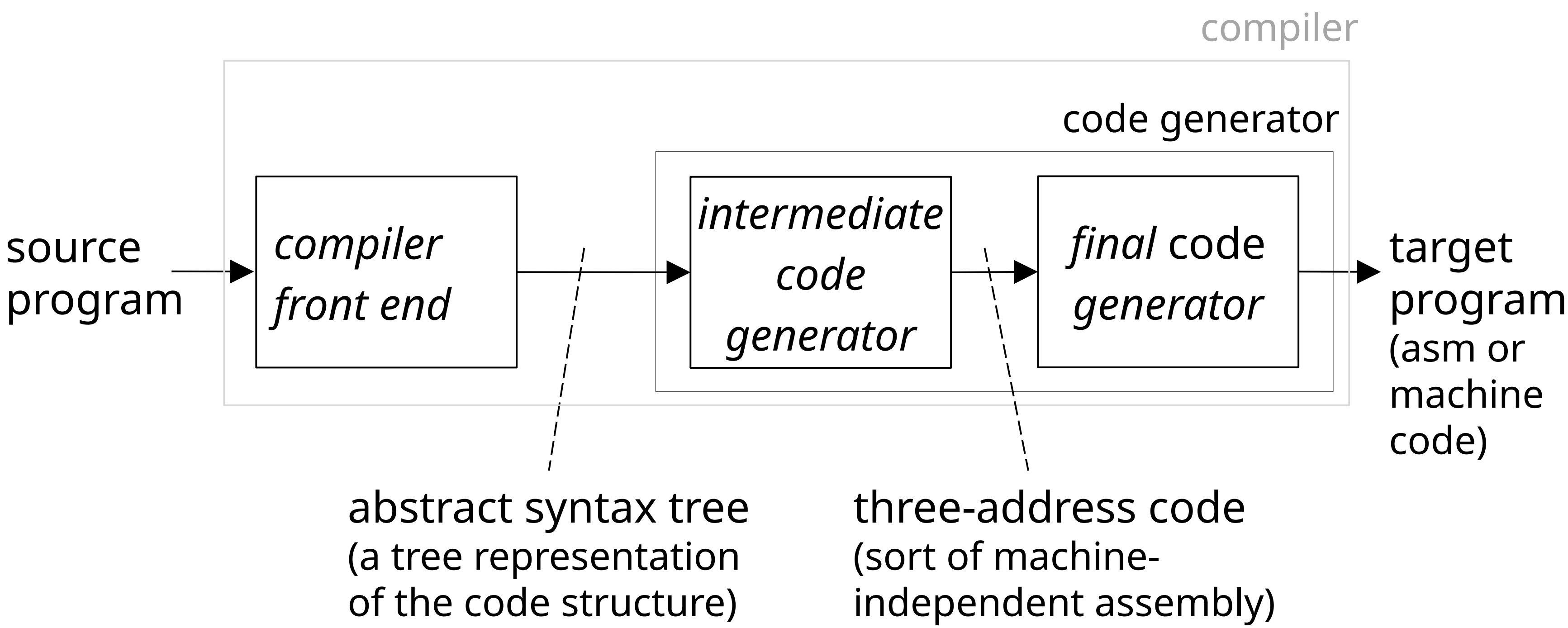
04. Code Generation

Saumya Debray

The University of Arizona

Tucson, AZ 85721

Overview


compiler
code generator
source
program
compiler
front end
intermediate
code
generator
final code
generator
target
program
(asm or
machine
code)
abstract syntax tree
(a tree representation
of the code structure)
three-address code
(sort of machine-
independent assembly)

Overview

Approach

Processing a function definition:

1

Intermediate Code Generation

- generate 3-addr code by recursively traversing syntax tree
- each AST node has a list of 3-addr instrs attached to it
- list of instructions at the root of the tree is the code for the entire function

2

Storage Allocation

- traverse the local symbol table
- allocate a stack frame slot for each local and temporary
- type info determines how much space to allocate

3

Final Code Generation

- traverse the list of 3-addr instrs at the root of the function's AST
- expand each 3-addr instruction into asm code
- write out the code generated

Overview: Intermediate Code Generation

1 Intermediate Code Generation

2 Storage Allocation

3 Final Code Generation

- Much of the work of code generation is here
- To be discussed soon

Overview: Storage Allocation

1 Intermediate Code Generation

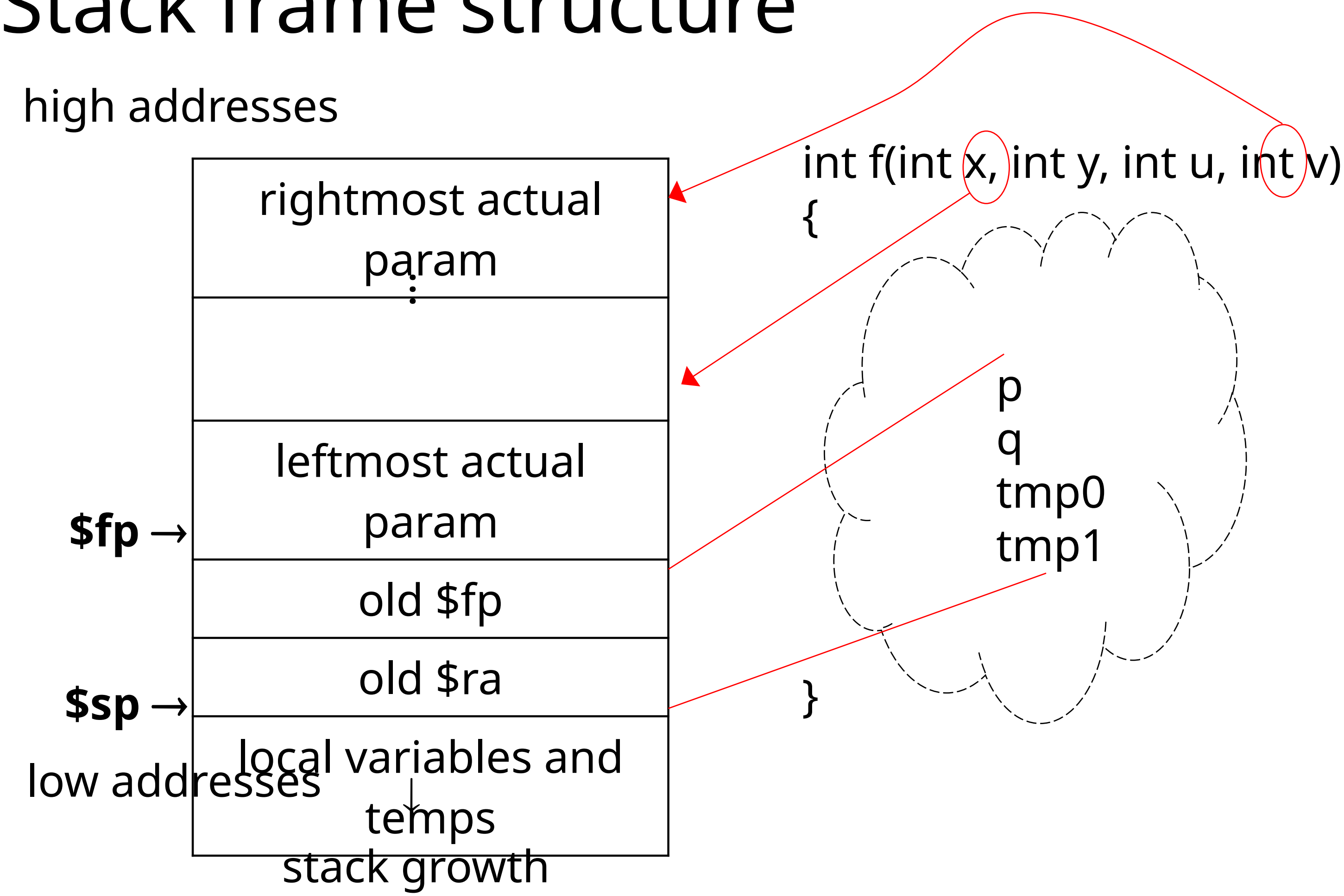
2 Storage Allocation

3 Final Code Generation

- Traverse the function's local symbol table
- Allocate storage location to each ST entry based on its type
  - allocated as slots in the function's stack frame
  - respect any alignment restrictions imposed by target machine architecture
- Formal parameter locations are fixed by argument position

Storage Allocation: Example

Stack frame structure



var	location
x	8($fp)
y	12($fp)
u	16($fp)
v	20($fp)
p	−4($fp)
q	−8($fp)
tmp0	−12($fp)
tmp1	−16($fp)

Overview: Final Code Generation

1 Intermediate Code Generation

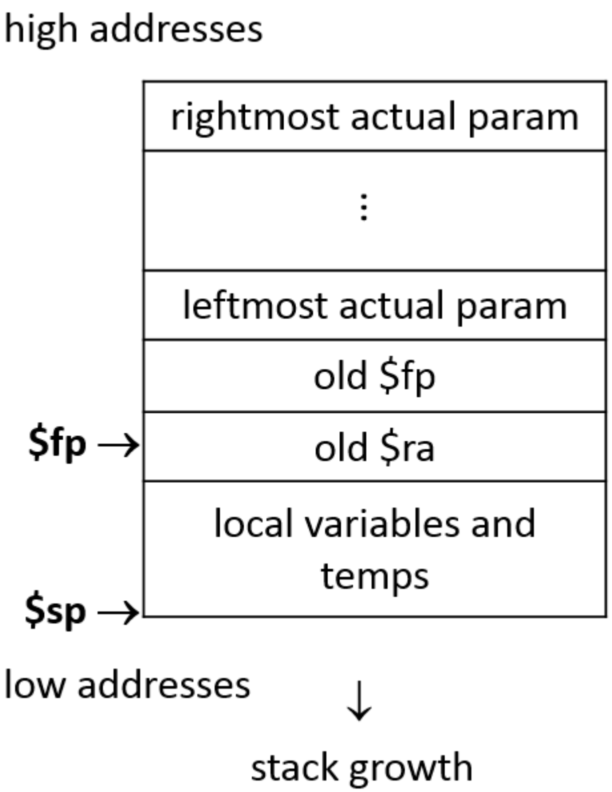
2 Storage Allocation

3 Final Code Generation

Expand each 3-addr instruction to asm

3-address code	*MIPS assembly code*
`x = y + z`	load **y** into *reg1* load **z** into *reg2* **add** *reg3, reg1, reg2* **sw** *reg3*, **x**
`if x ≥ y goto L`	load **x** into *reg1* load **y** into *reg2* **bge** *reg1, reg2*, **L**

EXERCISE



Source code

```
int fact(int n) {
    int p = 1;
    while (n > 0) {
        p *= n;
        n -= 1;
    }
    return p;
}
```

3-address code

```
        enter fact
        p = 1
L0:     if n <= 0 goto L1
        tmp0 = p * n
        p = tmp0
        tmp1 = n – 1
        n = tmp1
        goto L0
L1:     leave
        return p
```

var	p	n	tmp0	tmp1
location	??	??	??	??

Intermediate Code Generation

Approach

1. identify a program construct
(from source code or syntax tree)

2. understand the runtime actions needed

3. figure out the code to do those runtime actions

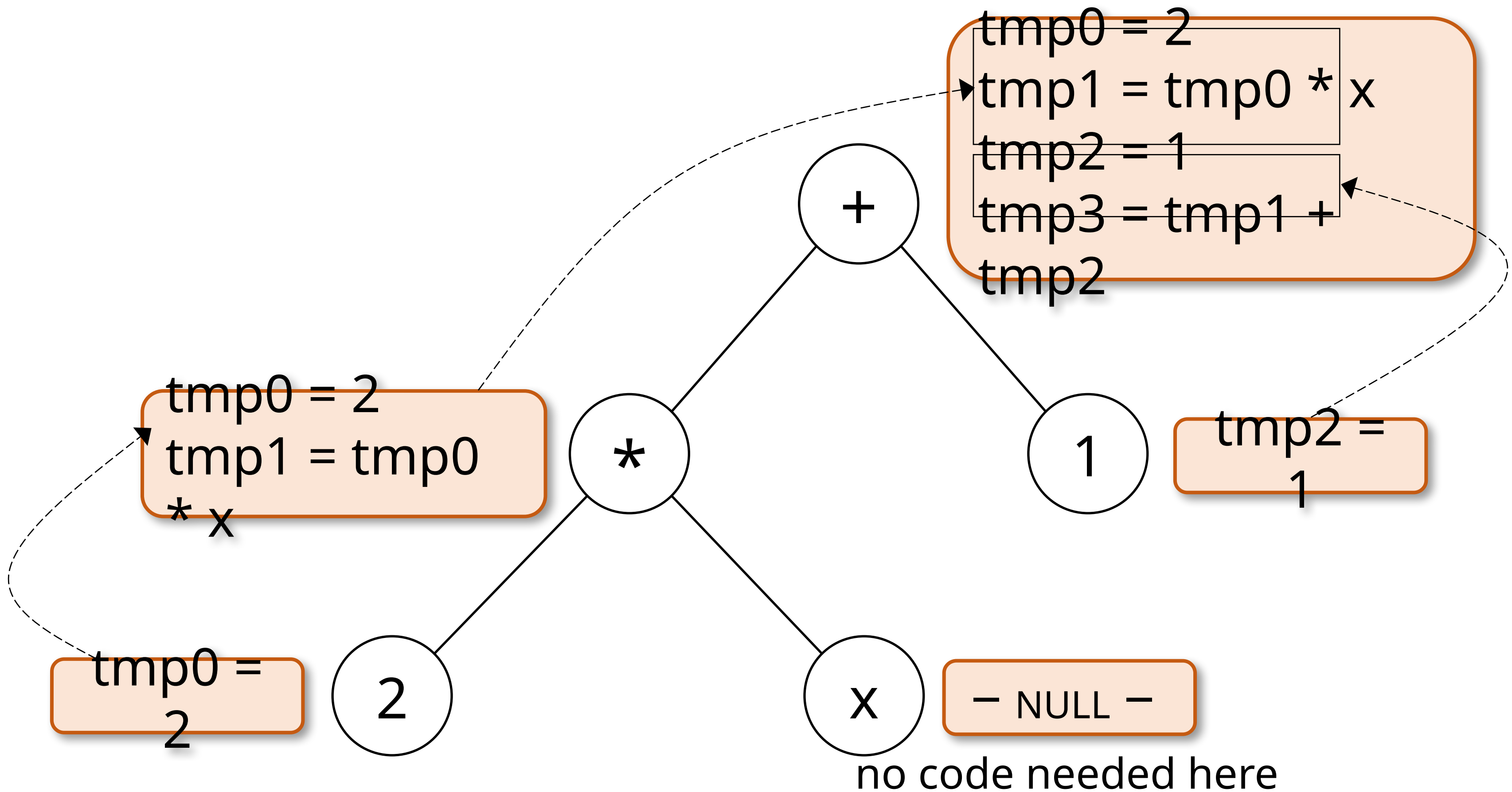
4. figure out what the code generator needs to do

Approach

Recursively traverse the syntax tree:

- Node type determines action at each node
- Code for each node is a list of three-address instructions
- Generate code for each node after processing its children
    - at each syntax tree node, create a list of instructions that executes the computation for the syntax tree rooted at the node
    - glue together the instructions for its children, plus code specific to that node

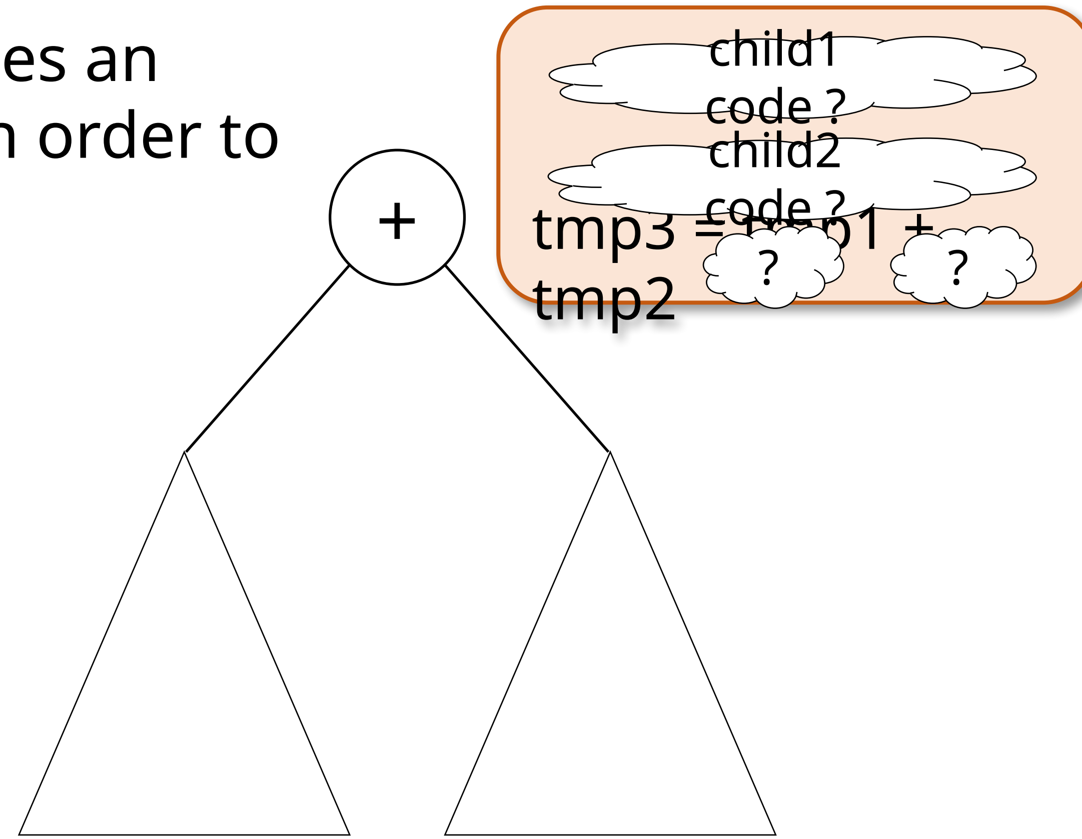
Example



list of 3-addr instructions at each node

Info needed for code generation

What information does an internal node need in order to generate code?

