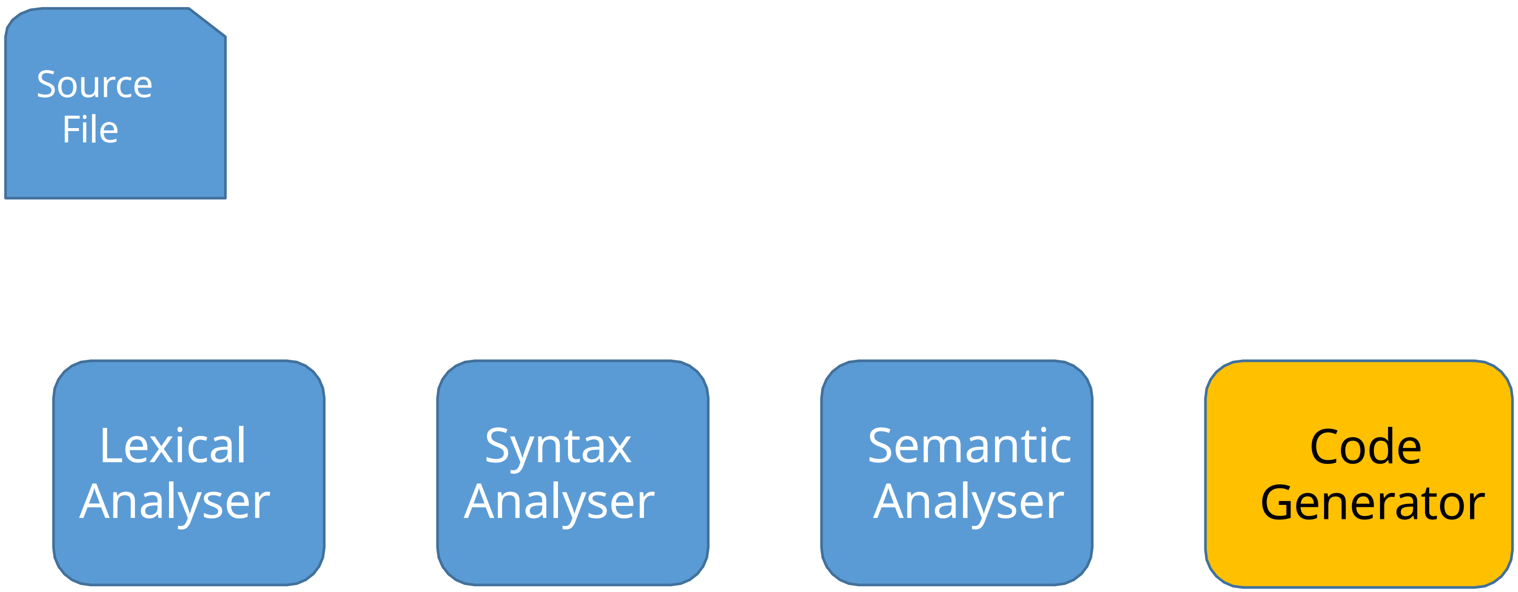
Unit 12: Code Generation and TAC : Three Address Code

A Structure Reminder



ELF Binaries

(Linux binaries look like this)

Actual program code

Read-only Data
(strings,constants)

ELF Header
Program Header Table
.text
.rodata
.data
Section Header Table

Non-Stack Variables

libzip.so

TAC : Three Address Code

- Three-address code (TAC) is a form of representing intermediate code used by compilers.
- Each instruction in three-address code can be described as a 4-tuple (also occasionally known as a 'quad'):

(operator, operand1, operand2, result).

(three addresses)

operator	operand1	operand2	result
op	**y**	**z**	**x**

- Each statement has the general form of:

  `x := y op z`

  where x, y and z are variables, constants or temporary variables generated by the compiler.
- op represents any operator, e.g. an arithmetic operator.

operator	operand1	operand2	result
op	**y**	**z**	**x**

The TAC Instructions

- Binary operators
  - `a := a + b`
  - `a := a - b`
  - `a := a * b`
  - `a := a / b`
- Unary operators
  - `a := -b`
- Assignment
  - `a := b`

These are essentially abstract machine instructions.

Closer to the actual machine code than the original source.

- Jump instructions
  - `goto a`
  - `ifzero b goto a`
  - `ifnotzero b goto a`
- Subroutine call
  - `arg a`
  - `a := call b`
  - `return a`
- Pseudo instructions
  - `var a`
  - `label a`

- Expressions containing more than one fundamental operation, such as:

```
p := x + y * z
```

  are not representable in three- address code as a single instruction.

- Instead, they are decomposed into an equivalent series of instructions, such as

```
t1 := y * z
p := x + t1
```

- Note the introduction of a temporary or intermediate variable.

- The term three-address code is still used even if some instructions use more or fewer than two operands.
  - The classic case for this is the no-operation... operation. NOP, which has no operands at all.
  - This has the useful effect that the read operations for intermediate code in this format are all the same length

- The key features of three-address code are that
  - every instruction implements exactly one fundamental operation
  - the source and destination may refer to any available register / memory location

Example

- The following C program, translated into three-address code, might look something like that shown on the following slide:

```
int main (void)
{
    int i;
    int b[12];
    i = 0;
    while (i < 12) {
        b[i] = 4*i;
        ++i;
    }
}
```

```
      i  := 0                     ; assignment
L1:   t3  := 12 - i
      ifnotzero t3 goto L2        ; conditional jump
      goto L3                     ; unconditional jump
L2:   t0  := 4*i
      t1  := &b                   ; address-of operation
      t2  := t1 + i               ; t2 holds the address
                                  of b[i]
      *t2  := t0                  ; store through
                                  pointer
      i  := i + 1
      goto L1
L3:
```

- int main (void)
  {
        int i;
-       int b[12];
-       i = 0;
-       while (i < 12)
         {
-           b[i] = 4*i;
-           ++i;
-       }
- }

```
    i := 0
L1: t3 = 12 - i
    ifnotzero t3 goto L2
    goto L3
L2: t0 := 4*i
    t1 := &b
    t2 := t1 + i
    *t2 := t0
    i := i + 1
    goto L1
L3:
```

Loop Control

Loop Body

Building Sequences of Code

- As we recognise expressions, we build the code that implements those expressions at run-time.

- Say we have recognised the following:
  - `expr ::= expr PLUS term`

- Which has matched the input stream
  - `b * c + d * 3`

b * c + d * 3

- Previously, when recognising “b * c”, we would have built the code that implements it (call this code A).

- Similarly, when recognising “d * 3”, we would have built that code (call this code B).

Code A
b * c

Code B
d * 3

b * c + d * 3

- We could rewrite the expression as :
  - `y = b * c`
  - `z = d * 3`
- And then realise the expression as
  - `x = y + z`.
- But what are x, y and z?

Code A
b * c

Code B
d * 3

b * c + d * 3

- y is the location of the result of Code A (b*c).
- z is the location of the result of Code B (d*3).
- So, as well as the code blocks A and B, we need to know where the code puts the result(s).

Code A **b * c**	**Code B** **d * 3**
result **in y**	**result** **in z**

What about x (, y and z)?

- This needs to be a temporary location to store an intermediate result, until we know what to do with it

- The parser is going to generate temporary locations as it generates the code

- Later, in the code generation phase, the parser will allocate it to a real store location (or to a register, if appropriate)

expr ::=	**expr**	**PLUS**	**term**
	b * c		**d * 3**
	$\boxed{\texttt{code A}}$		$\boxed{\texttt{code B}}$
	$\boxed{\texttt{result in y}}$	**+**	$\boxed{\texttt{result in z}}$

- We would generate :

`code A`
`code B`

`generate temp. variable Tn;`
`generate code to calculate`
`Tn = y + z`

`result in Tn`

Example 2

- Now suppose the parser has recognised the rule:

expr ::= expr PLUS term

- We want to generate something like:

a + 3

```
Generate temp. variable Tn;
generate code to calculate
Tn := a + 3
```

```
result in Tn
```

Example 2

- So we generate the following for “a” (and “3” similarly) :
  - $\boxed{}$ **no code**

    $\boxed{\texttt{result in a}}$

- It’s not really a “result” as there is no expression associated with it.
- It’s a reference to the “address of a” in the symbol table.
- In the case of “3”, it’s a reference to the “address of 3” in the literal table.

The Literal Table

- The Literal Table is similar to the Symbol Table, except its entries are any literal representations of values that appear in the input stream.
- For example, numeric literals such as integer literals (“3”,”109”) or float/real literals (“3.14” or other notations for floating point numbers).
- Or character literals (‘a’, ‘1’)
- Or string literals (“hello, world!”).

The Literal Table

- By holding these in a table, the compiler can reference them using their address in the table, rather than use the literal directly.
  - The same way it references symbols via their address in the symbol table.
  - But see possible optimisations later in this unit

- Some of these tables can be populated by the syntax analyser
  - String literals, for example whenever it encounters a constant string
    - Beyond lookup, this table can also be useful later for generating the static value block in the output binary (.rodata in ELF)

- Others can be 'populated' as a function
  - **`integerTable = func(key) => index;`**
  - **`floatTable = func(key) => key;`**

Code and Result (C/R) Blocks

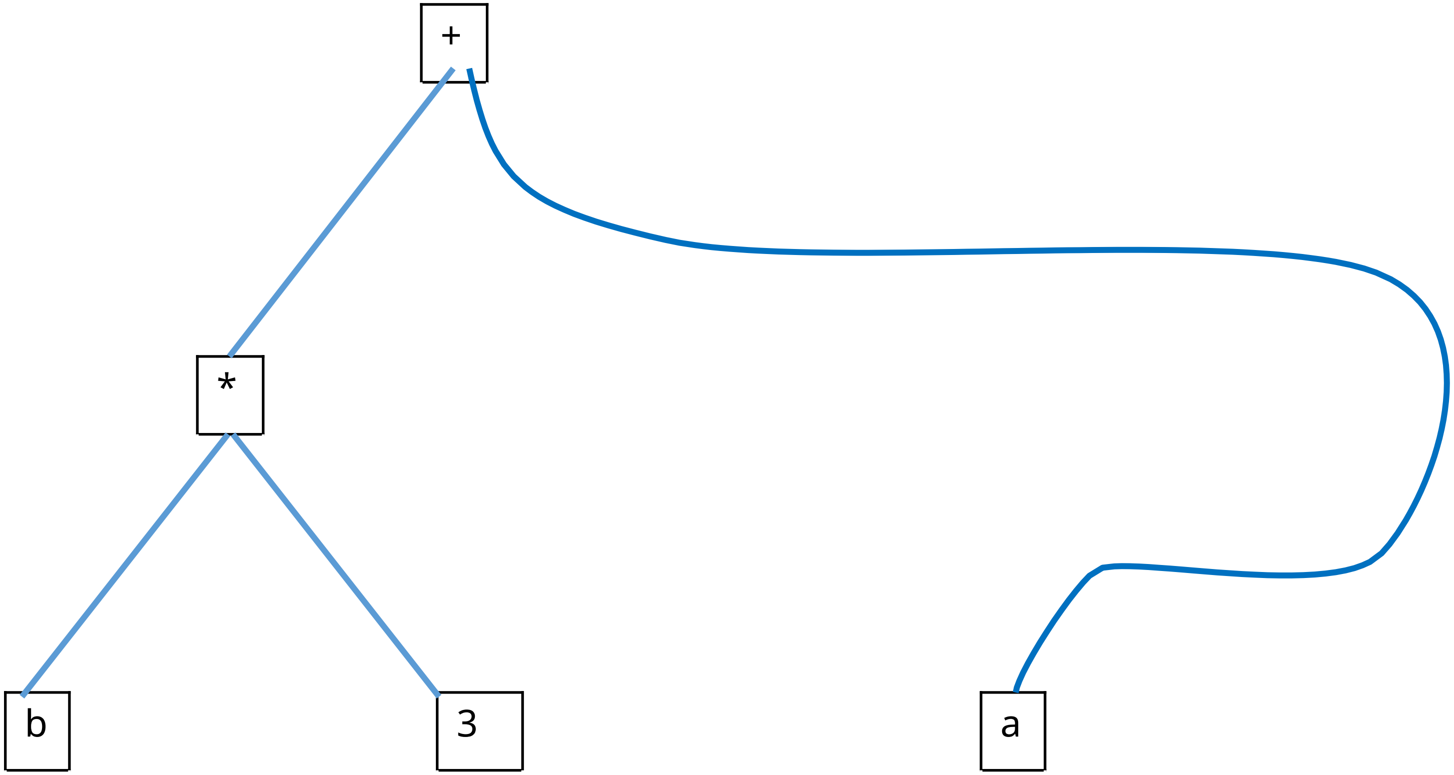
expr ::= expr PLUS term

b * c **d * 3**

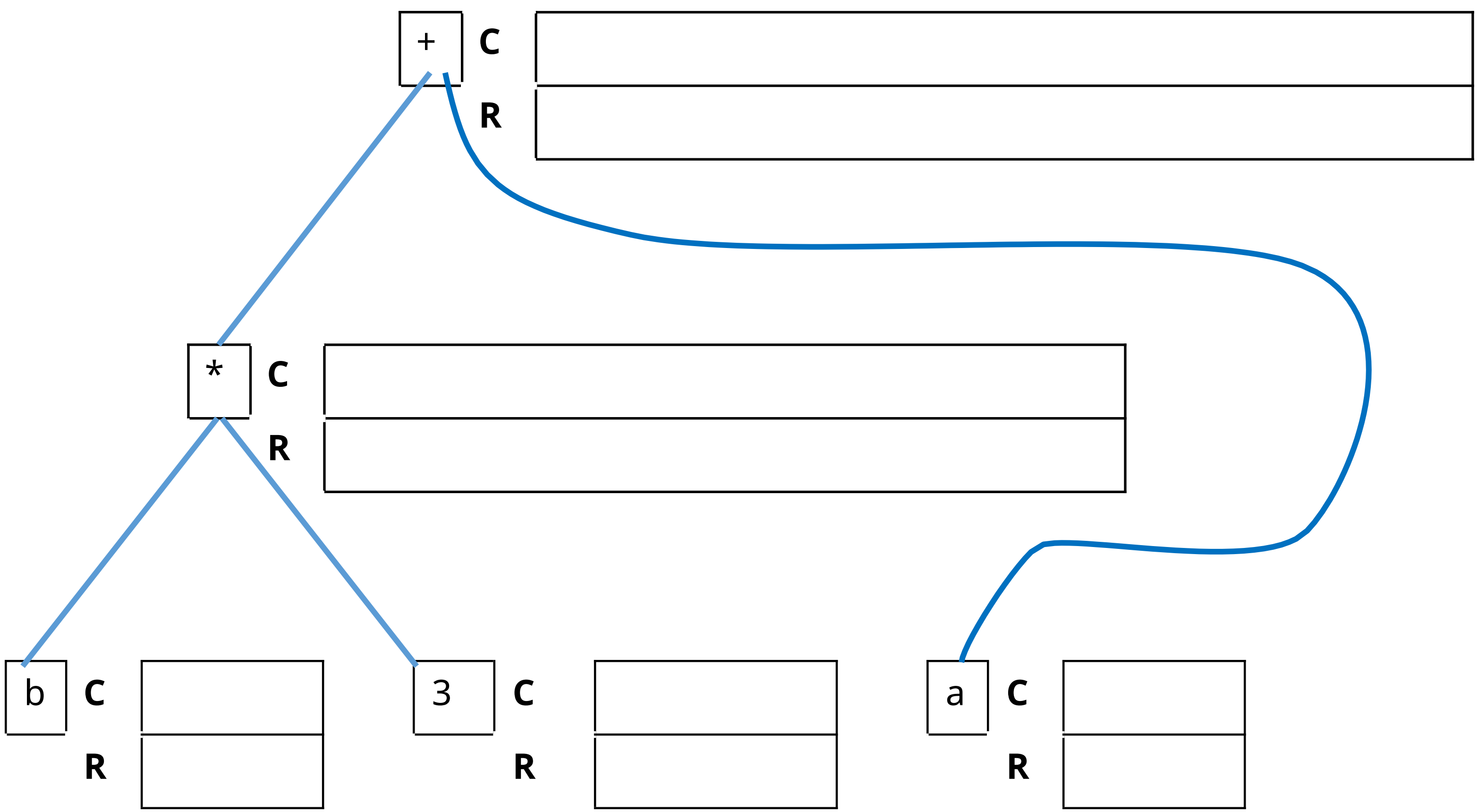
	code A			code B
C	**code A**		**C**	**code B**
R	**result in y**	**+**	**R**	**result in z**