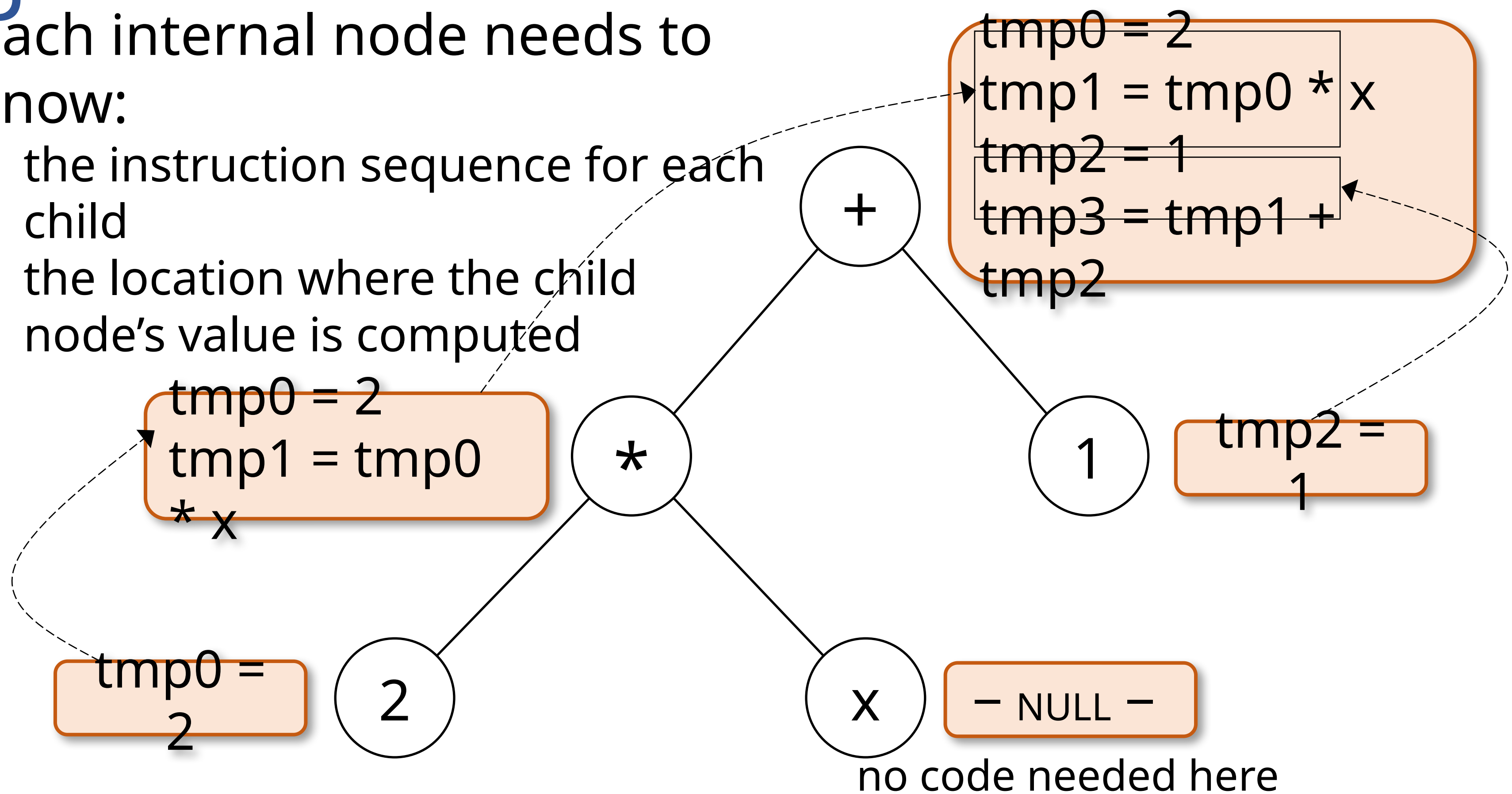


list of 3-addr instructions at each node

Info needed for code generation

Each internal node needs to know:

- the instruction sequence for each child
- the location where the child node's value is computed



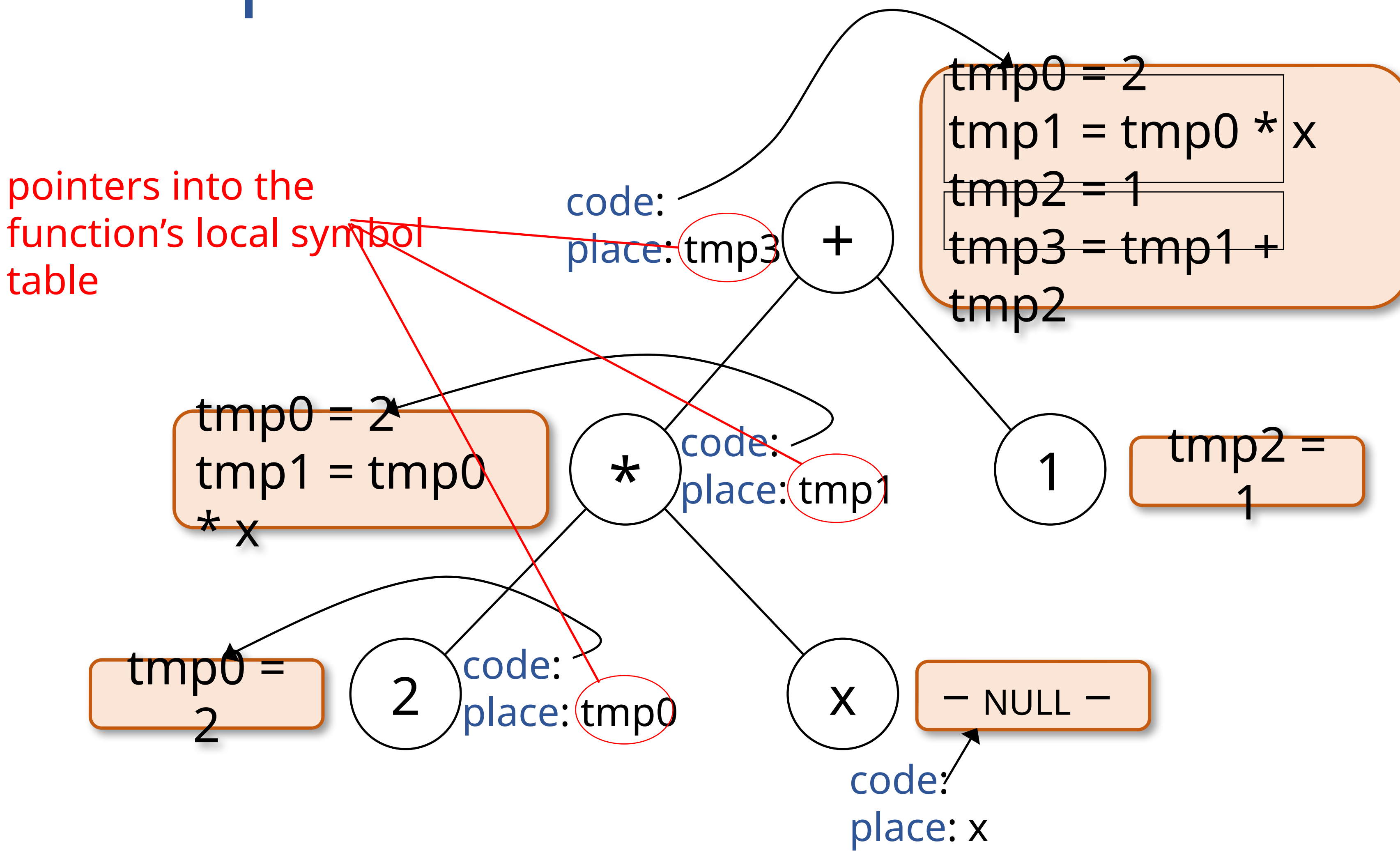
list of 3-addr instructions at each node

Info needed for code generation

We augment syntax tree nodes with the following fields:

- **code**: a list of intermediate code instructions for executing that node and its children
- the address/value field [expressions only]: one of
  - **place**: the location where the expression's value is stored at runtime
  - **loc**: the address of the location where the expression's value will be stored at runtime (used for array elements)
- the place/loc field refers to a symbol table entry for a variable or temporary
  - the variable/temporary is mapped to an actual memory location when going to final code
  - **place** vs. **loc** indicates whether or not to treat it as a pointer

Example


pointers into the function's local symbol table
code:
place: tmp3
+
tmp0 = 2
tmp1 = tmp0 * x
tmp2 = 1
tmp3 = tmp1 + tmp2
*
code:
place: tmp1
tmp0 = 2
tmp1 = tmp0 * x
1
tmp2 = 1
2
code:
place: tmp0
tmp0 = 2
x
– NULL –
code:
place: x

Approach

```
codeGen_stmt(synTree_node S)
{
    switch (S.nodetype) {
        case FOR:        ... ; break;
        case WHILE :     ... ; break;
        case IF:         ... ; break;
        case '=' :       ... ; break;
    ...
}
```

```
codeGen_expr(synTree_node E)
{
    switch (E.nodetype) {
        case '+':     ... ; break;
        case '*' :    ... ; break;
        case '–':     ... ; break;
        case '/' :    ... ; break;
    ...
}
```

At each syntax tree node:

- *recursively process the children*
- *then generate code for this node*
- *then glue it all together*

Auxiliary routines

- struct symtab_entry *newtemp(typename t)
  - *create a symbol table entry for a new temporary*
  - *return a pointer to this ST entry.*
- struct instr *newlabel()
  - *return a new label instruction*
- struct instr *newinstr(op, arg_1, arg_2, ...)
  - *create a new instruction, fill in the arguments supplied*
  - *return a pointer to the result*

Auxiliary routines: newtemp()

```
struct symtab_entry *newtemp( t )
{
        struct symtab_entry *ntmp = malloc( … );
        ntmp->name  =  …create a new name that doesn’t conflict…
        ntmp->type = t;
        …insert ntmp into the function's local symbol table…
        return ntmp;
}
```

Auxiliary routines: newinstr()

```
struct instr *newinstr(opType, src1, src2, dest)
{
        struct instr *ninstr = malloc( … );
        ninstr->op = opType;
        ninstr->src1 = src1; ninstr->src2 = src2; ninstr->dest = dest;
        return ninstr;
}
```

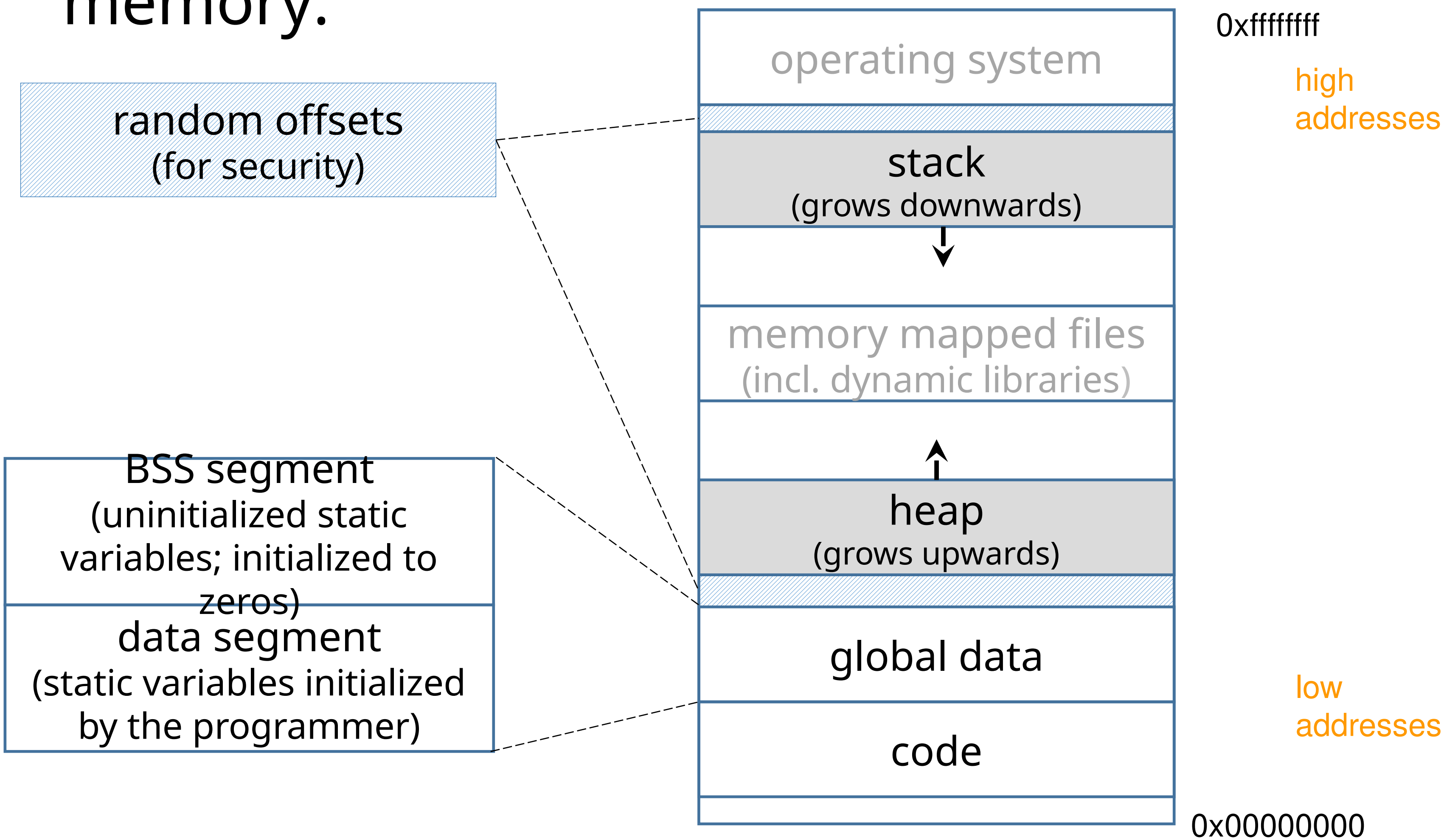
Auxiliary routines: newlabel()

```
static int label_num = 0;
struct instr *newlabel()
{
        return  newinstr(LABEL, label_num++);
}
```

Understanding what happens at runtime

Runtime Memory Organization (Linux)

Layout of an executing process's virtual memory:



Activation Records

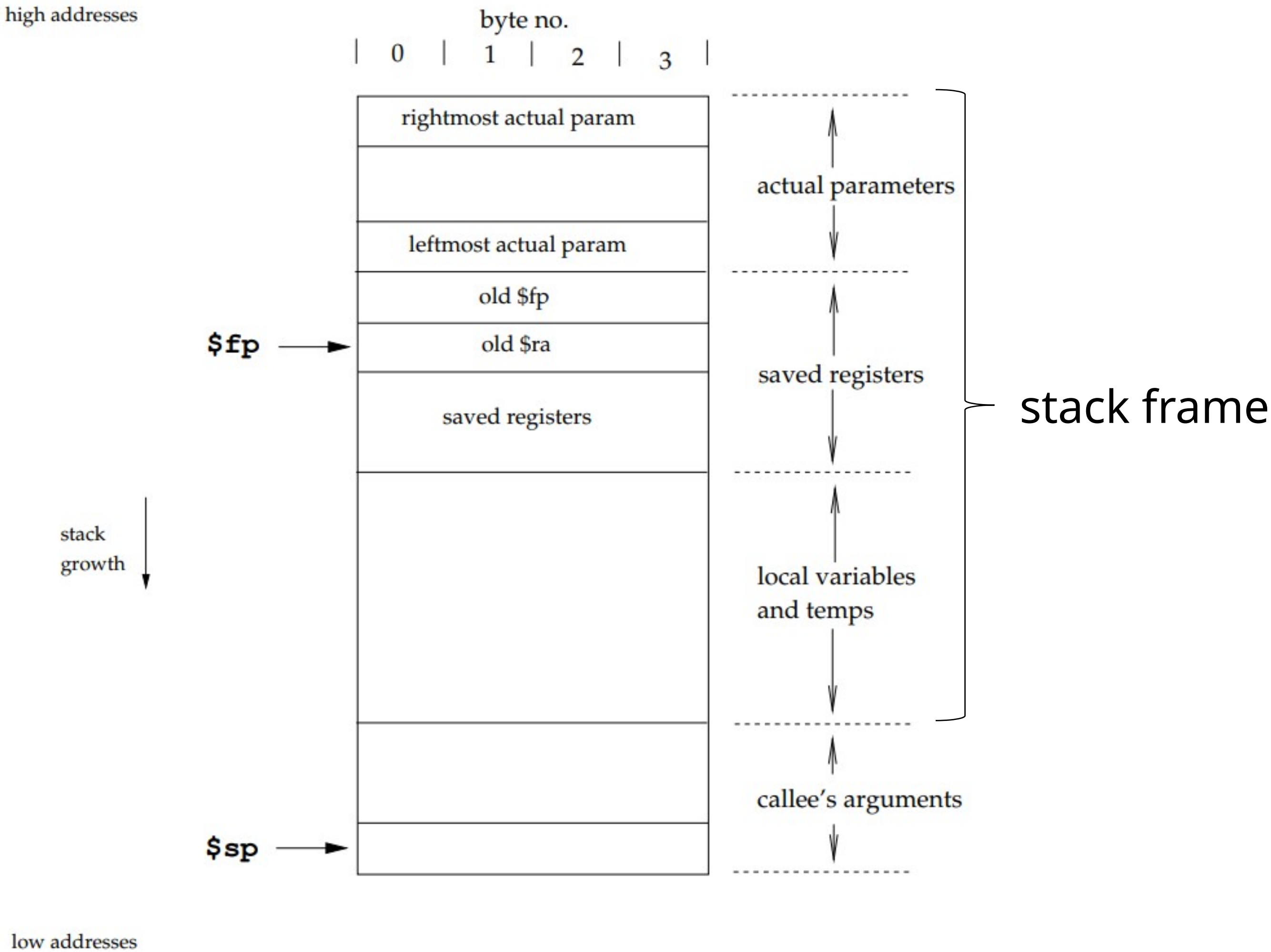
- An *activation record* contains information needed to manage a single activation of a procedure, e.g.:
  - saved machine state (PC, registers, return address);
  - actual parameter values;
  - local and temporary variables.
- The contents of an activation record may be spread across the stack frame and registers.

Activation Records: Layout

- Some aspects of activation record layout are specified by the *calling convention*. E.g.:
  - location of actual parameters
  - some machine state info

- The compiler decides the layout for local variables and temporaries:
  - the amount of storage needed for an object is determined by its type;
  - storage layout must conform to any *alignment restrictions* of the underlying architecture.

Example: Stack frame for project



Procedure Calls and Returns

- *Calling sequence*: handles a call to a procedure:
  - loads actual parameters where callee can find them;
  - saves machine state (return address, …);
  - branches to callee;
  - allocates an activation record.
- *Return sequence*: handles the return from a procedure call:
  - loads the return value where the caller can find it;
  - deallocates the activation record;
  - restores machine state (saved registers, PC, etc.);
  - branches back to caller.

Calling Conventions

- A *calling convention* for an architecture and/or language specifies how values are communicated between procedures:
  - register usage (e.g., caller-saved vs. callee-saved registers)
  - argument and return value placement

  E.g.: on the x86 [C calling convention]: an integer return value is placed in register **eax**.

- We can have multiple calling conventions, e.g.: __cdecl, __stdcall, __fastcall in MS Windows.