A Worked Example : b * 3 + a

The Parse Tree : b * 3 + a



Parse Tree showing C/R blocks



Code/Result block for “b”

- To save space in the diagram, we introduce a notation : st[b] means the address of b in the symbol table.

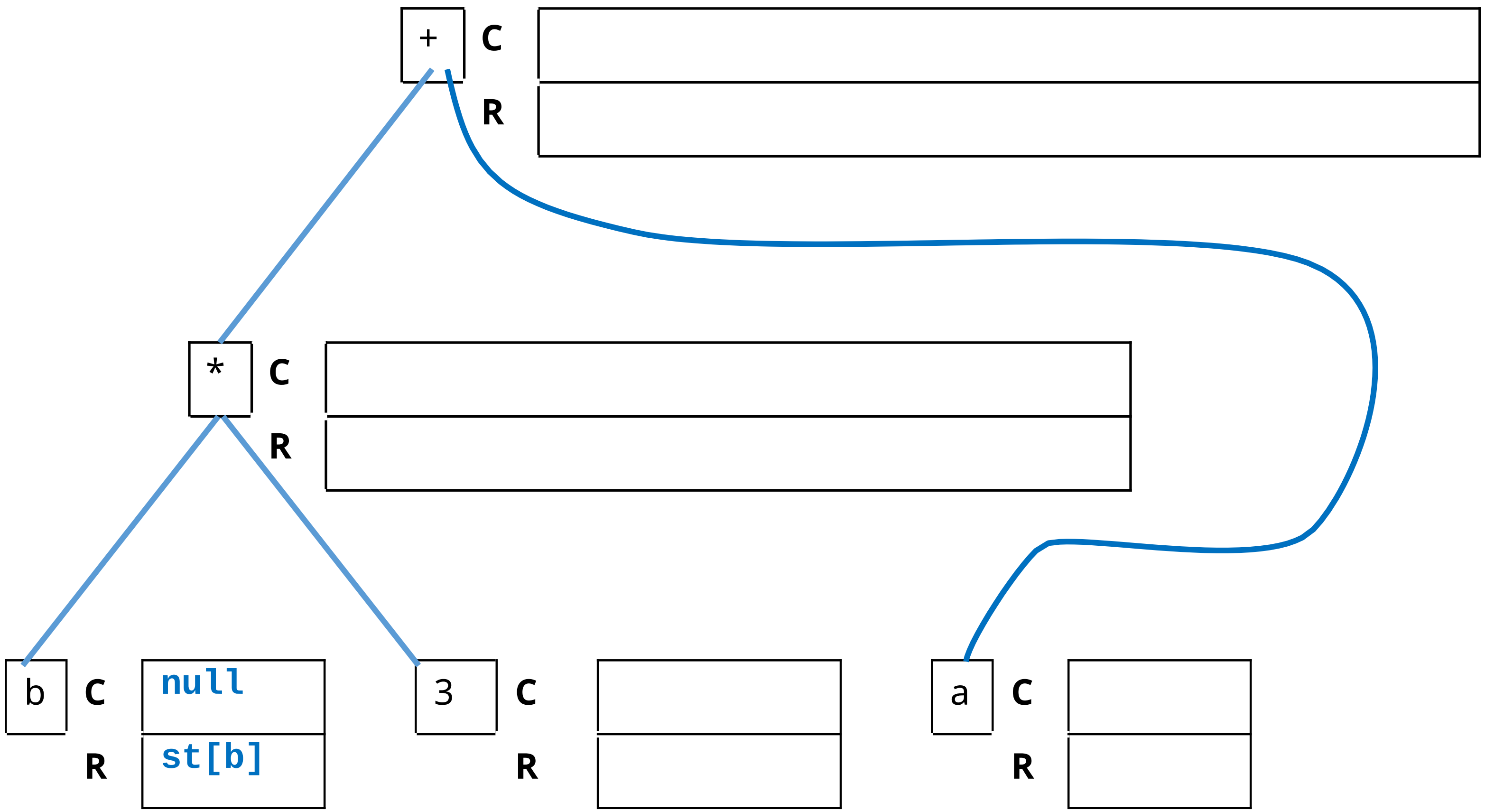
C	
R	address of b in symbol table

no code

C	null
R	st[b]

no code

So far …



C/R block for “3”

- To save space in the diagram, we introduce a notation : lt[3] means the address of ‘3’ in the literal table.