Caller-saved vs. Callee-saved Registers

- A calling convention typically divides registers into two classes:
  - *caller-saved*: registers whose values may be overwritten by a function call
    - used, e.g., for scratch values
  - *callee-saved*: registers whose values will survive across a function call
    - used, e.g., for values inside loops that contain function calls
- A function using a callee-saved register must save it on entry and restore it on exit

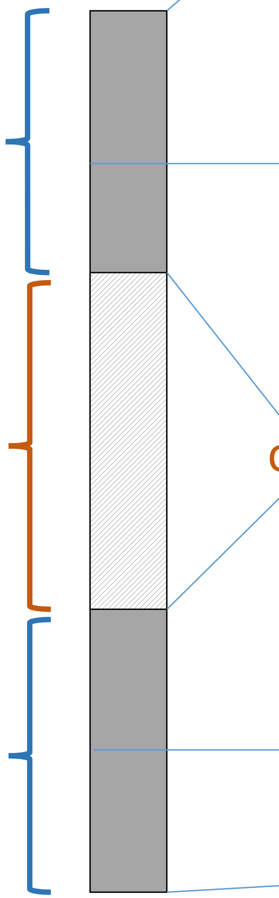
Code executed for a function call

calling sequence

- set up function arguments
- record machine state (e.g., return address)
- jump to callee

- allocate stack frame
- save machine state (e.g., return address, old frame ptr, callee-saved regs)

function body



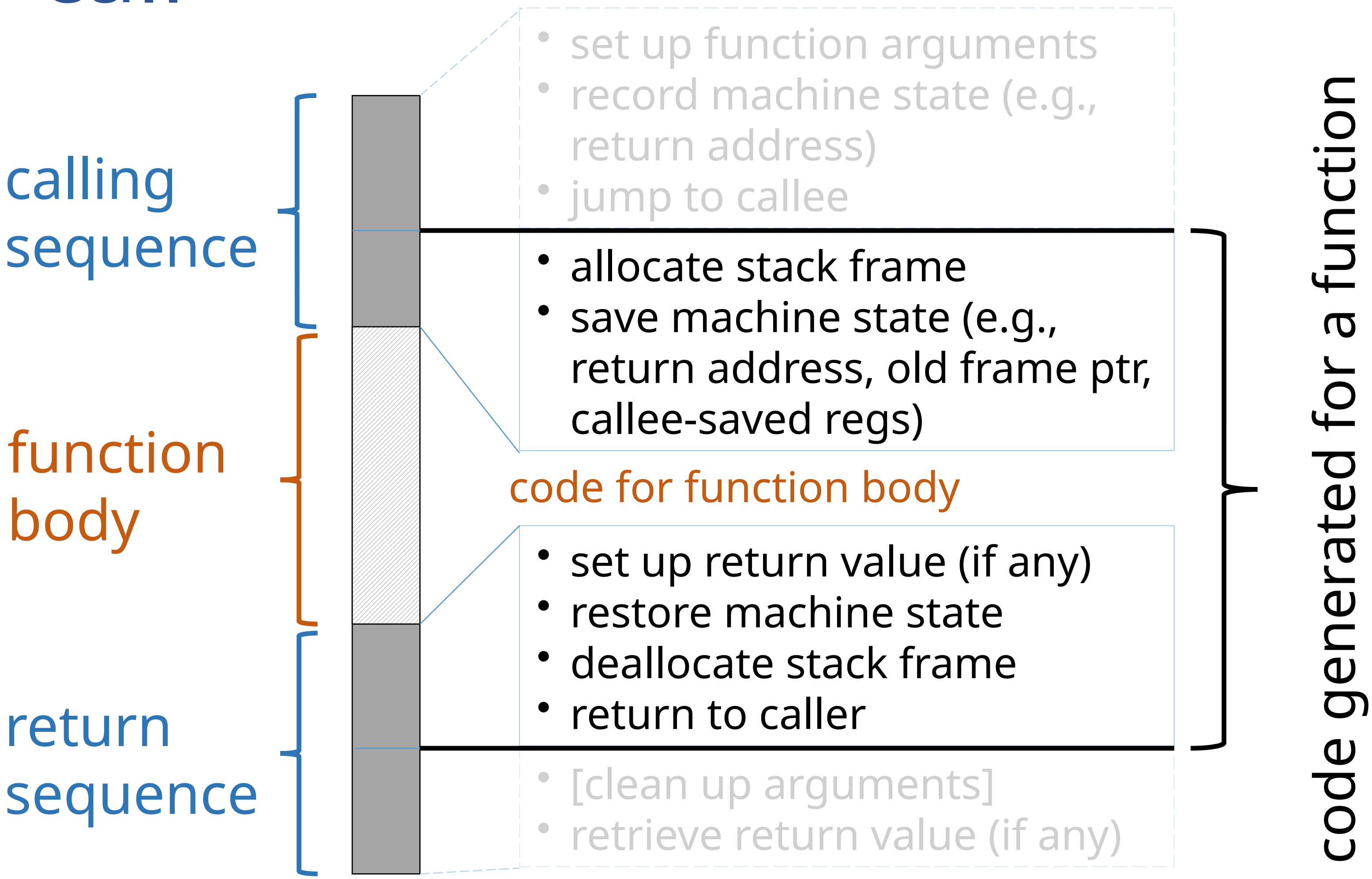
code for function body

return sequence

- set up return value (if any)
- restore machine state
- deallocate stack frame
- return to caller

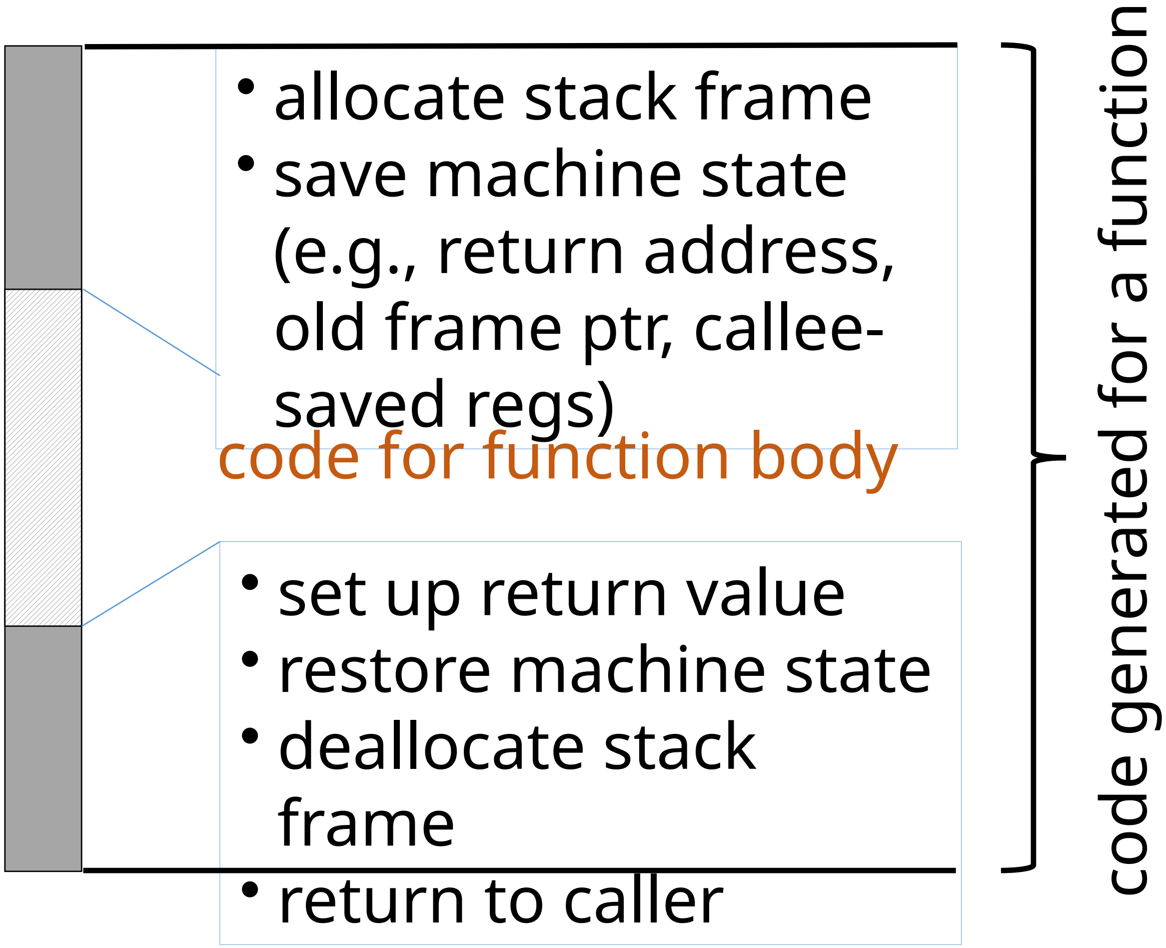
- [clean up arguments]
- retrieve return value (if any)

Code executed for a function call

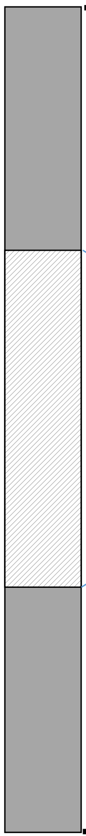


Code generation for function definitions

Code generated for function definitions



Code generated for function definitions



Three-address code

- allocate stack frame
- save machine state (e.g., return address, old frame ptr, callee-saved regs)

enter f

// f: pointer to the function's

// symbol table entry

code for function body

generated using codeGen_stmt()
[to be discussed soon]

- set up return value
- restore machine state
- deallocate stack frame
- return to caller

leave f

// f: pointer to the function's

// symbol table entry

return

Examples

Source code

```
int add3(x,y,z) {
    return x+y+z;
}
```

```
int max(x,y) {
    if(x > y)
        return x;
    else
        return y;
}
```

Three-address code

enter add3
  ... code for body of add3()...
leave add3
return tmp

enter max
  ... code for body of max()...
leave max
return x
  ... code for body of max()...
leave max
return y

pointers to symbol table entries

EXERCISE

What's wrong with this code?

Source Code	3-addr Code

int x;
void init_x(y)
{
    x = y;
}
	enter init_x
	x = y

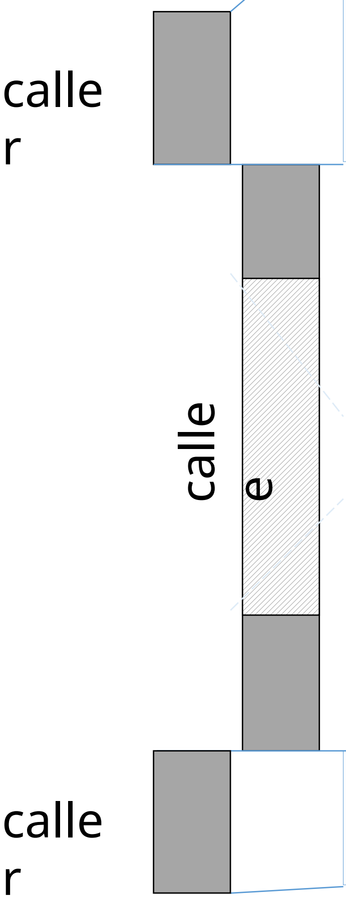
EXERCISE

What 3-address code should the compiler generate?

Source Code	3-addr Code
void myfun(void) { }	?

Code generation for function calls

Code executed for a function call



- set up function arguments
- record machine state (e.g., return address)
- jump to callee

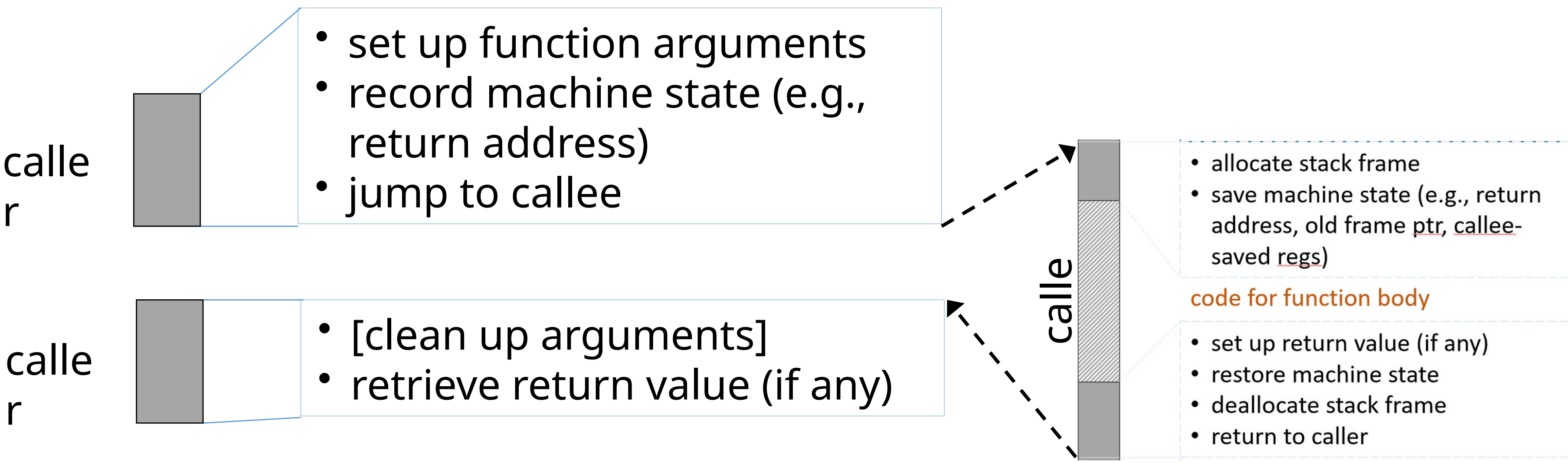
- allocate stack frame
- save machine state (e.g., return address, old frame ptr, callee-saved regs)

code for function body

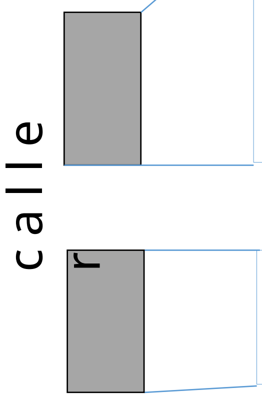
- set up return value (if any)
- restore machine state
- deallocate stack frame
- return to caller

- [clean up arguments]
- retrieve return value (if any)

Code executed for a function call



Code executed for a function call



- set up function arguments
- record machine state (e.g., return address)
- jump to callee

- [clean up arguments]
- retrieve return value (if any)

Three-address code

… evaluate actuals …

$$\left.\begin{array}{l}\text{param } x_k \\ \vdots \\ \text{param } x_1\end{array}\right\} \text{R-to-L}$$

call f, k

retrieve t0 *// t0 a temporary*

copies the value returned into t0

EXERCISE

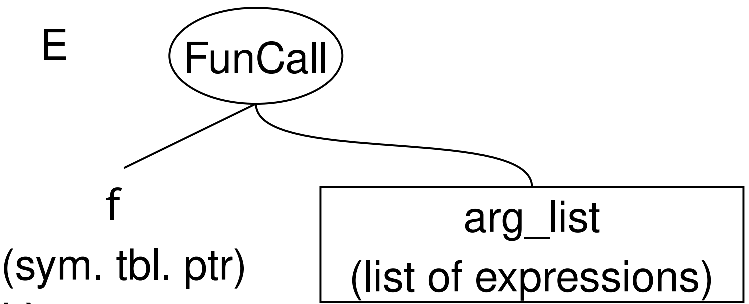
What 3-address code should the compiler generate?

Source Code	3-addr Code

void myfun(void) {
    f(1,2,3);
}
	enter myfun
	param 3
	param 2
	param 1
	call f, 3
	leave myfun
	return

Code generator: Function Calls

```
codeGen_expr(E)      /* E.nodetype == FunCall */
{
   codeGen_expr(arg_list);
   E.place = newtemp( f.returnType );
   E.code = arg_list.code
        ⊕ newinstr(PARAM,    arg_k ,     NULL, NULL)
            ...
        ⊕ newinstr(PARAM,    arg_1 ,     NULL, NULL)
        ⊕ newinstr(CALL,         f,        k,        NULL)
        ⊕ newinstr(RETRIEVE, NULL, NULL, E.place)
}
                    op       src1     src2     dest
```



omit if return type is **void**

/ ⊕ : list concatenation */*

Example

Source code

```
int add3(x,y,z) {
    u = add(x,y)
    v = add(u,z)
    return v
}
```

Three-address code

```
enter add3
param y
param x
call add
retrieve tmp0
u = tmp0
param z
param u
call add
retrieve tmp1
v = tmp1
leave add3
return tmp1
```

Example

Source code

```
int add3(x,y,z) {
    u = add(x,y)
    v = add(u,z)
    return v
}
```

Three-address code

```
enter add3
param y
param x
call add, 2
retrieve tmp0
u = tmp0
param z
param u
call add, 2
retrieve tmp1
v = tmp1
leave add3
return tmp1
```

MIPS assembly code

```
la   $sp, -8$sp
sw   $fp, 4($sp)
sw   $ra, 0($sp)
la   $fp, 0($sp)
la   $sp, -16($sp)
lw   $t0, 12($fp)
la   $sp, -4($sp)
sw   $t0, 0($sp)
lw   $t0, 8($fp)
la   $sp, -4($sp)
sw   $t0, 0($sp)
jal  _add
la   $sp, 8($sp)
⋮
```

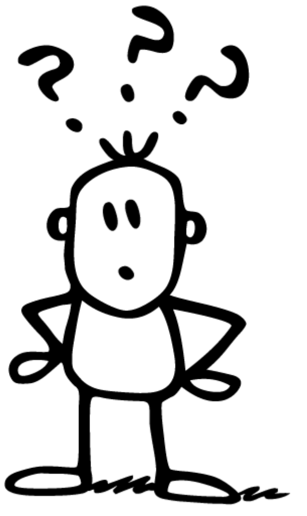
l-values and r-values

l-values and *r*-values

```
% cat prog.c
int x, y;
int main(void) {
  x++;
  x++ ++;

  x++ = ++y;

  ++y++;

  return 0;
}
%
% gcc prog.c
prog.c: In function 'main':
prog.c:4:7: error: lvalue required as increment operand
   x++ ++;
       ^
prog.c:6:7: error: lvalue required as left operand of assignment
   x++ = ++y;
       ^
prog.c:8:3: error: lvalue required as increment operand
   ++y++;
   ^
```

OK

not OK



l-values and *r*-values

```
% cat prog.c
int x, y;
int main(void) {
  x++;
  x++ ++;

  x++ = ++y;

  ++y++;

  return 0;
}
%
% gcc prog.c
prog.c: In function 'main':
prog.c:4:7: error: lvalue required as increment operand
   x++ ++;
       ^
prog.c:6:7: error: lvalue required as left operand of assignment
   x++ = ++y;
       ^
prog.c:8:3: error: lvalue required as increment operand
   ++y++;
   ^
```

We can only assign to (or update) expressions that correspond to locations.