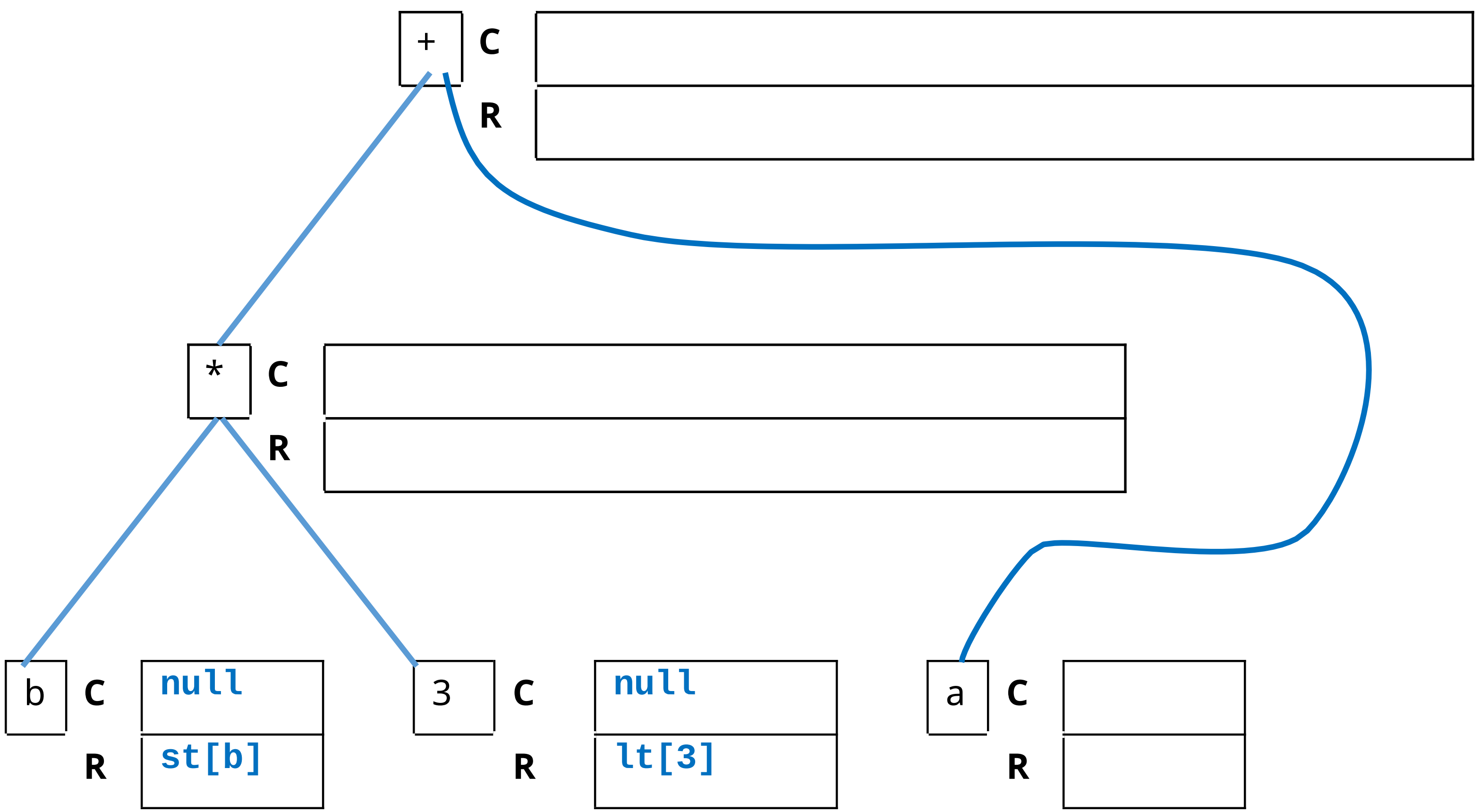
C	
R	address of 3 in literal table

no code

C	null
R	lt[3]

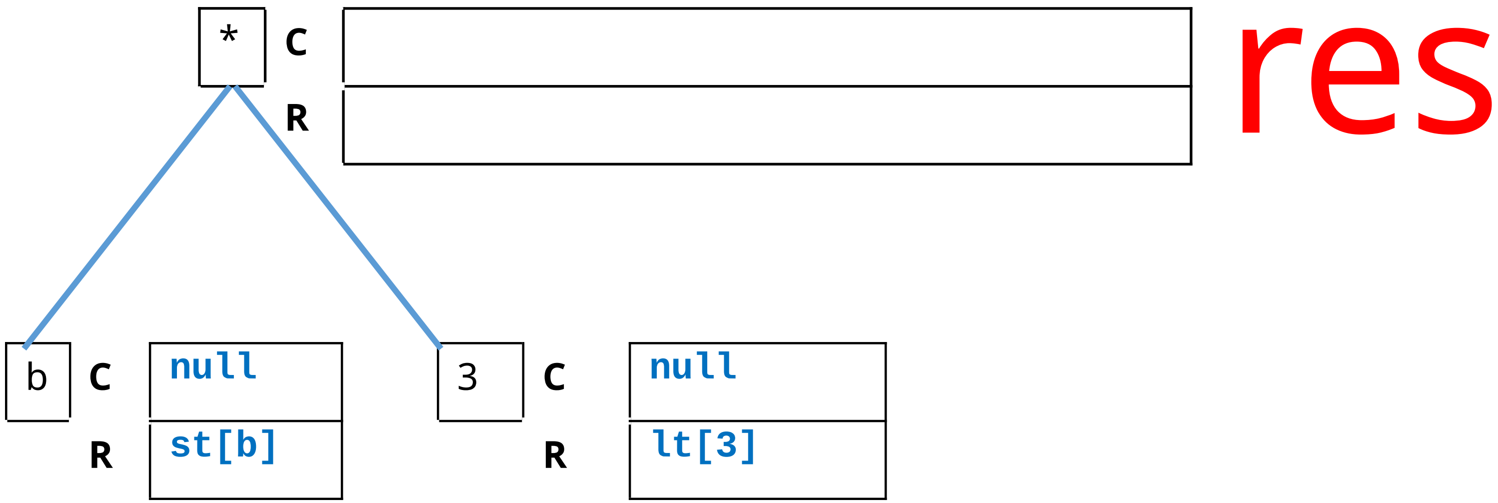
no code

So far …



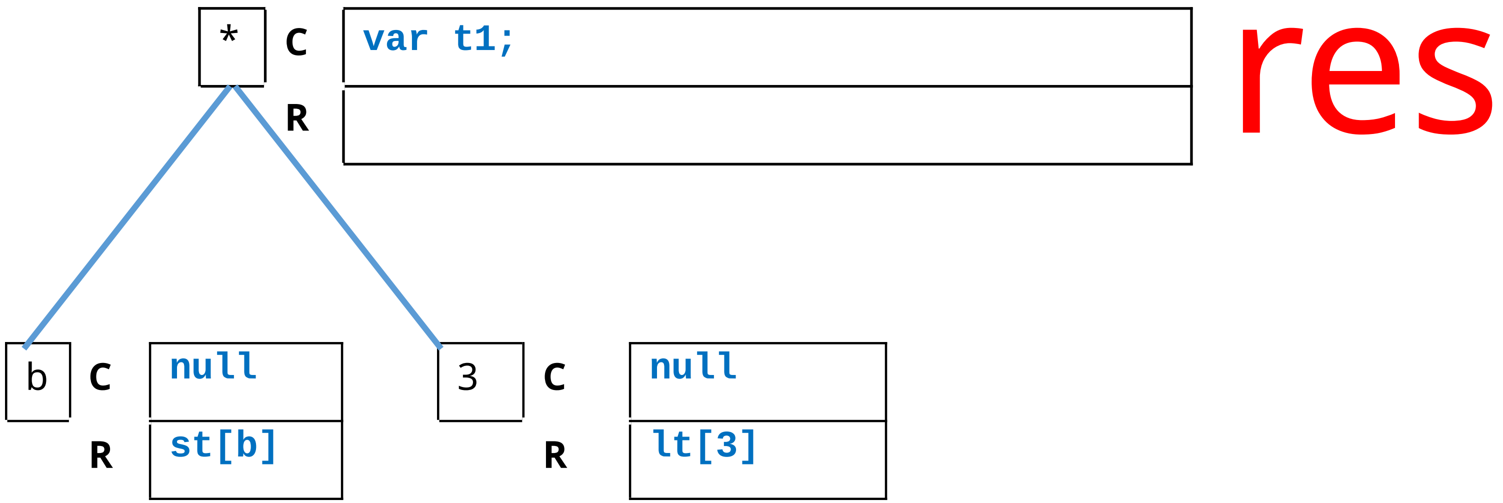
The algorithm for generating code for a binary operator

- To create the code/result block for the ‘*’ node, we
- a) create new c/r block to be the result (res).
- b) copy the code elements of the participating c/r blocks into res.code. In this case, it would be two null entries (hence nothing).



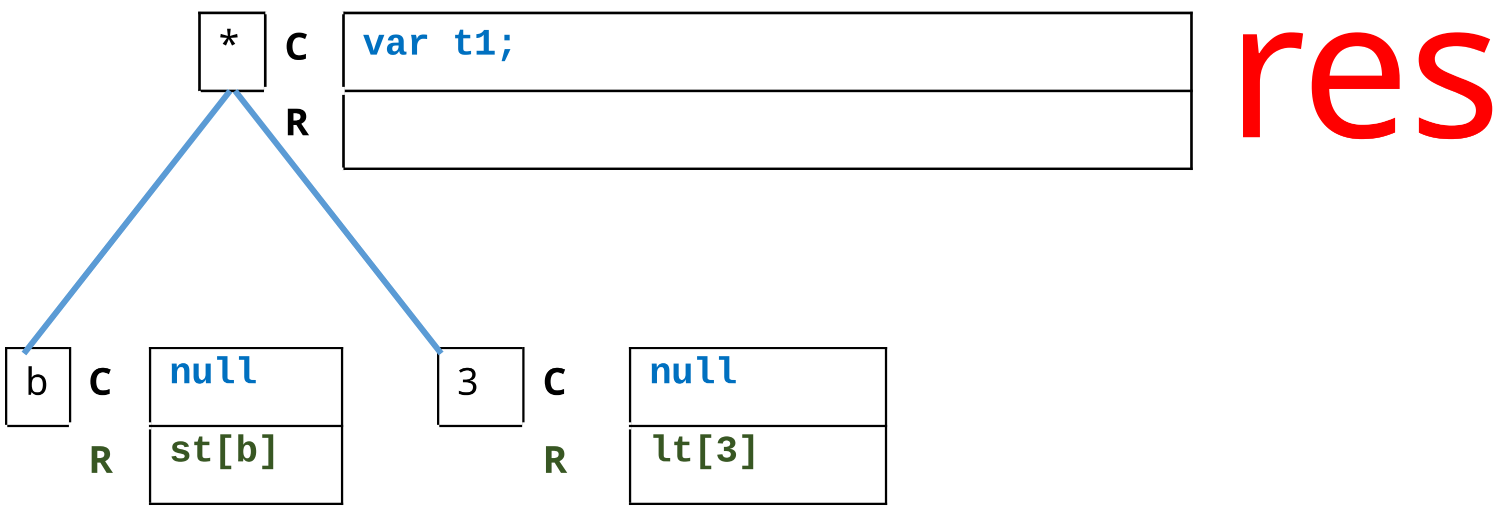
- c) generate a new “temporary” variable (call it **`t1`**).

- d) Add the TAC instruction that declares the temp. variable (“**`var t1`**”) into res.code. It now contains that single TAC declaration instruction.



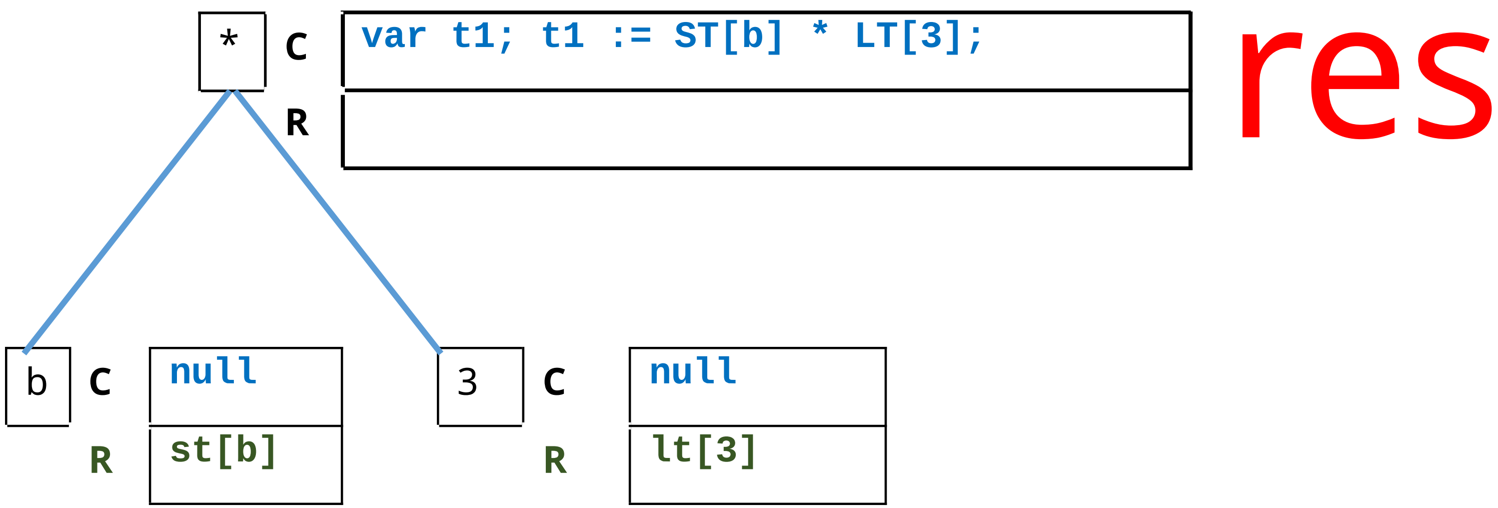
The algorithm for generating code for a binary operator

- e) Add the TAC instruction that implements the ‘*’ operation to res.code. To construct this, we find out the names of the operands by looking at the result field of the participating C/R blocks.



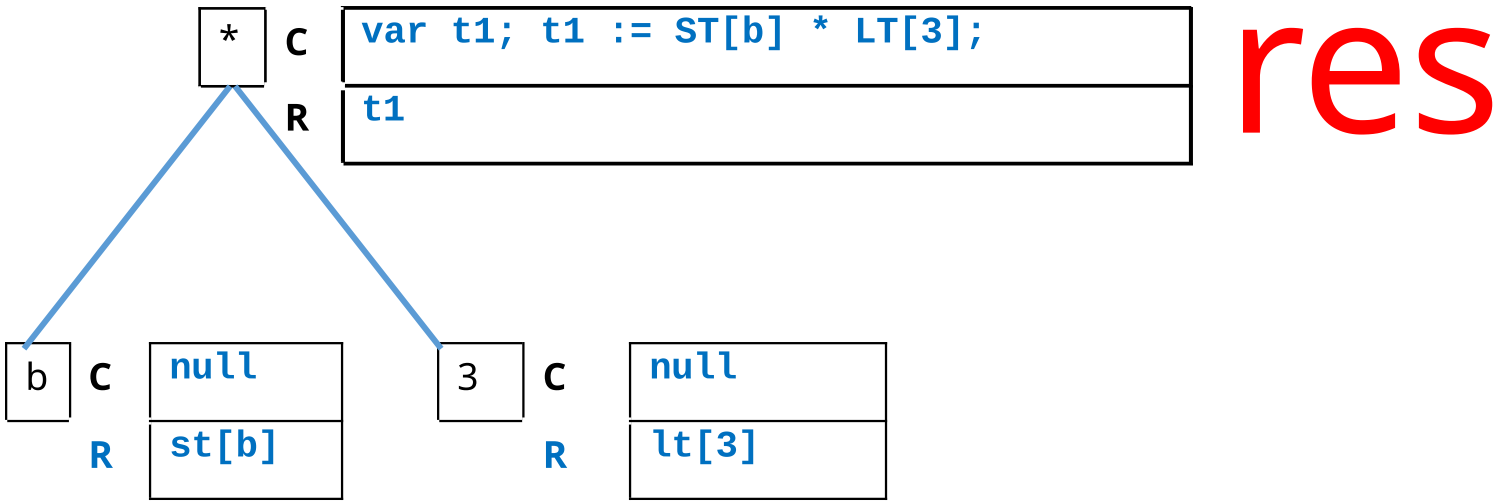
The algorithm for generating code for a binary operator

- The new instruction is:
  "**`t1 := ST[b] * LT[3]`** ".
- Add this to res.code, which now contains "**`var t1; t1 := ST[b] * LT[3]`**".