Such expressions are called *l-values*

l-values and *r*-values

- l-value: something that can appear on the left-hand side of an assignment
- r-value (or simply "value"): something that can appear on the right-hand side of an assignment

Some expressions can be both l-values and r-values:

need the location to update

a[i] = a[i] + 1

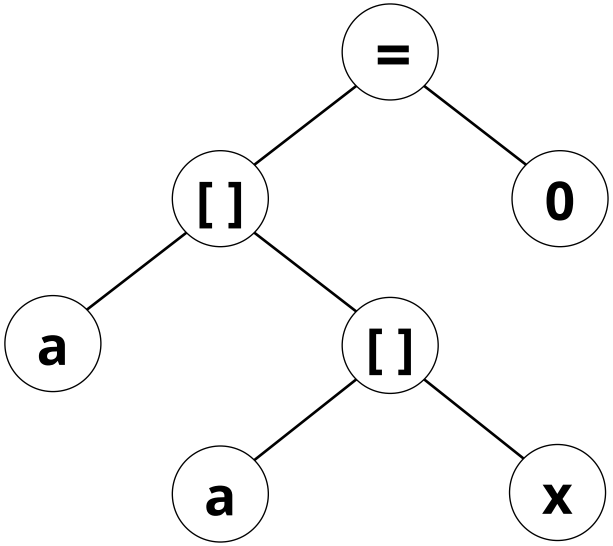
need the value to update it with

When generating code, the compiler needs to know whether to compute an l-value of an r-value

EXERCISE

Source code: a[a[x]] = 0

At each node of the syntax tree, identify whether the value is a l-value or r-value:



l-values and *r*-values: code generation

- Pass an argument to codeGen_expr() indicating whether l-value or r-value needed

  codeGen_expr(expr, lr)

  syntax tree of expression　　*L_value or R_value*

- The generated code uses this argument to return a value of the appropriate kind

Code generation for simple statements

Assignments

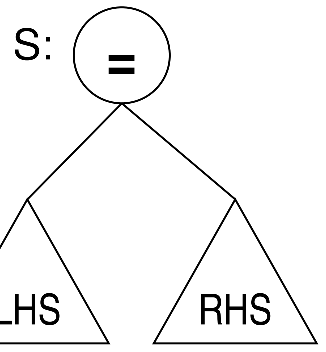
<u>*Source Code:*</u> LHS = RHS

<u>*Code structure*</u>:

evaluate LHS (l-value)

evaluate RHS (r-value)

copy value of RHS into LHS



Assignments

codeGen_stmt(S)   /* S.nodetype == '=' */

{

    codeGen_expr(LHS, *L_value*);

    codeGen_expr(RHS, *R_value*);

    S.code = LHS.code

        ⊕ RHS.code

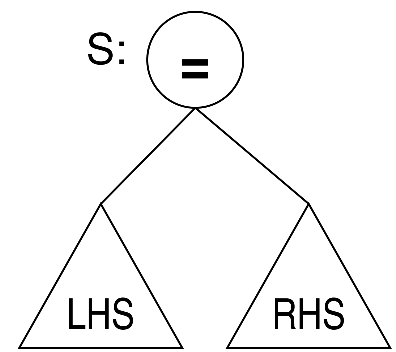
        ⊕ *newinstr*(**ASSG**, RHS.place, NULL, LHS.place) ;

op *src1* *src2* *dest*

}

enum indicating the desired kind of value

if LHS is an array element or struct field, LHS.place ≡ **deref**(LHS.loc)



Type conversions

Type conversion

- Values of different types may be represented differently at the machine level
- When the code to be compiled operates on values of different types, the compiler may have to insert code to convert from one representation to another
  - *implicit type conversion*: type conversion done by the compiler without explicit user input
  - *explicit type conversion*: user-directed type casting

Implicit type conversion: Examples

Assignments: *var* = *exp* • *var* has a different type than *exp* • the compiler adds code to convert the value of exp to the type of var before the assignment	**Arithmetic expressions**: exp1 + exp2 • exp1 has a different type than exp2 • the type conversion is language-specified: - usually the smaller type is converted to the larger, e.g., char to int
Array indexing: A[exp] • exp is not an int (e.g., a char) • the compiler adds code to convert exp to an int value	**Parameter passing**: f(exp) • exp is not the right type for parameter passing (e.g., a char) • the compiler adds code to convert exp appropriately

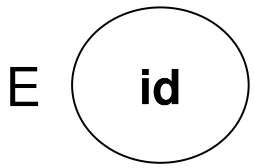
Code generation for expressions I

Expressions 1: Scalar variables

Source Code: **id**

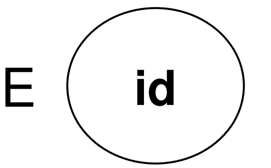
Code structure:

(no code needed)



Expressions 1: Scalar variables

```
codeGen_expr(E, lr)    /* E.nodetype == Var; */
{
    E.place = id.loc;  /* location: from symbol table */
    E.code = NULL;
}
```



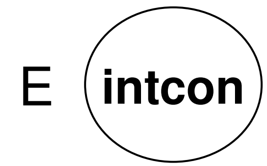
Expressions 2: Integer constants

Source Code: **intcon**

Code structure:

tmp = **intcon**.value

temporary
assigned the value of the constant



Expressions 2: Integer constants

```
codeGen_expr(E, lr)    /* E.nodetype == INTCON; */
{
    if (lr == L_value) {
        ERROR;
    }
    else {
        E.place = newtemp(E.type);
        E.code = newinstr(ASSG, intcon.val, NULL, E.place);
    }                   op        src1        src2      dest
}
```

E **intcon**

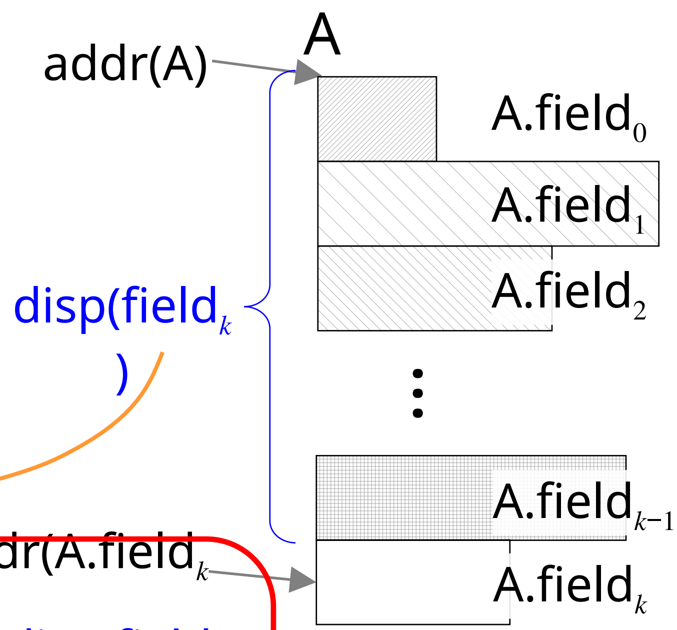
value of the integer constant
(computed by scanner)

Example

Source code	Three-address code	MIPS assembly code
x = 12	tmp0 = 12	li $t0, 12 sw $t0, -4($fp)
	x = tmp0	lw $t1, -4($fp) sw $t1, -12($fp)

Expressions 3: *struct* fields

- Source code: A.$field_k$
- Generated code needs to:
  1. compute the address of A.$field_k$ $\equiv$ addr(A.$field_k$)
  2. access addr(A.$field_k$) (read/write as appropriate)



= addr(A) + disp($field_k$)

= addr(A) + width($field_0$) + width($field_1$) ⋮ + width($field_{k-1}$)

from symbol table

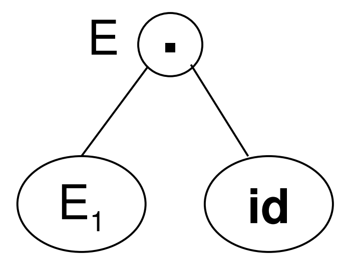
disp($field_k$) can be computed by the compiler

Expressions 3: *struct* fields

```
codeGen_expr(E, lr)          /* E.nodetype == STRUCT_REF */
{
   codeGen_expr(E1, L_value);
   tmp1 = newtemp( address );
   E.code = E1.code
        ⊕ newinstr(PLUS, E1.loc, OFFSET(E, id) , tmp1)
   if (lr == L_value) {
      E.loc = tmp1;
   }
   else {    /* R_value */
      E.place = newtemp( E1.id.type )
      E.code = E.code ⊕ newinstr(DEREF, tmp1, E.place)
   }
}
```

this offset is computed by the compiler

$E \equiv E_1$.**id**



Code generation for arithmetic operations

Arithmetic Expressions 1: Unary Ops

Source Code: $-$E

Code Structure:

some_loc = *...value of E...*

tmp = $-$ some_loc

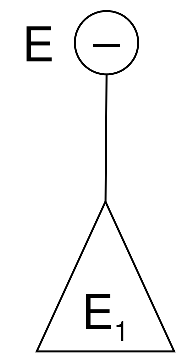
temporary

some_loc $\equiv$ E.place

Arithmetic Expressions 1: Unary Ops

```
codeGen_expr(E, lr)    /* E.nodetype == UNARY_MINUS */
{
   if (lr == L_value) {
      ERROR;
   }
   else {
      codeGen_expr(E1, R_value);
      E.place = newtemp( E.type );
      E.code = E1.code
               ⊕  newinstr(UMINUS, E1.place, NULL, E.place);
                           op      src1      src2   dest
   }
}
```

$E \equiv -E_1$



Arithmetic Expressions 2: Binary Ops

Source Code: $E_1 + E_2$

Code Structure:

$loc_1 \equiv E_1.place$

$\boxed{loc_1}$ = *...value of E1...*

$\boxed{loc_2}$ = *...value of E2...*

tmp = $\boxed{loc_1} + \boxed{loc_2}$

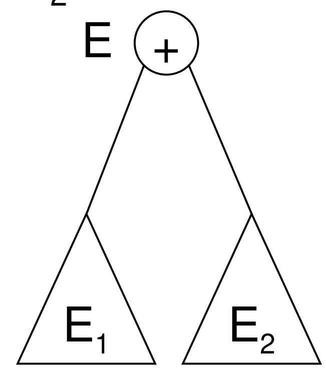
$loc_2 \equiv E_2.place$

temporary
contains the value of the expression

Arithmetic Expressions 2: Binary Ops

```
codeGen_expr(E, lr)        /* E.nodetype == PLUS */
{
  if (lr == L_value) {
    ERROR;
  }
  else {
    codeGen_expr(E1, R_value);
    codeGen_expr(E2, R_value);
    E.place = newtemp( E.type );
    E.code = E1.code
          ⊕ E2.code
          ⊕ newinstr(PLUS, E1.place, E2.place, E.place);
                     op    src1      src2      dest
  }
}
```

$E \equiv E_1 + E_2$



other binary arithmetic operators are similar

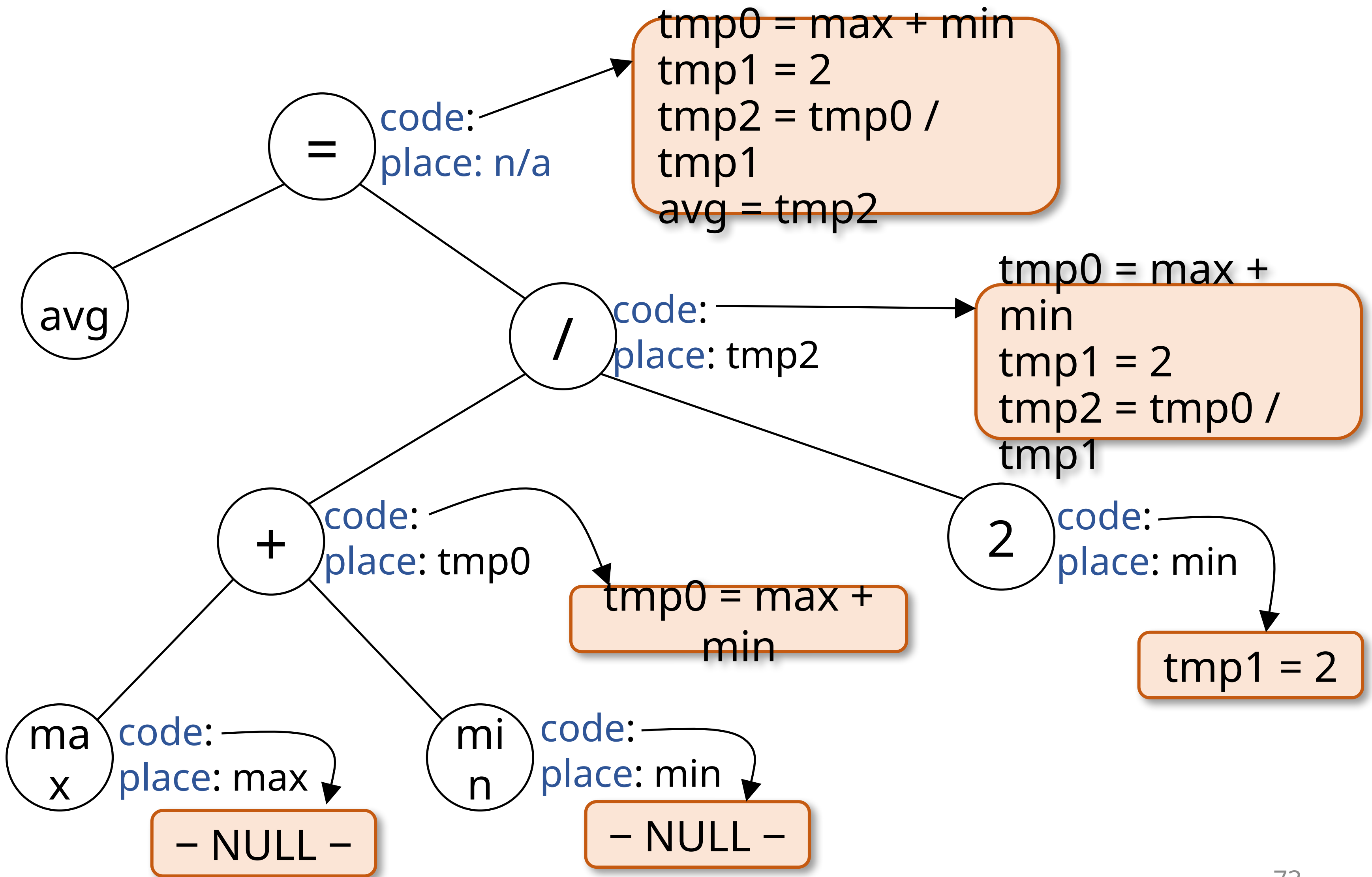
EXERCISE

Consider the statement

$$avg = (max + min)/2$$

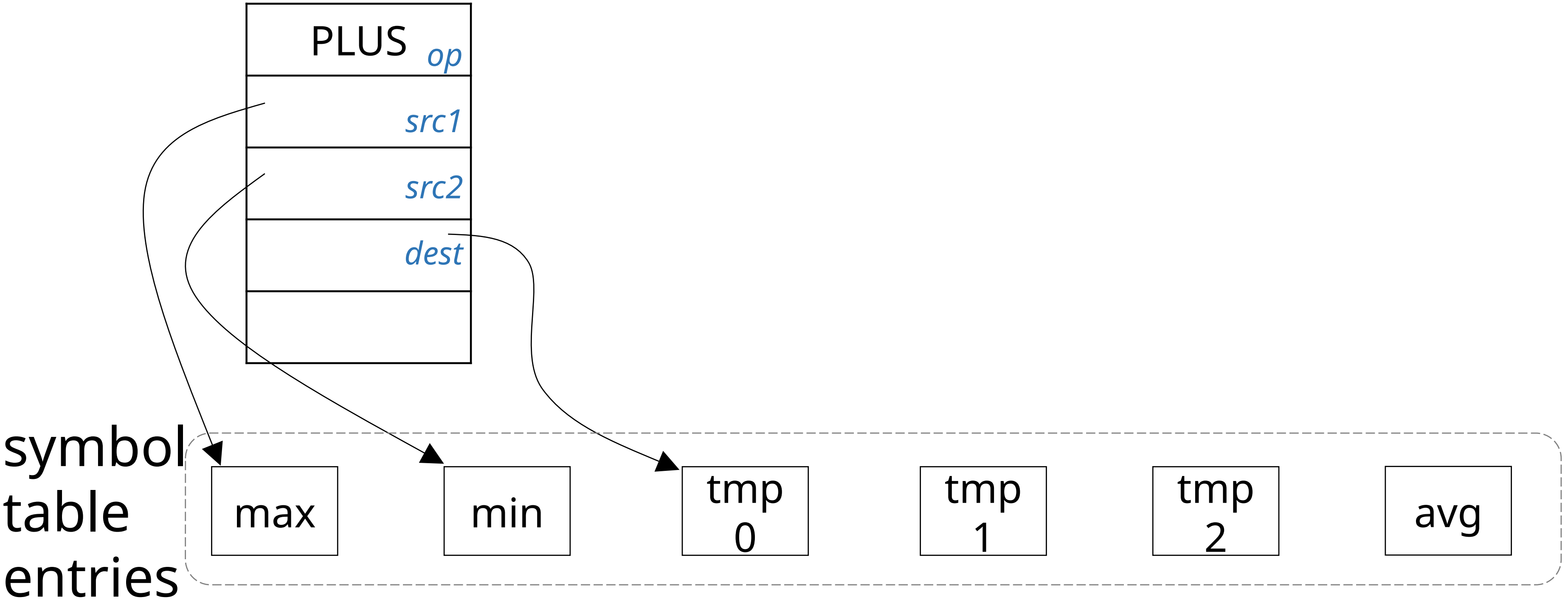
1. Construct the AST for this code
2. Show the three-address code at each node of your AST
3. Based on your answer to part 2 of this problem, indicate the values of the place and code fields at each node of your AST

EXERCISE - solution



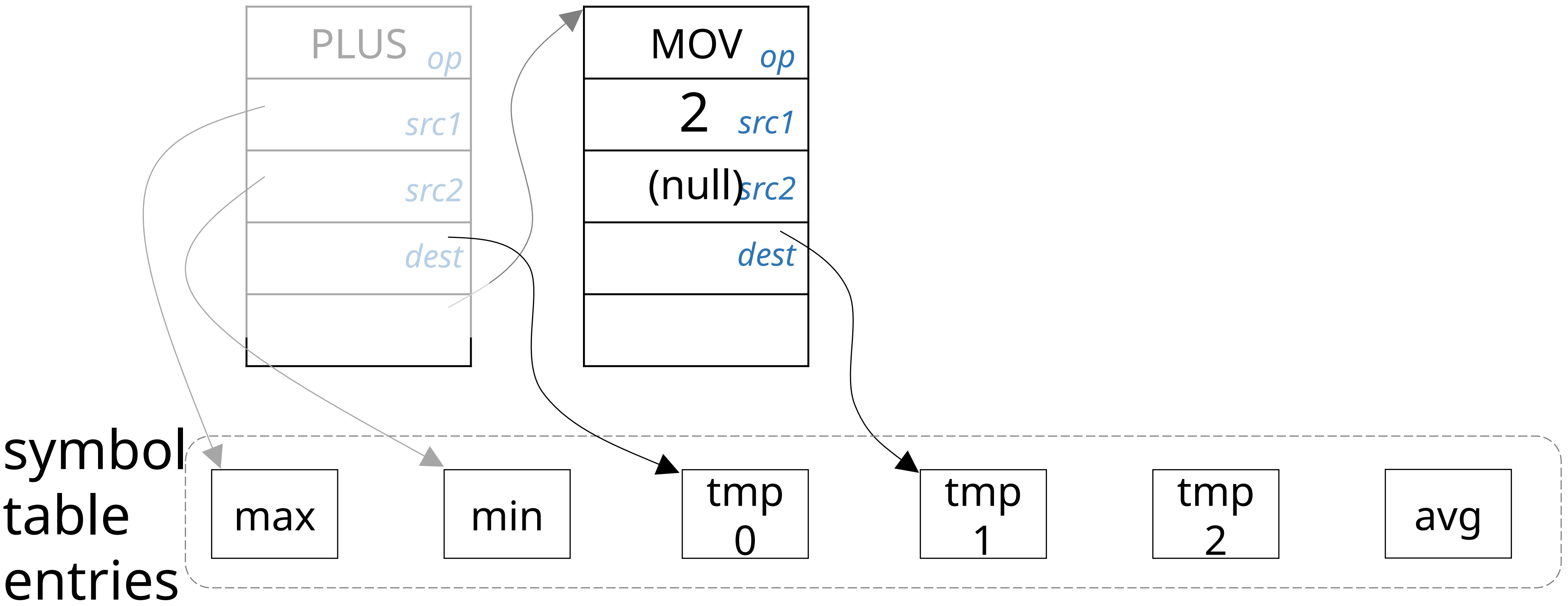
EXERCISE - solution

```
tmp0 = max + min
tmp1 = 2
tmp2 = tmp0 / tmp1
avg = tmp2
```