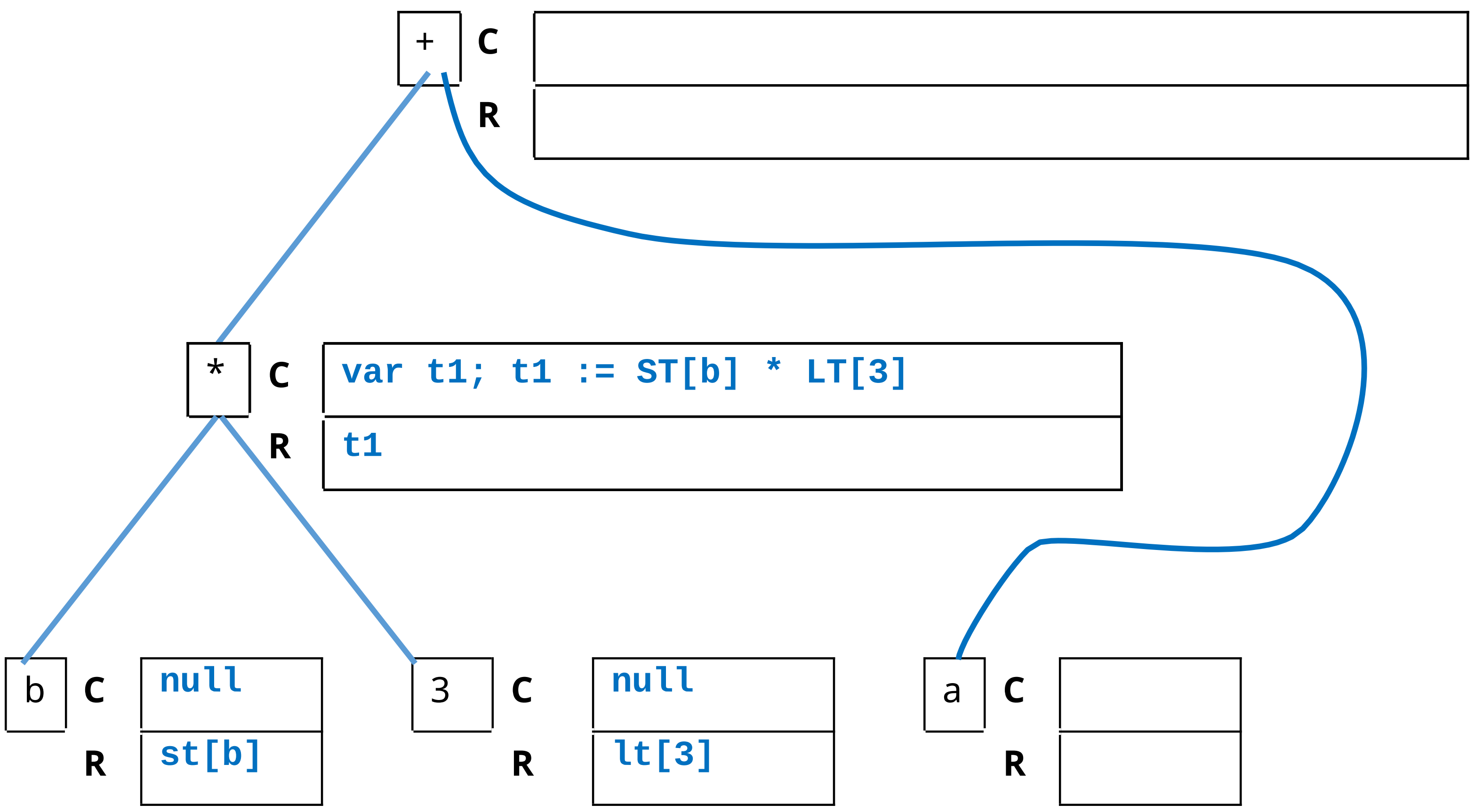


The algorithm for generating code for a binary operator

- f) set res.result to "**t1**". (The temp. variable t1 holds the result of res.code).



So Far …



C/R block for “a”

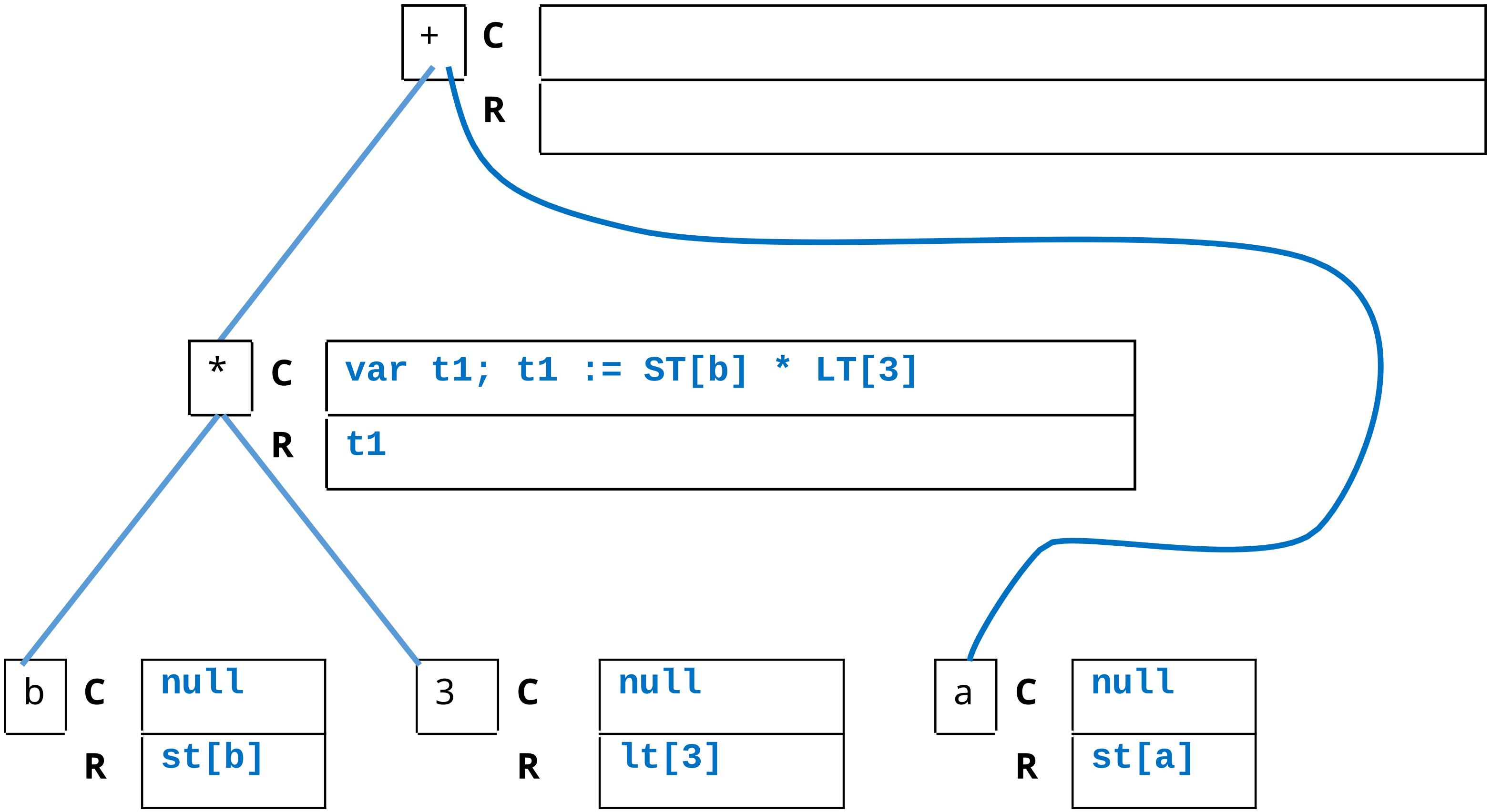
C	
R	address of a in symbol table

no code

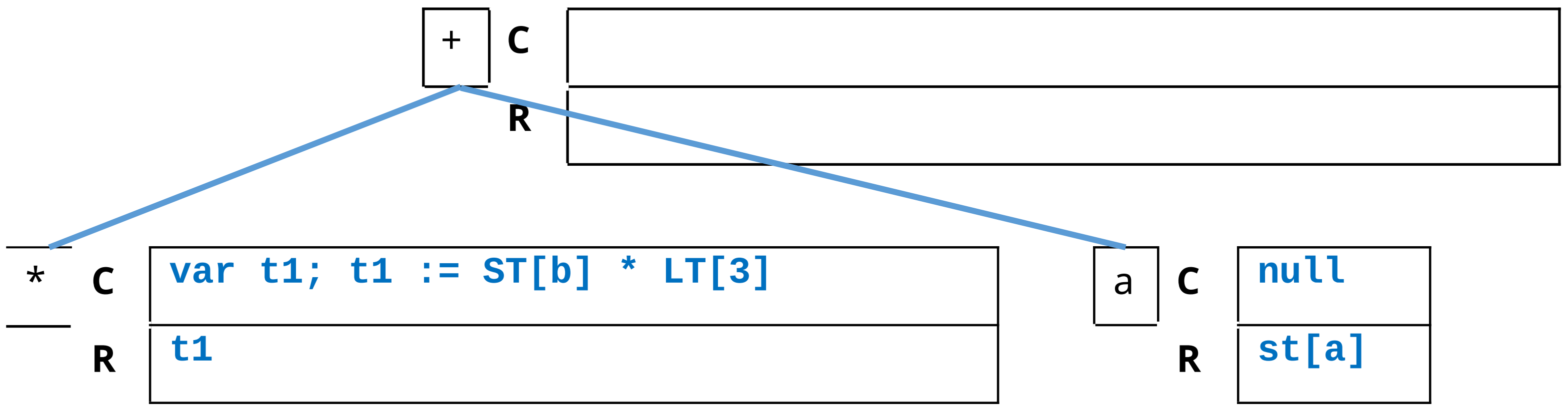
C	null
R	st[a]

no code

So Far …

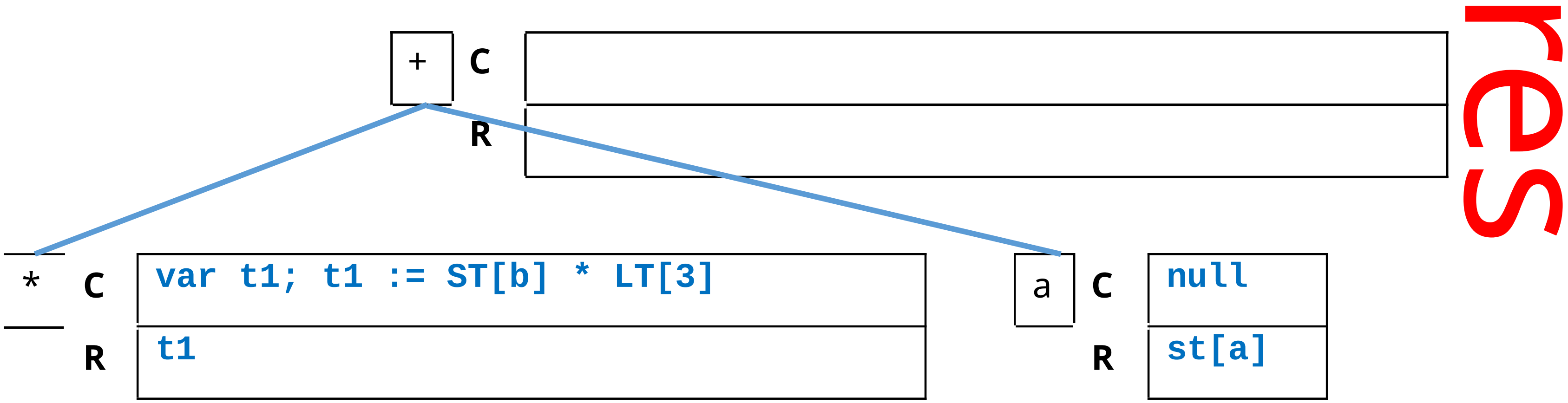


Next Step …



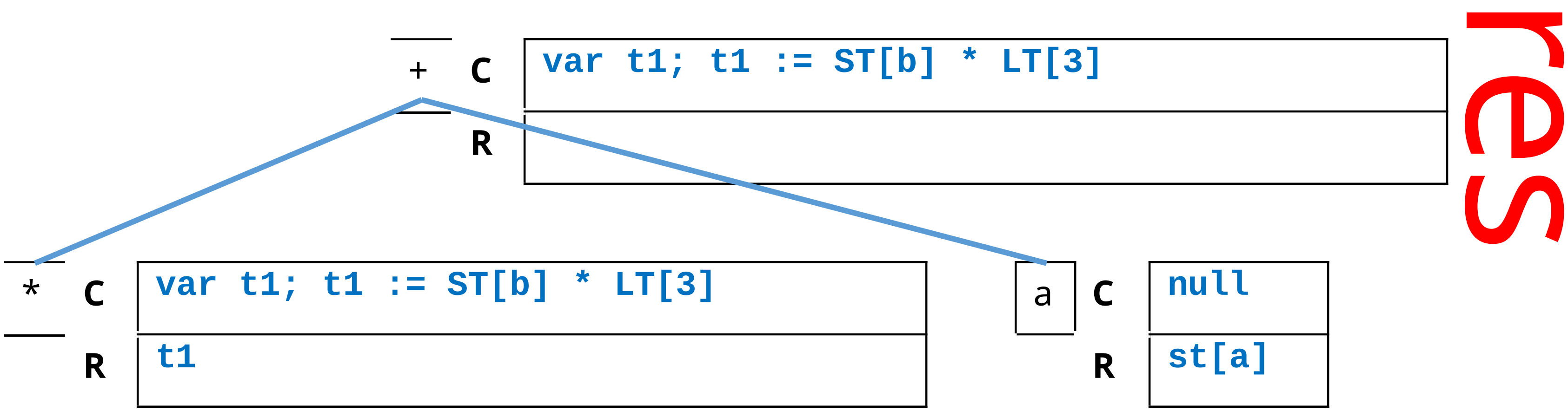
Apply the bin-op algorithm again.

- To create the code/result block for the '+' node, we
- a) create new c/r block to be the result (res).



Apply the bin-op algorithm again.

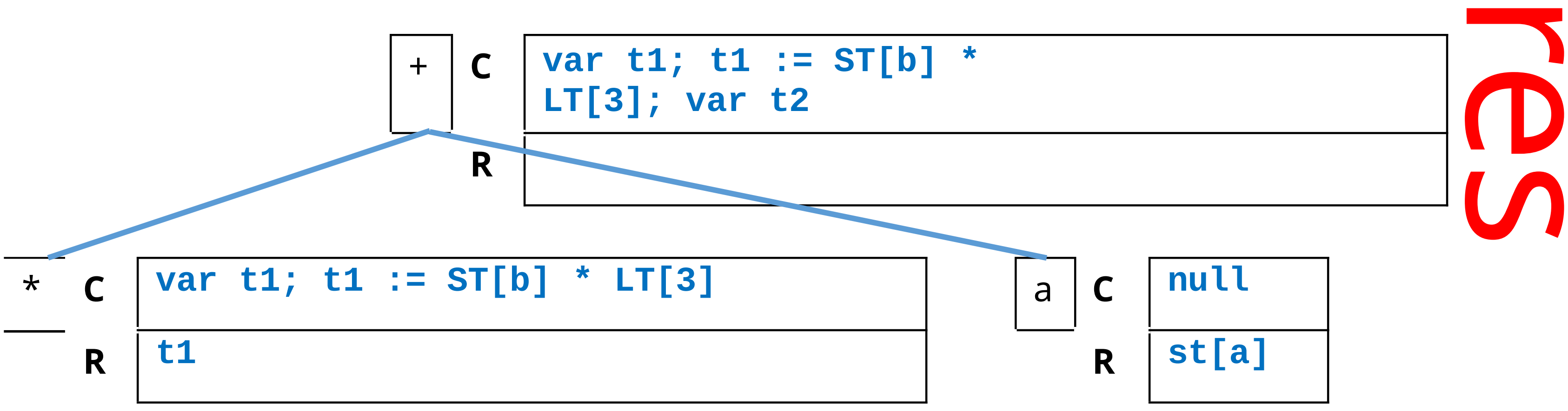
- b) copy the code elements of the participating c/r blocks into res.code. In this case, it would be **`var t1; t1 := ST[b] * LT[3]`** plus null (nothing).