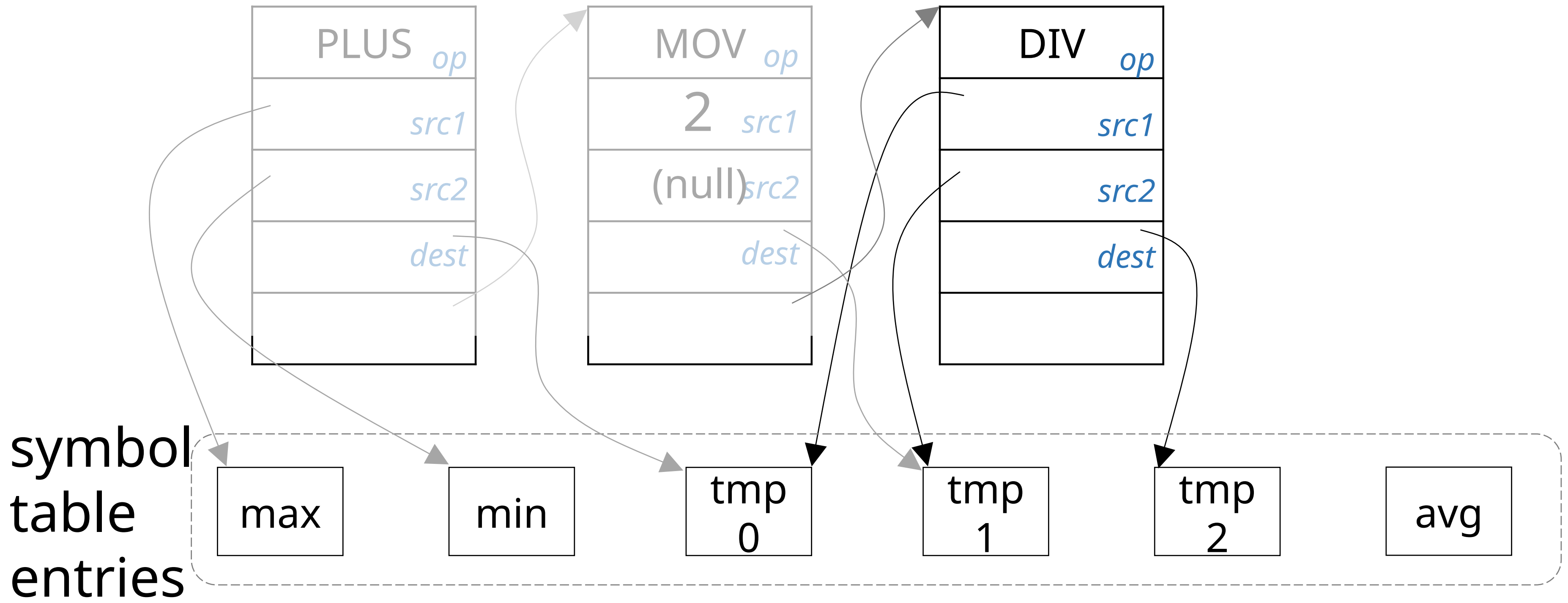
EXERCISE - solution

tmp0 = max + min
tmp1 = 2
tmp2 = tmp0 / tmp1
avg = tmp2



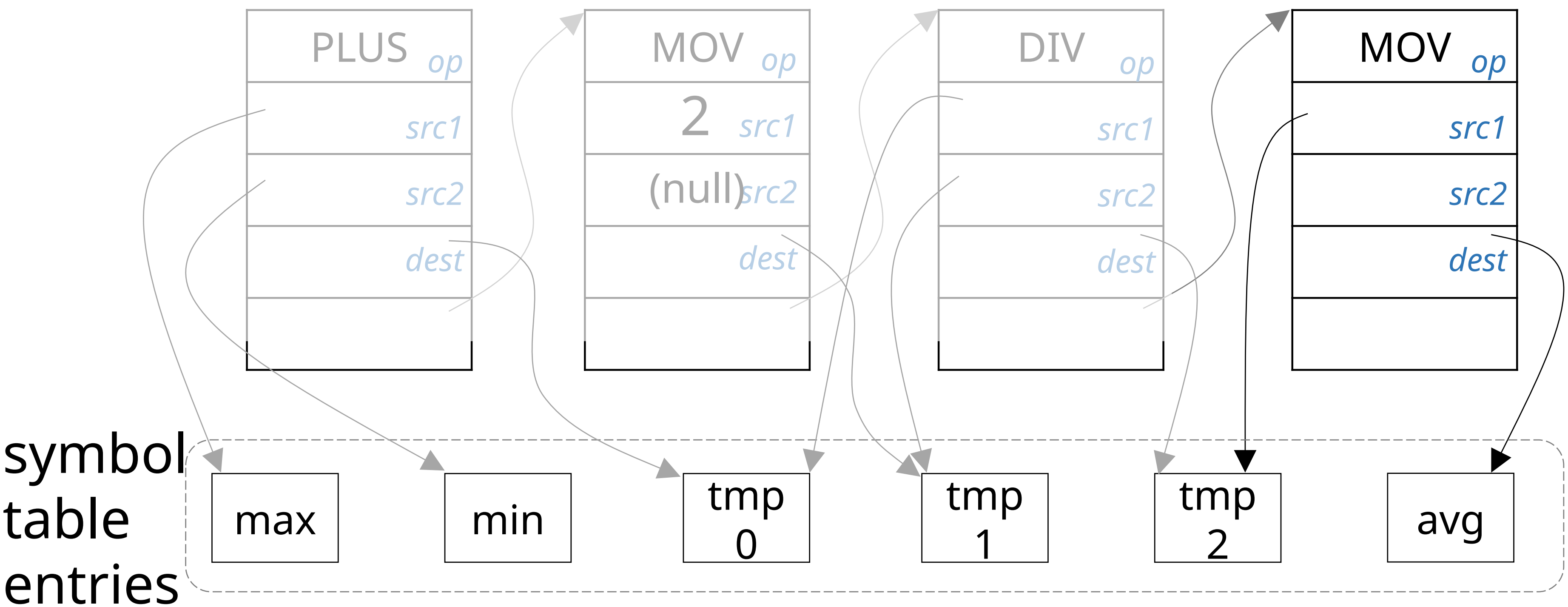
EXERCISE - solution

tmp0 = max + min
tmp1 = 2
tmp2 = tmp0 / tmp1
avg = tmp2



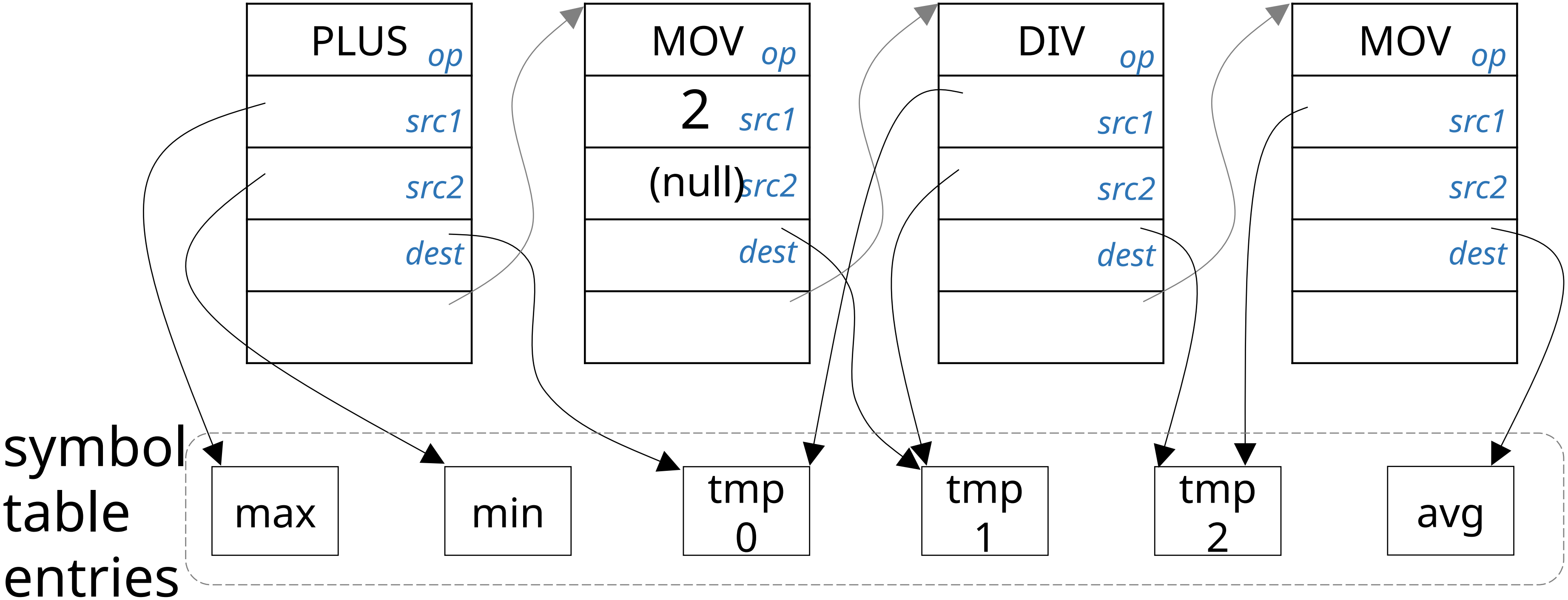
EXERCISE - solution

tmp0 = max + min
tmp1 = 2
tmp2 = tmp0 / tmp1
avg = tmp2



EXERCISE - solution

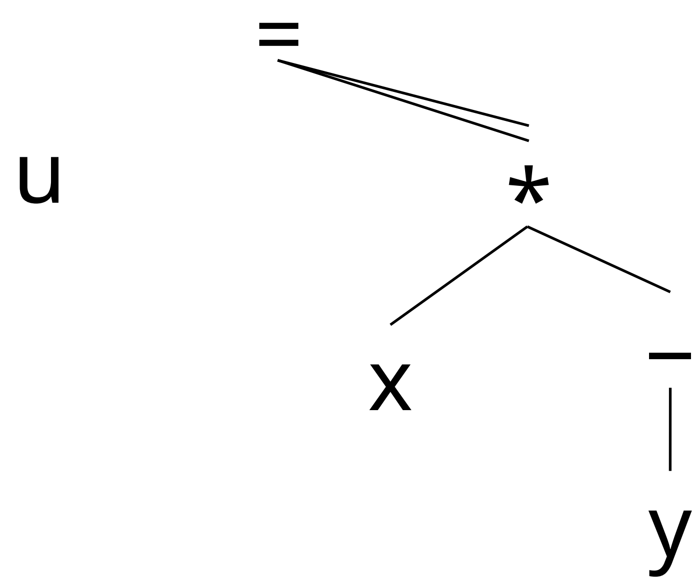
tmp0 = max + min
tmp1 = 2
tmp2 = tmp0 / tmp1
avg = tmp2



EXERCISE

Source code: u = x * − y

Work out the 3-addr code at each node:



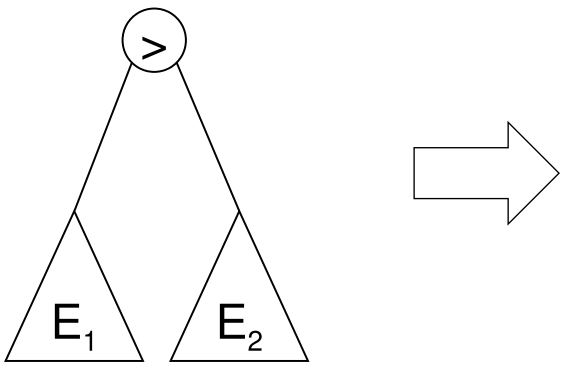
Reusing Temporaries

- Storage usage can be reduced considerably by reusing temporaries:
  - For each type T, keep a “free list” of temporaries of type T;
  - *newtemp*(T) uses a temp from the free list for type T whenever possible
- Putting temps on the free list:
  - free only compiler-generated temps, not user variables
  - free a temp after the point of its last use (i.e., when its value is no longer neded).

Code generation for comparison and logical operations

Logical Expressions

Naïve but Simple Approach (TRUE=1, FALSE=0):



tmp1 = … *evaluate* E_1 …

tmp2 = … *evaluate* E_2 …

tmp3 = 1 /* TRUE */

if (tmp1 > tmp2) goto L

tmp3 = 0 /* FALSE */

L: …

Disadvantage: lots of unnecessary memory references

Logical Expressions

Observation: Logical expressions are usually used to direct flow of control.

```
if ( x > 0 ) {
    S1
}
else {
    S2
}
```

True

False

```
while ( x > 0 ) {
    S1
}
```

True

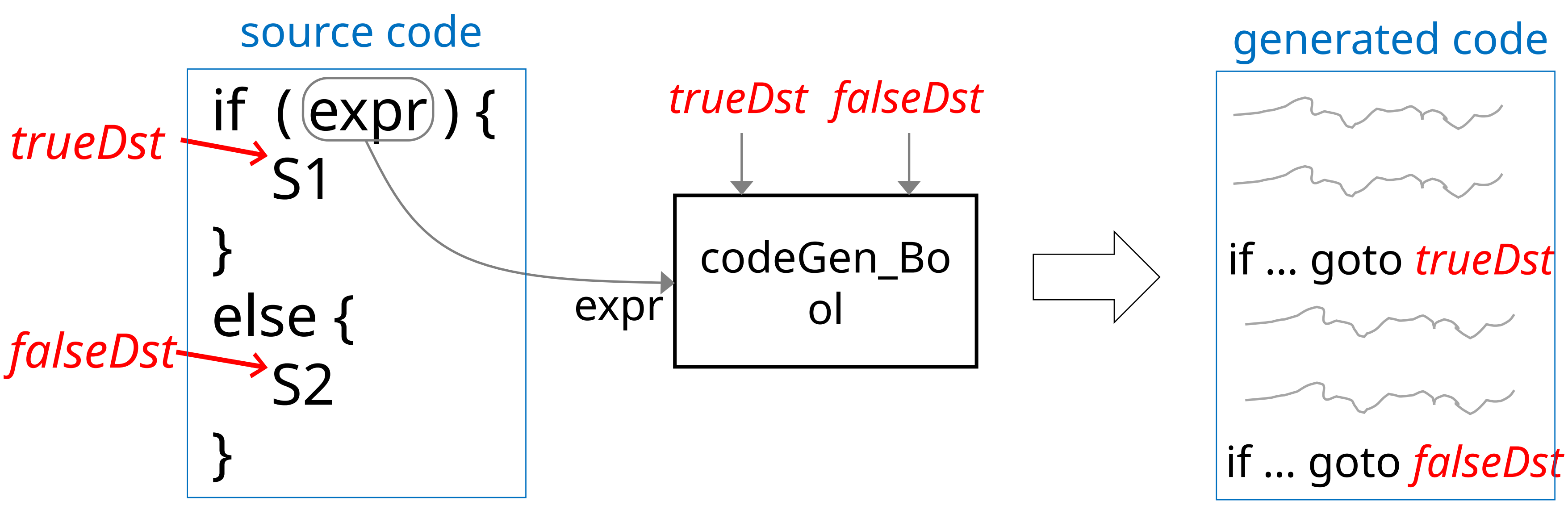
False

Can we generate code to do this without all the unnecessary memory operations?

Logical Expressions

Idea: "Tell" the code generator where the generated code should jump:

- *trueDst*: label to jump to if expression is True
- *falseDst*: label to jump to if expression is False



Logical Expressions

codeGen_Bool(E, trueDst, falseDst) /* E.nodetype == '>' */

{

  codeGen_expr(E_1, *R_value*);

  codeGen_expr(E_2, *R_value*);

IF_GT ≡ "if … greater than"

E.code = E_1.code

⊕ E_2.code

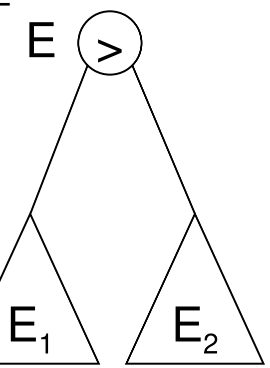
⊕ *newinstr*(**IF_GT**, E_1.place, E_2.place, trueDst)

⊕ *newinstr*(**GOTO**, NULL, NULL, falseDst)

op src1 src2 dest

}

$E \equiv E_1 > E_2$



other binary comparison operators are similar

Example

if (x+y != 2*z) {

L_{true} → S1

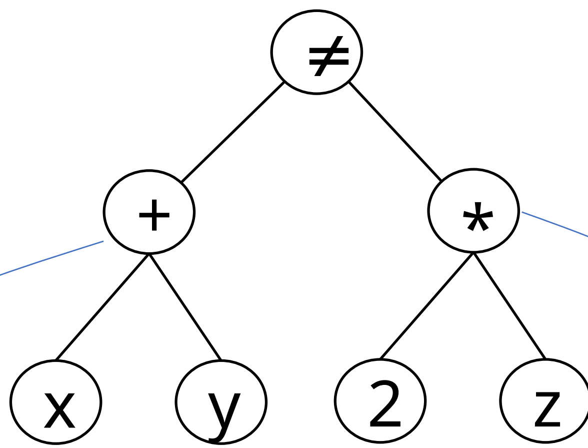
}

else {

L_{false} → S2

}

$E.code = E_1.code \oplus E_2.code$
$\oplus$ *newinstr*(IF_NE, E_1.place, E_2.place, trueDst)
$\oplus$ *newinstr*(GOTO, NULL, NULL, falseDst)



place: tmp0
code:
    tmp0 = x + y

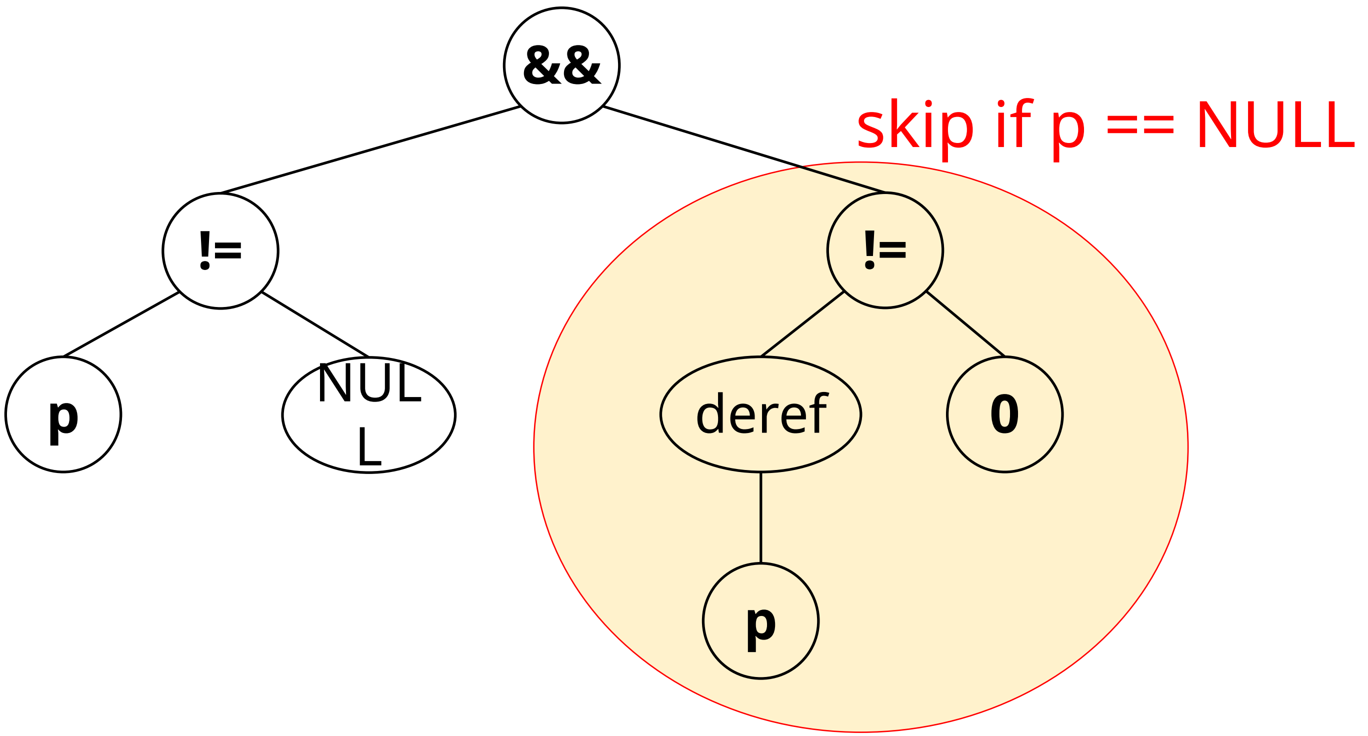
place: tmp2
code:
    tmp1 = 2
    tmp2 = tmp1 * z

generated code:

```
tmp0 = x + y
tmp1 = 2
tmp2 = tmp1 * z
if tmp0 != tmp2 goto L_true
goto L_false
```

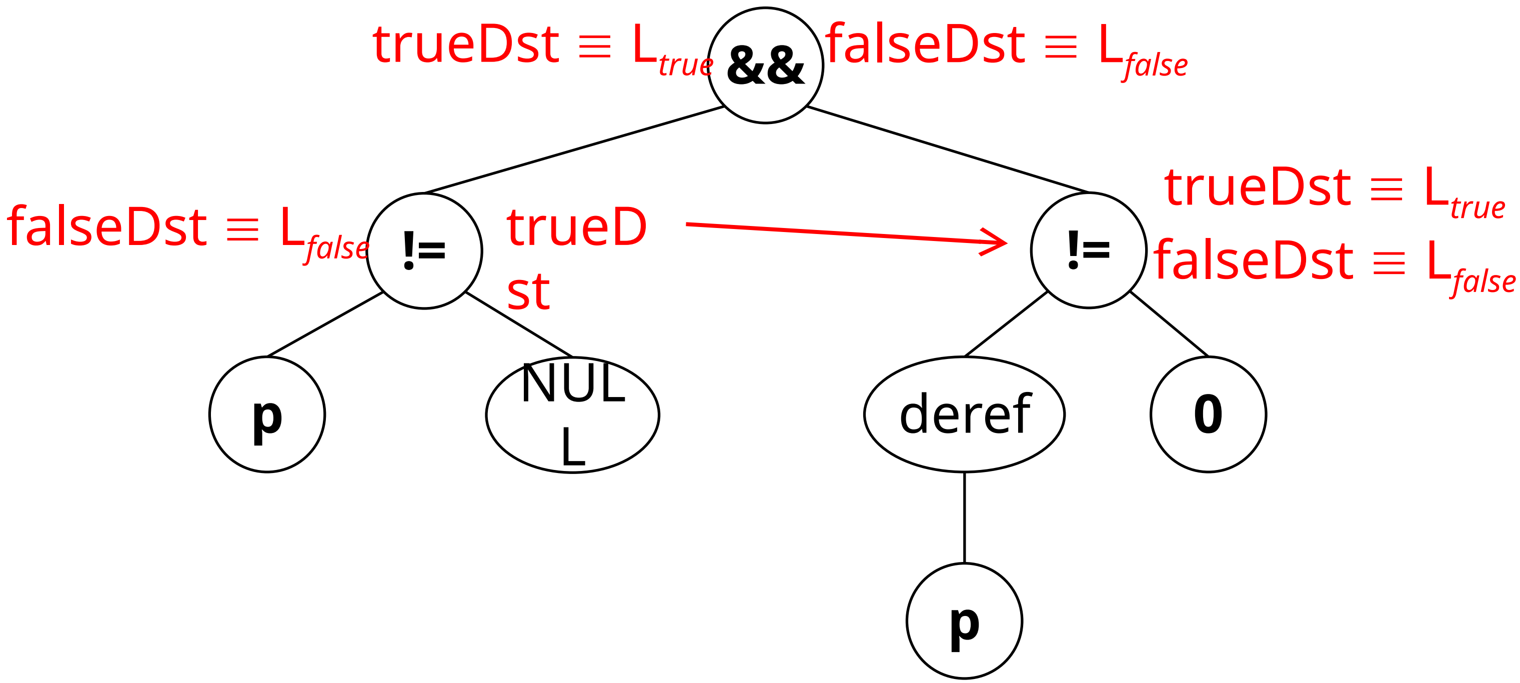
Short Circuit Evaluation

Idea: Evaluate logical expressions only to the extent necessary to determine their truth values



Short Circuit Evaluation

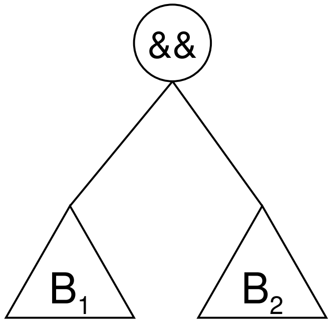
<u>Idea</u>: Evaluate logical expressions only to the extent necessary to determine their truth values



Short Circuit Evaluation

```
codeGen_Bool(B, trueDst, falseDst)        /* B.nodetype == '&&' */
{
    L1 = newlabel( );
    codeGen_bool(B1, L1, falseDst);
    codeGen_bool(B2, trueDst, falseDst);
    B.code = B1.code ⊕ L1 ⊕ B2.code;
}
```

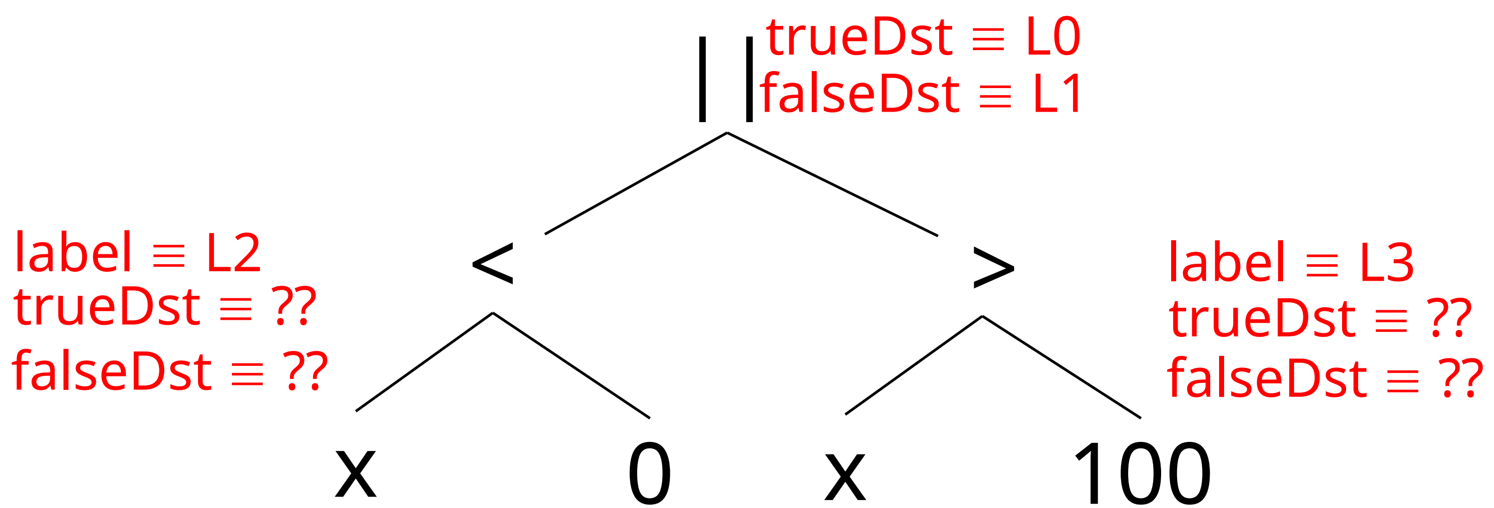
$B \equiv B_1$ && B_2



other logical operators (||, !) are analogous

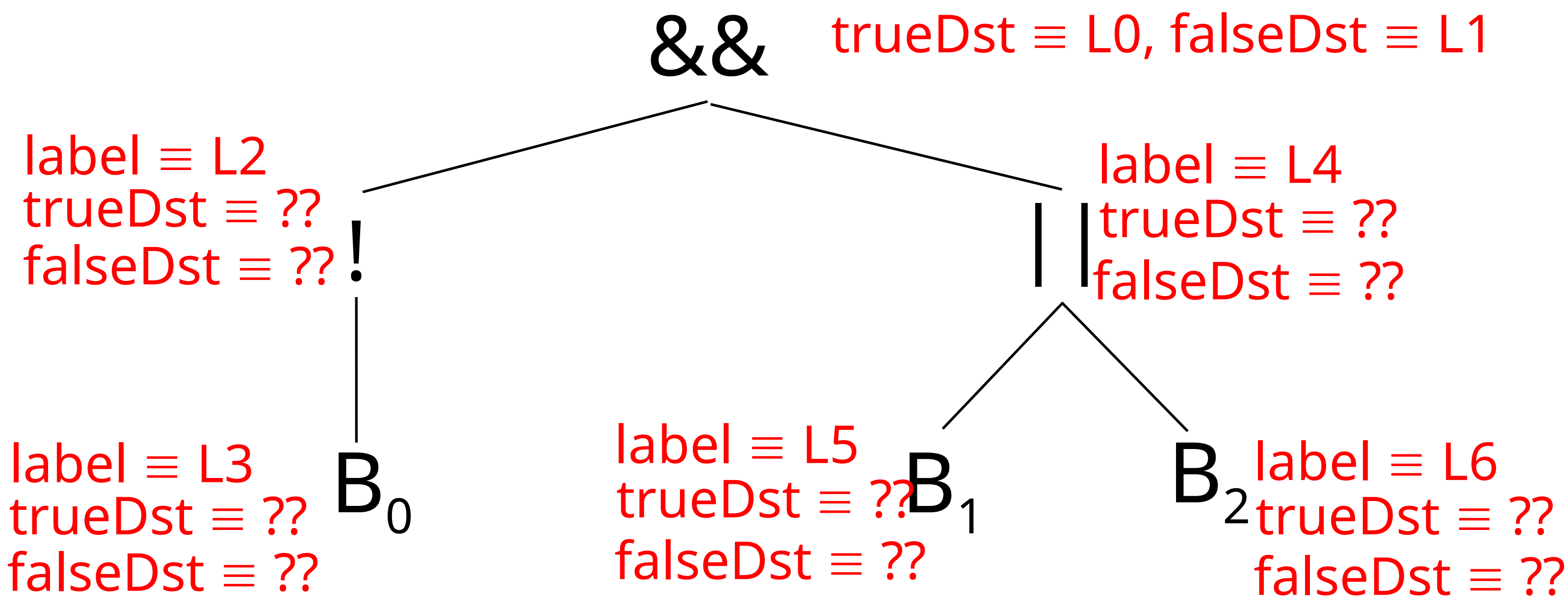
EXERCISE

Compute trueDst and falseDst at each comparison/logical node:



EXERCISE

Compute trueDst and falseDst at each comparison/logical node:



Code generation for control flow statements

Conditionals

Source Code: if B then S_1 else S_2

Code Structure:

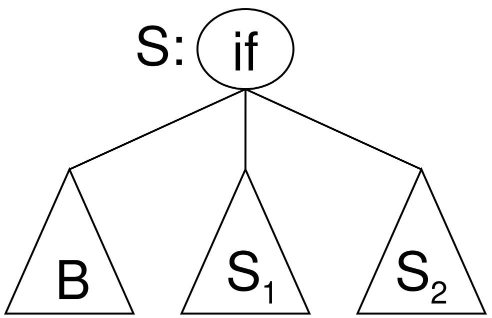
code to evaluate B

L_{then}: code for S_1

goto L_{after}

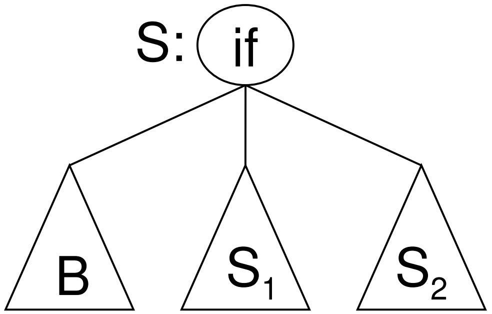
L_{else}: code for S_2

L_{after}: ...

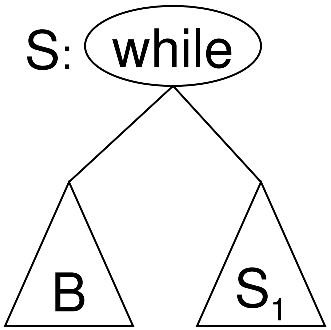


Conditionals

codeGen_stmt(S) /* S.nodetype == IF */
{
L_{then} = *newlabel*(); L_{else} = *newlabel*(); L_{after} = *newlabel*();
codeGen_bool(B, L_{then}, L_{else});
codeGen_stmt(S_1);
codeGen_stmt(S_2);
S.code = B.code
⊕ L_{then} ⊕ S_1.code
⊕ *newinstr*(**GOTO**, NULL, NULL, L_{after})
⊕ L_{else} ⊕ S_2.code
⊕ L_{after} ;
}



Loops 1



Code Structure:

L_{top} : code to evaluate B

if (!B) goto L_{after}

$L_{body:}$ code for S_1

goto L_{top}

L_{after}: …

codeGen_stmt(S)
{ /* S.nodetype == WHILE */

L_{top} = *newlabel*();

L_{body} = *newlabel*();

L_{after} = *newlabel*();

codeGen_bool(B, L_{body}, L_{after});

codeGen_stmt(S_1);

S.code = L_{top}

⊕ B.code

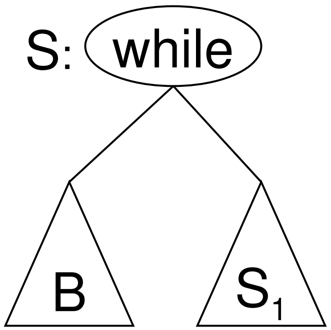
⊕ L_{body}

⊕ S1.code

⊕ *newinstr*(**GOTO**, NULL, NULL, L_{top})

⊕ L_{after} ;

Loops 2



<u>*Code Structure*</u>:

goto L_{eval}

$L_{top:}$ code for S_1

L_{eval}: code to evaluate B

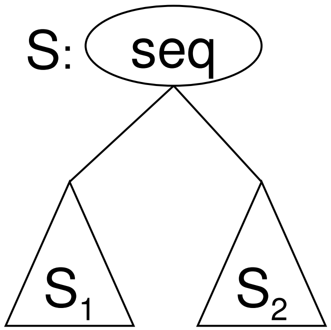
if (B) goto L_{top}

L_{after}: ...

This code executes fewer branch ops

codeGen_stmt(S)
{ /* S.nodetype == WHILE */
L_{top} = *newlabel*();
L_{eval} = *newlabel*();
L_{after} = *newlabel*();
codeGen_bool(B, $L_{top,}$ L_{after});
codeGen_stmt(S_1);
S.code =
newinstr(**GOTO**, NULL, NULL, L_{eval})
$\oplus$ L_{top} $\oplus$ S_1.code
$\oplus$ L_{eval} $\oplus$ B.code
$\oplus$ L_{after}
}

Handling a Sequence of Statements



codeGen_stmt(S) {   /* S.nodetype == SEQ */

codeGen_stmt(S_1);

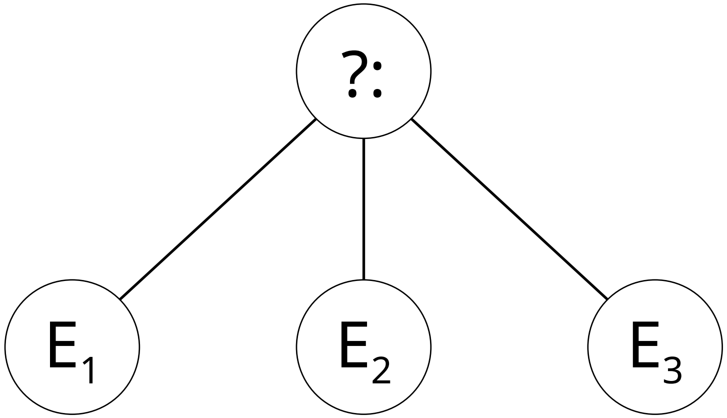
codeGen_stmt(S_2);

S.code = S_1.code $\oplus$ S_2.code

}

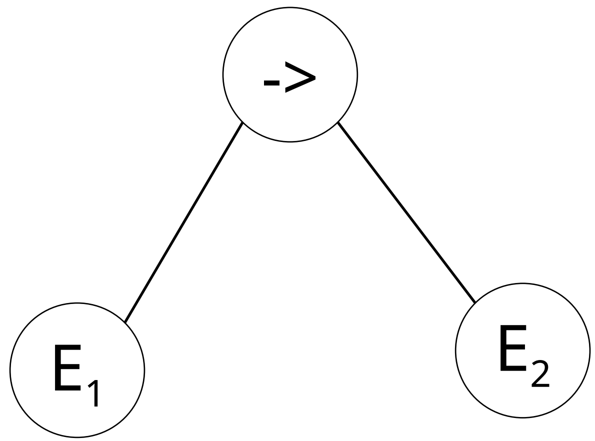
EXERCISE

- Ternary expressions: $E \equiv E_1 ? E_2 : E_3$



EXERCISE

- Deref and load: $E \equiv E_1 \text{->} E_2$

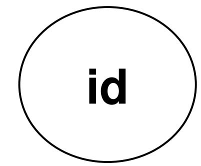


Code generation for runtime-computed addresses (Arrays)

Expressions 4a: Array/struct variables

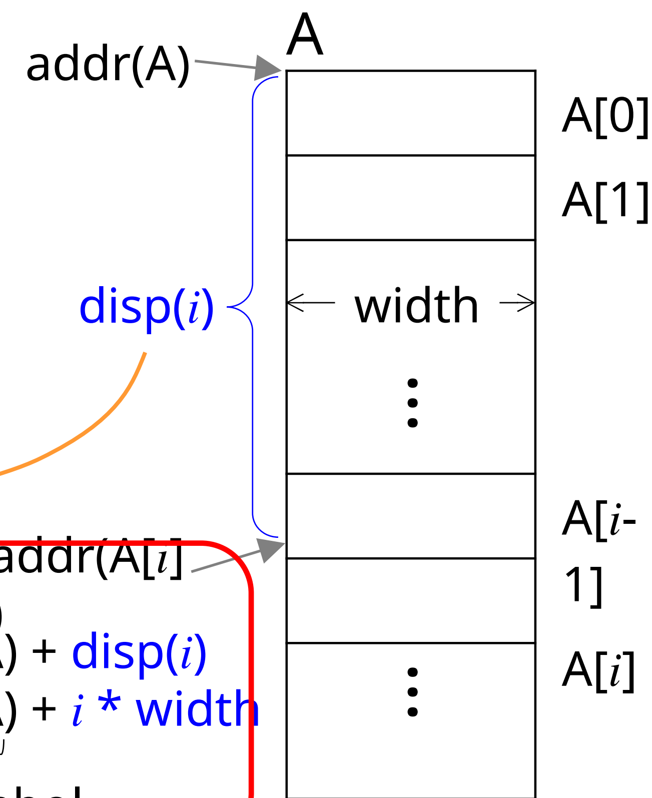
```
codeGen_expr(E, lr)    /* E.nodetype == ARRAY */
{
    if (lr == L_value) {
        E.loc = id.loc;        /* location: from symbol table */
    }
    else {
        ERROR
    }
     E.code = NULL;
}
```

E



Expressions 4b: array references

- Source code: A[i]
- Generated code needs to:
  1. evaluate i
  2. compute address of A[i] $\equiv$ addr(A[i])
  3. access addr(A[i]) (read/write as appropriate)



= addr(A) + disp(i)
= $\underbrace{\text{addr(A)}}_{\text{from symbol table}}$ + i * width

disp(i) usually cannot be computed by the compiler

∴ need to generate code to compute it when the program executes

Expressions 4b: Array references

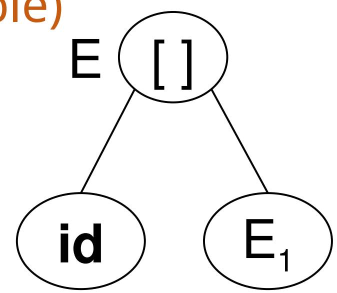
```
codeGen_expr(E, lr)          /* E.nodetype == ARRAY_REF; */
{
   codeGen_expr(E1, R_value);
   tmp1 = newtemp( int );
   tmp2 = newtemp( address );
   E.code = E1.code
      ⊕ newinstr(MULT, E1.place, WIDTH(id.elt_type), tmp1)
      ⊕ newinstr(PLUS, id.loc, tmp1, tmp2)
   if (lr == L_value) {
      E.loc = tmp2
   }
   else {   /* R_value */
      E.place = newtemp( id.elt_type )
      E.code = E.code ⊕ newinstr(DEREF, tmp2, NULL, E.place)
   }
}
```

id's location (from symbol table)

displacement (in bytes) to E_1^{th} element

address of E_1^{th} element

$E \equiv \textbf{id}\,[E_1]$



Expressions 4b: Array references

codeGen_expr(E, lr)   /* E.nodetype == ARRAY_REF; */

{

  codeGen_expr(E_1, *R_value*);

  tmp1 = *newtemp*(int);

  tmp2 = *newtemp*(address);

  E.code = E_1.code

    ⊕ *newinstr*(**MULT**, E_1.place, WIDTH(**id**.elt_type), tmp1)

    ⊕ *newinstr*(**PLUS**, **id**.loc, tmp1, tmp2)

  if (lr == *L_value*) {

    E.loc = tmp2

  }

  else {   /* *R_value* */

    E.place = newtemp(**id**.elt_type)

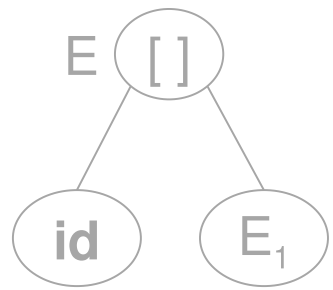
    E.code = E.code ⊕ newinstr(**DEREF**, tmp2, NULL, E.place)

  }

}

compute the address of the array element

E ≡ **id** [E_1]



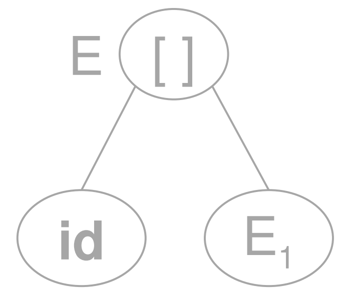
Expressions 4b: Array references

```
codeGen_expr(E, lr)          /* E.nodetype == ARRAY_REF; */
{
   codeGen_expr(E1, R_value);
   tmp1 = newtemp( int );
   tmp2 = newtemp( address );
   E.code = E1.code
      ⊕ newinstr(MULT, E1.place, WIDTH(E1.elt_type), tmp1)
      ⊕ newinstr(PLUS, id.loc, tmp1, tmp2)
   if (lr == L_value) {
      E.loc = tmp2
   }
   else {    /* R_value */
      E.place = newtemp( id.elt_type )
      E.code = E.code ⊕ newinstr(DEREF, tmp2, NULL, E.place)
   }
}
```

the L-value of the array element is its address

to compute its R-value, dereference its address

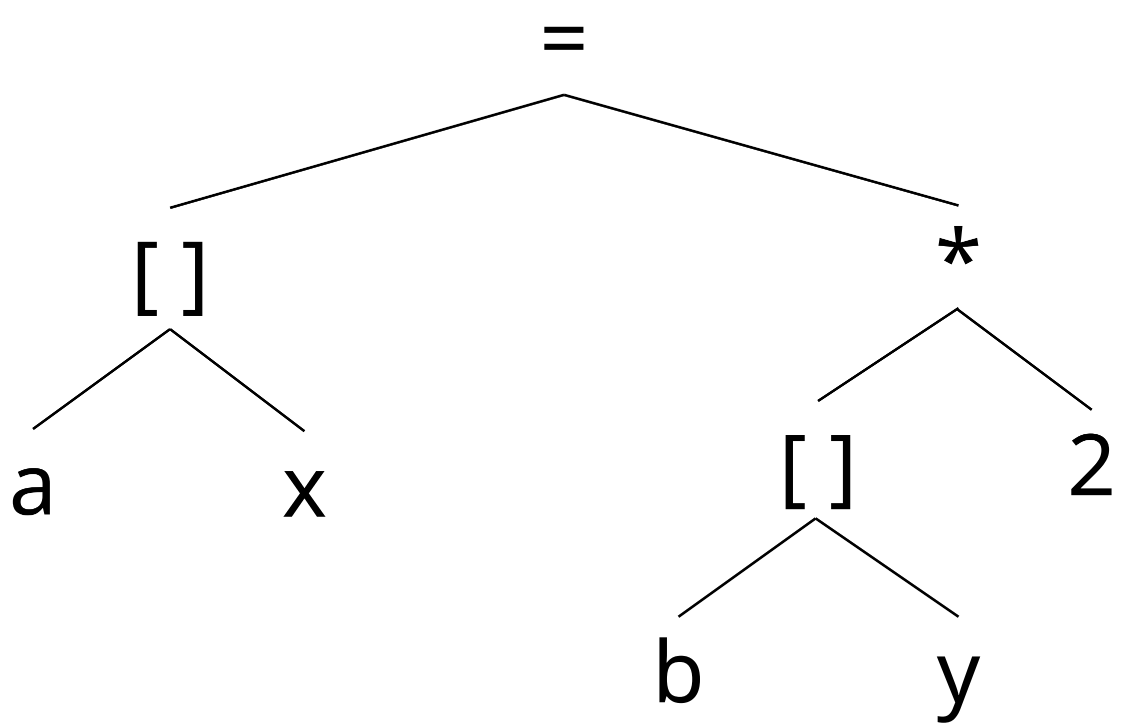
$E \equiv$ **id** $[E_1]$



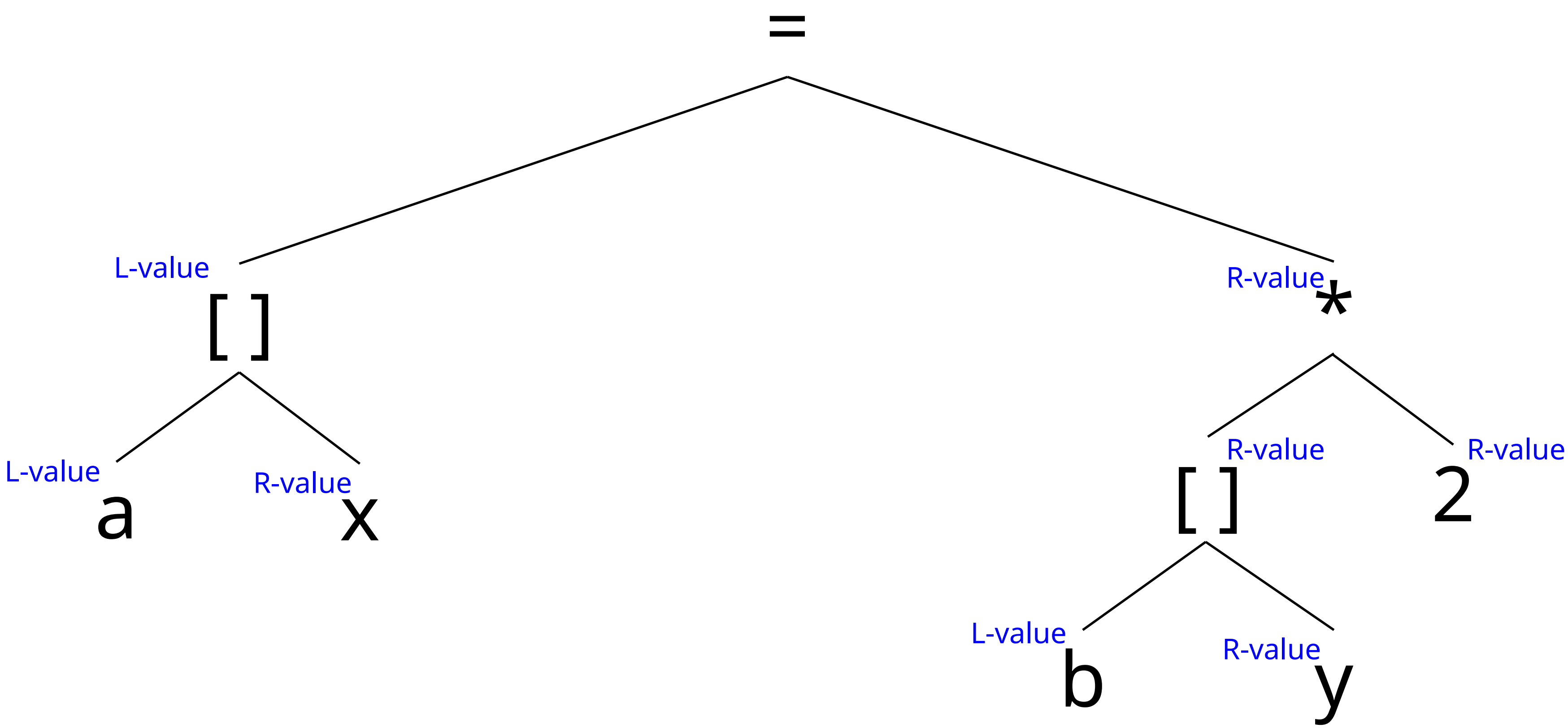
Example 1

Source code: a[x] = b[y] + 1

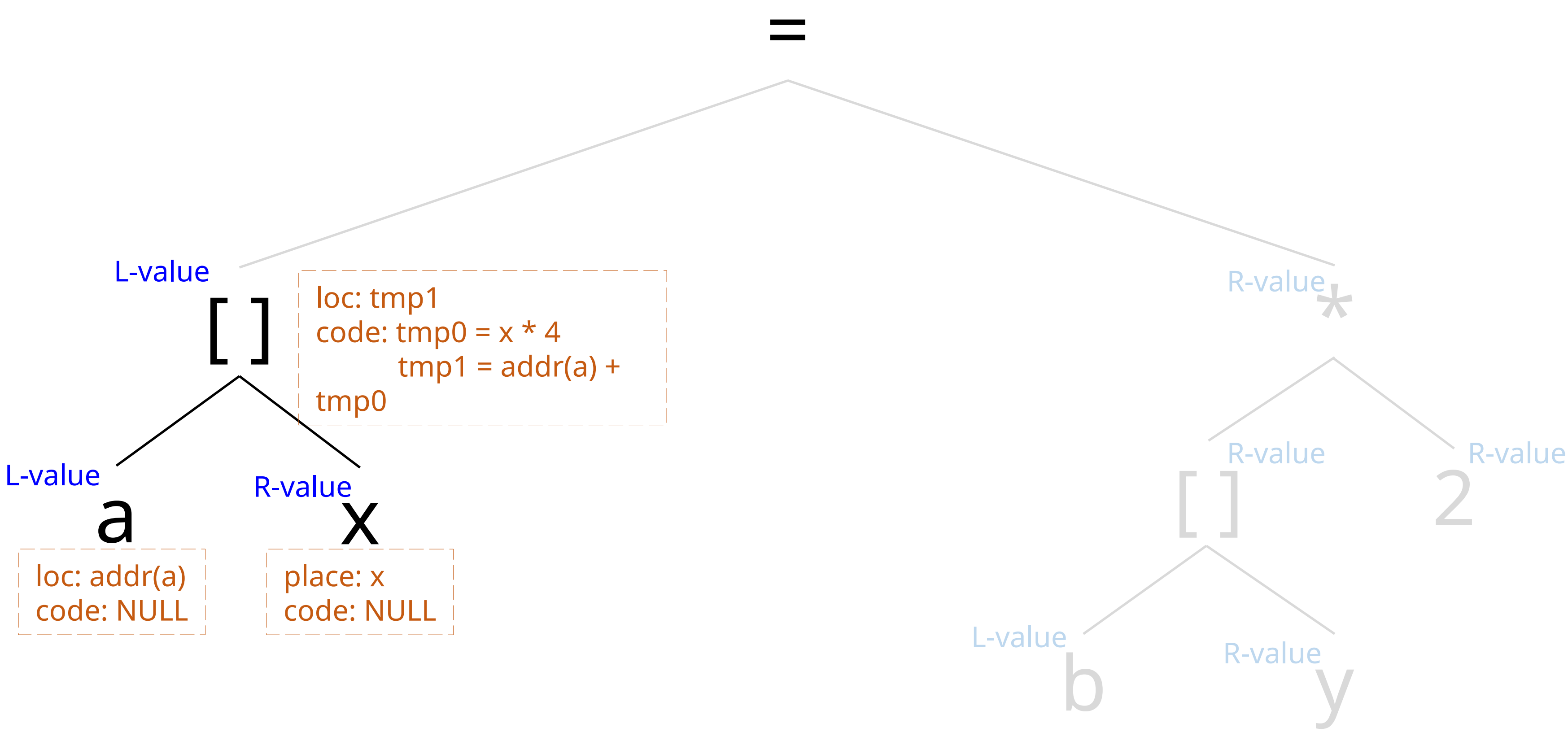
AST:



Example 1

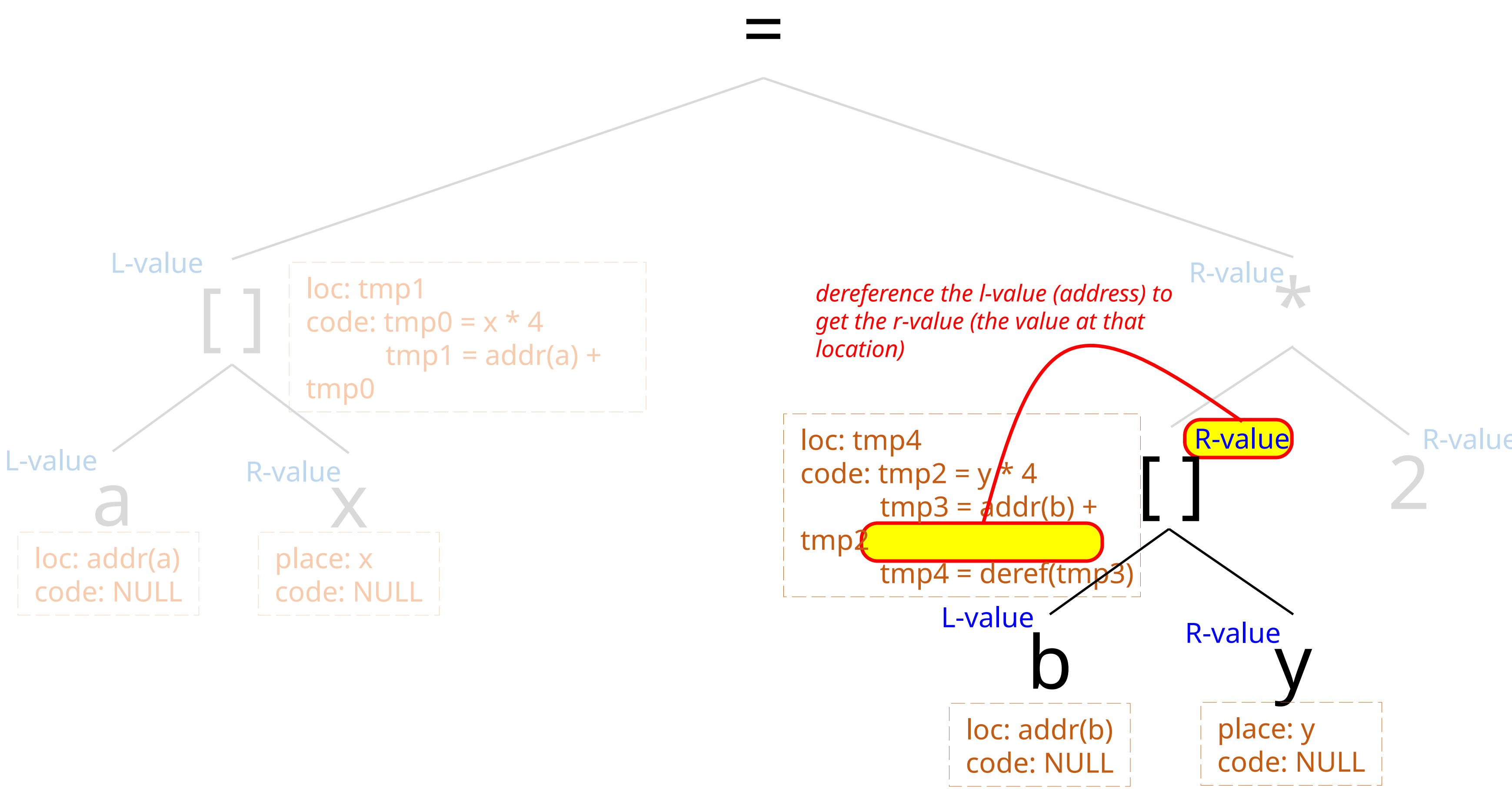

=
L-value
[ ]
R-value
*
L-value
a
R-value
x
R-value
[ ]
R-value
2
L-value
b
R-value
y

Example 1

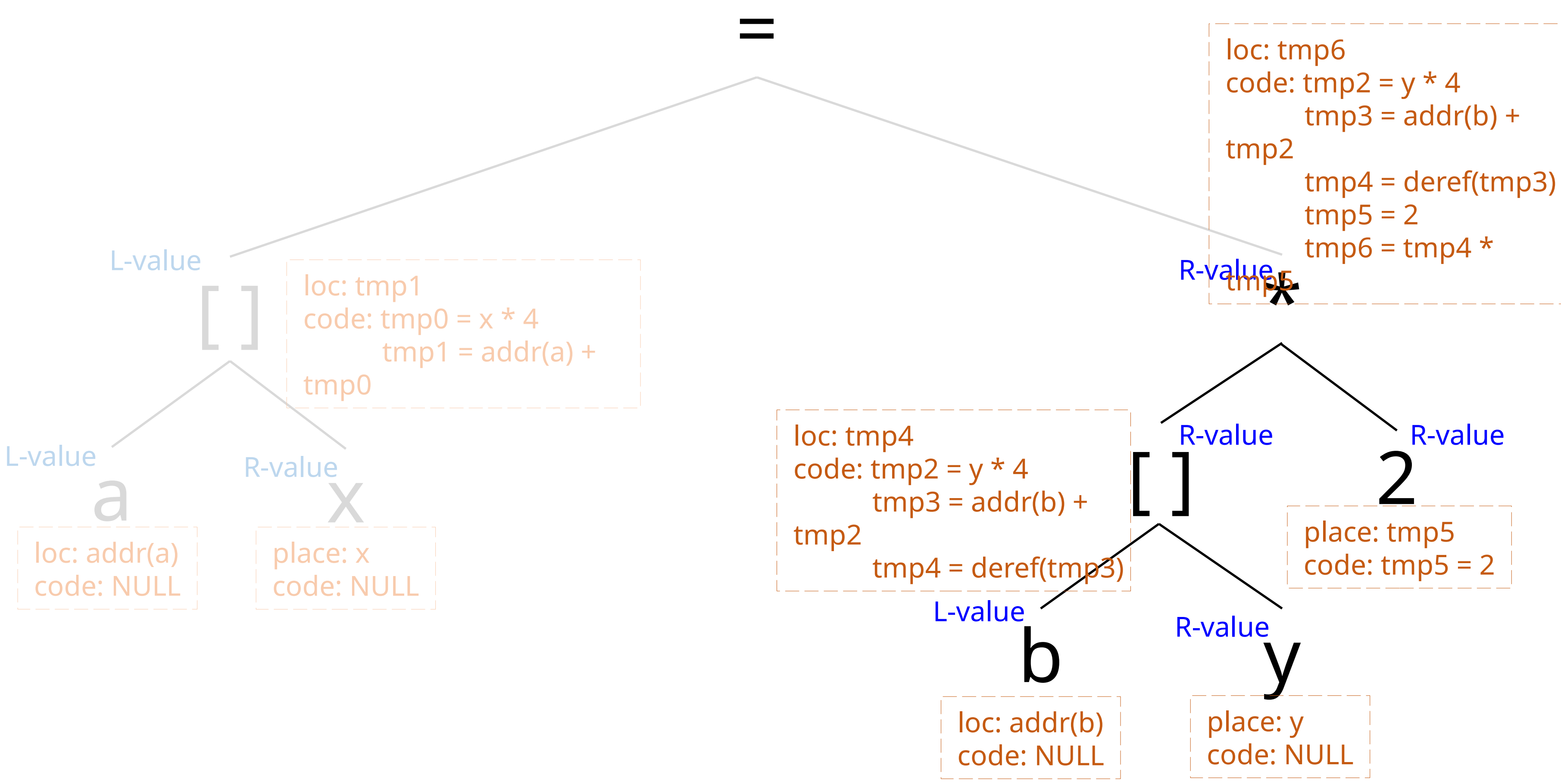


=
L-value
[ ]
loc: tmp1
code: tmp0 = x * 4
tmp1 = addr(a) + tmp0
L-value
a
R-value
x
loc: addr(a)
code: NULL
place: x
code: NULL
R-value
*
R-value
[ ]
R-value
2
L-value
b
R-value
y

Example 1

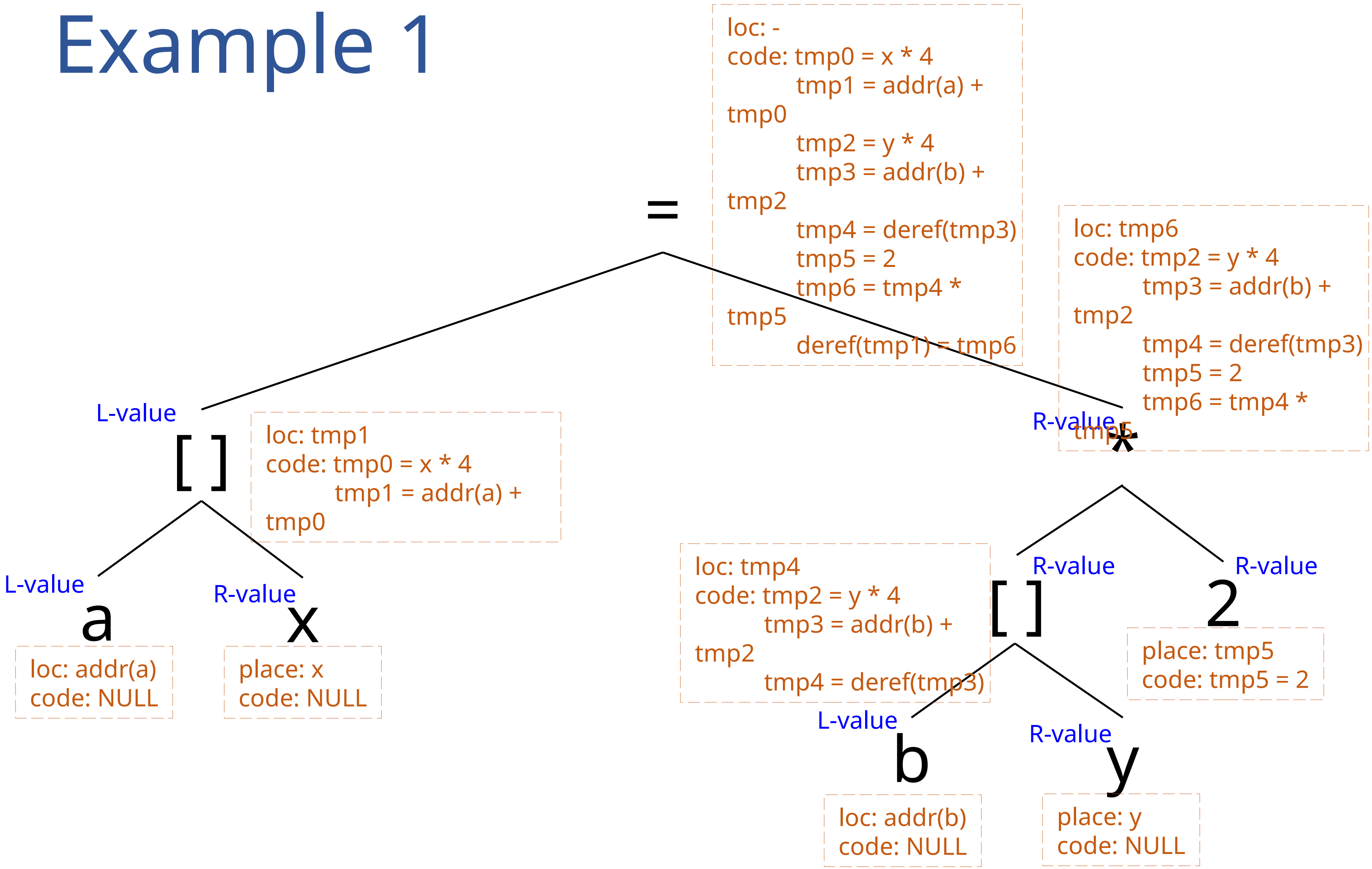

=
L-value
[ ]
loc: tmp1
code: tmp0 = x * 4
tmp1 = addr(a) + tmp0
L-value
a
R-value
x
loc: addr(a)
code: NULL
place: x
code: NULL
R-value
*
dereference the l-value (address) to get the r-value (the value at that location)
loc: tmp4
code: tmp2 = y * 4
tmp3 = addr(b) + tmp2
tmp4 = deref(tmp3)
R-value
[ ]
R-value
2
L-value
b
R-value
y
loc: addr(b)
code: NULL
place: y
code: NULL

Example 1



=
L-value
[ ]
loc: tmp1
code: tmp0 = x * 4
tmp1 = addr(a) + tmp0
L-value
a
R-value
x
loc: addr(a)
code: NULL
place: x
code: NULL
loc: tmp6
code: tmp2 = y * 4
tmp3 = addr(b) + tmp2
tmp4 = deref(tmp3)
tmp5 = 2
tmp6 = tmp4 * tmp5
R-value
*
R-value
[ ]
loc: tmp4
code: tmp2 = y * 4
tmp3 = addr(b) + tmp2
tmp4 = deref(tmp3)
R-value
2
place: tmp5
code: tmp5 = 2
L-value
b
R-value
y
loc: addr(b)
code: NULL
place: y
code: NULL

Example 1



loc: -
code: tmp0 = x * 4
tmp1 = addr(a) + tmp0
tmp2 = y * 4
tmp3 = addr(b) + tmp2
tmp4 = deref(tmp3)
tmp5 = 2
tmp6 = tmp4 * tmp5
deref(tmp1) = tmp6
=
L-value
[ ]
loc: tmp1
code: tmp0 = x * 4
tmp1 = addr(a) + tmp0
L-value
a
loc: addr(a)
code: NULL
R-value
x
place: x
code: NULL
loc: tmp6
code: tmp2 = y * 4
tmp3 = addr(b) + tmp2
tmp4 = deref(tmp3)
tmp5 = 2
tmp6 = tmp4 * tmp5
R-value
*
loc: tmp4
code: tmp2 = y * 4
tmp3 = addr(b) + tmp2
tmp4 = deref(tmp3)
R-value
[ ]
R-value
2
place: tmp5
code: tmp5 = 2
L-value
b
loc: addr(b)
code: NULL
R-value
y
place: y
code: NULL

Example 2

Source code: a[x] = b[y]   (a: int, b: char)

3-address code:

```
tmp1 = y * 1
tmp2 = addr(b) + tmp1
tmp3 = deref(tmp2)

tmp4 = x * 4
tmp5 = addr(a) + tmp4
deref(tmp5) = tmp3
```

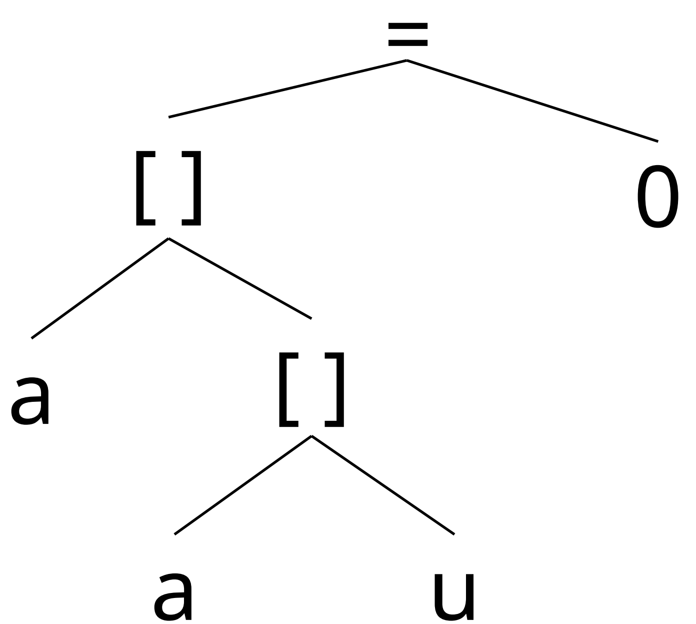
wants l-value

wants r-value

EXERCISE

Source code: a[a[u]] = 0

Work out the 3-addr code at each node:



Multi-way branches (Switch statements)

Multi-way branches: switch statements

- Goal: generate code to choose between a fixed set of alternatives based on the value of an expression
- Implementation Choices:
  - *linear search*
    - best for a small number of case labels ($\approx$ 3 or 4)
    - cost increases with no. of case labels; later cases more expensive.
  - *binary search*
    - best for a moderate number of case labels ($\approx$ 4 – 8)
    - cost increases with no. of case labels.
  - *jump tables*
    - best for large no. of case labels ($\geq$ 8)
    - may take a large amount of space if the labels are not clustered.

Background: Jump Tables

- A jump table is an array of code addresses:
  - *Tbl*[i] is the address of the code to execute if the expression evaluates to i.
  - if the set of case labels have "holes", the correspond jump table entries point to the default case.
- *Bounds checks*:
  - Before indexing into a jump table, we must check that the expression value is within the proper bounds (if not, jump to the default case).
  - The check

    lower_bound $\leq$ *exp_value* $\leq$ *upper bound*

    can be implemented using a single unsigned comparison.

Jump Tables: cont'd

- Given a ***switch*** with max. and min. case labels c_{max} and c_{min}, the jump table is accessed as follows:

Instruction	***Cost*** (cycles)
$t_0 \leftarrow$ value of expression	…
$t_0 = t_0 - c_{min}$	1
if $\neg(t_0 \leq_u c_{max} - c_{min})$ goto *DefaultCase*	4 to 6
t_1 = JmpTbl_BaseAddr	1
t_1 += 4*t_0	1
jmp *t1	3 to 5
	$\boxed{\Sigma: \ 10 \text{ to } 14}$

Jump Tables: Space Costs

- A jump table with max. and min. case labels c_{max} and c_{min} needs $\approx c_{max} - c_{min}$ entries.
  This can be wasteful if the entries aren't "dense enough", e.g.:

```
switch (x) {
  case 1: …
  case 1000: …
  case 1000000: …
}
```

- Define the ***density*** of a set of case labels as

  density = no. of case labels / ($c_{max} - c_{min}$)

- Compilers will not generate a jump table if density below some threshold (typically, 0.5).

Switch Statements: Overall Algorithm

- if no. of case labels is small ($\leq$ ~ 8), use linear or binary search.
  - use no. of case labels to decide between the two.
- if density $\geq$ threshold (~ 0.5) :
  - generate a jump table;

  else :
  - divide the set of case labels into sub-ranges s.t. each sub-range has density $\geq$ threshold;
  - generate code to use binary search to choose amongst the sub-ranges;
  - handle each sub-range recursively.

Code generation for expressions II evaluation order optimization

Evaluation-order optimization

- **Goal**: Choose an evaluation order for the subexpressions of an expression so as to minimize the no. of registers used
- **Algorithm**: Given a syntax tree for an expression:
  1. [**Pass 1**]: Use a postorder traversal to assign a label to each syntax tree node

     The label of a node gives the max. no. of registers needed to evaluate the subexpression rooted at that node.

  2. [**Pass 2**]: Traverse the expression tree and generate code, using node labels to guide which subexpression gets evaluated first

Labeling: *Sethi-Ullman Numbering*

Labeling algorithm:

if n is a leaf node:

$label(n) = 1;$

else:

let the labels for the children of n be l_1, l_2;

$label(n) = (l_1 \neq l_2 \ ? \ max(l_1, l_2) \ : \ l_1+1);$

Evaluation Order: Code Generation

if *n* is a leaf node **:**

locate a free register r, and generate a load into r

else :

let the children of *n* be n_1 and n_2, with labels l_1 and l_2 resp.

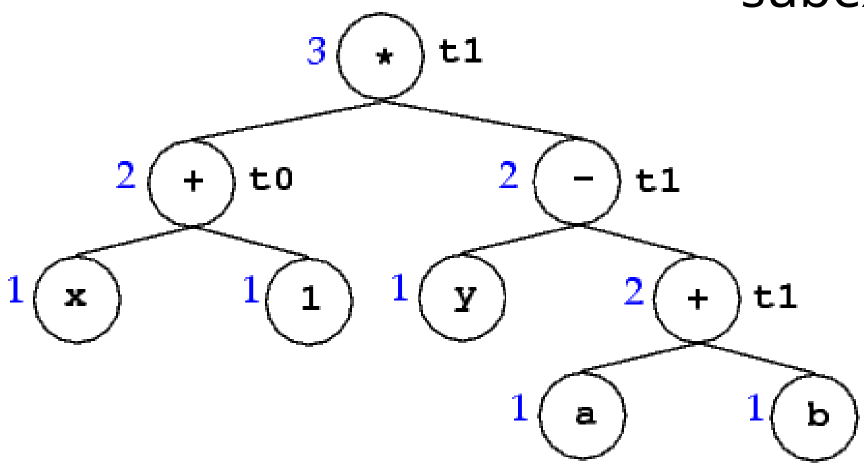
if $l_1 \neq l_2$ **then :** /* needs max(l_1, l_2) registers */

- generate code to evaluate the subexpression with larger label
- free all but one register used by that subexpression
- generate code to evaluate the other subexpression

else : /* needs l_1+1 registers */

- generate code to evaluate n_1 using l_1 registers
- free up l_1-1 registers
- use l_1 registers to evaluate n_2

Example



Code generated (assume that ties are broken in favor of the left subexpression):

r0 = x
r1 = 1
r0 = r0 + r1

r1 = a
r2 = b
r1 = r1 + r2

r2 = y
r1 = r2 - r1

r1 = r0 + r1

Comments on Evaluation Order Algorithm:

- O(n) in the size of the expression tree.
- Easily adapted to the case where leaf node variables don't have to be loaded into registers.
- If there are no common subexpressions, the algorithm is provably optimal in terms of no. of registers used.
  - Optimal code generation for expression DAGs is NP-complete.
  - The algorithm can be adapted to handle DAGs.

    This produces usually good code that is not necessarily optimal.