Apply the bin-op algorithm again.

- c) generate a new "temporary" variable (call it **`t2`**).

- d) Add the TAC instruction that declares the temp. variable ("**`var t2`**") into res.code. It now contains "**`var t1; t1 := ST[b] * LT[3]; var t2`**"



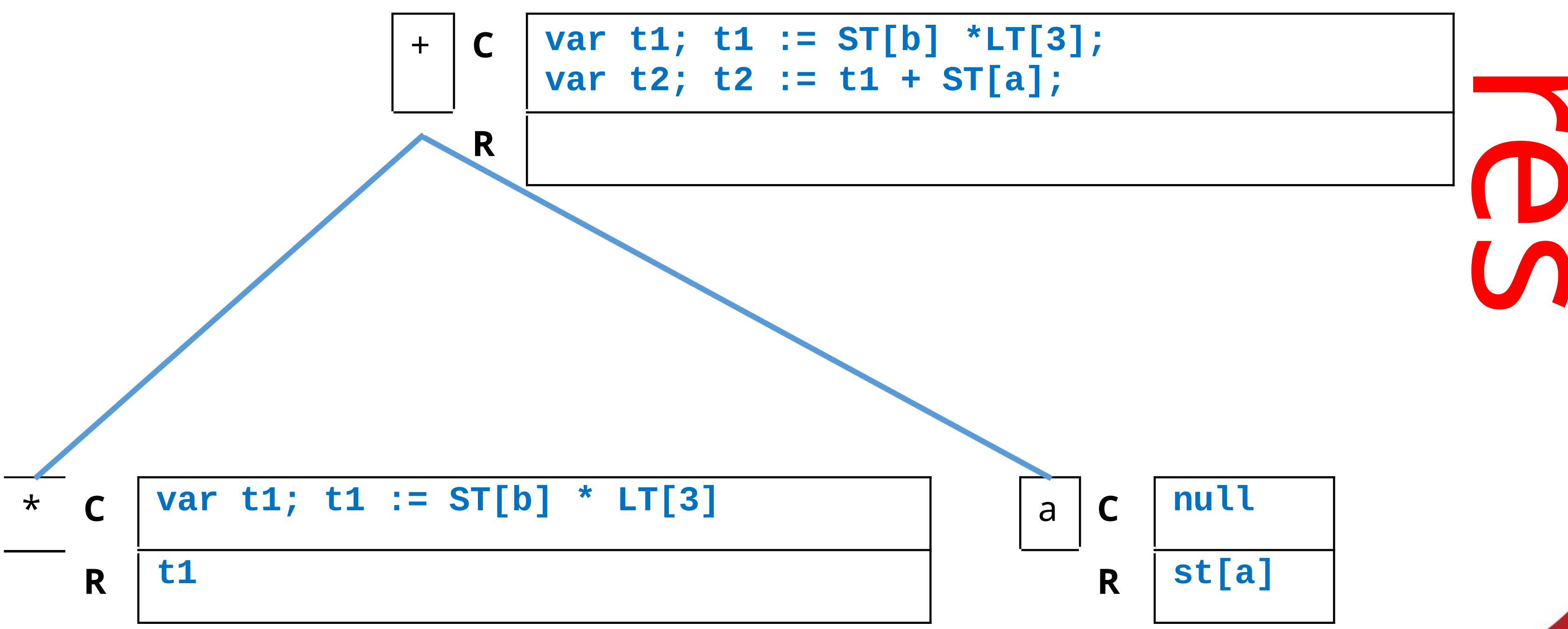
Apply the bin-op algorithm again.

- e) Add the TAC instruction that implements the ‘+’ operation to res.code. To construct this, we find out the names of the operands by looking at the result field of the participating C/R blocks.

- The new instruction is: “`t2 := t1 + ST[a]` ”.

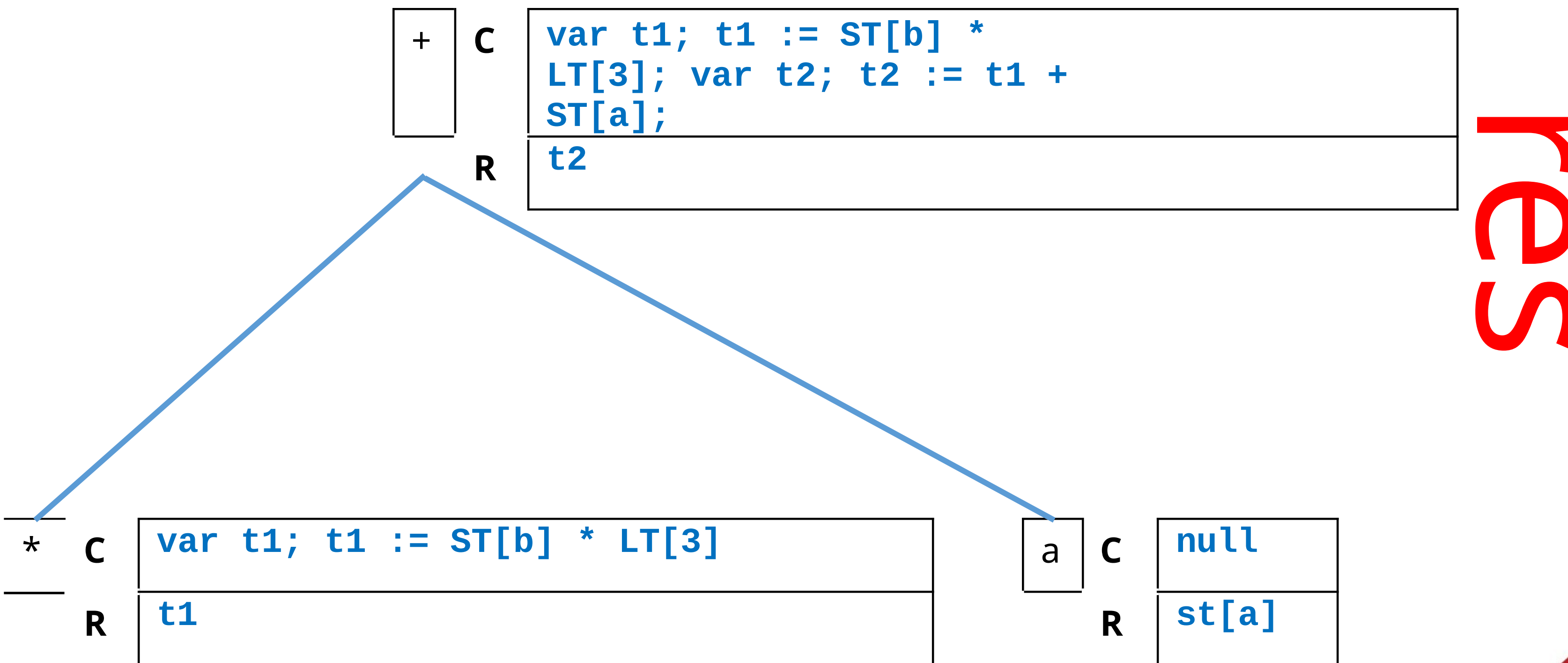
Apply the bin-op algorithm again.

- Add this to res.code, which now contains "**`var t1; t1 := ST[b] * LT[3]; var t2; t2 := t1 + ST[a]`** "

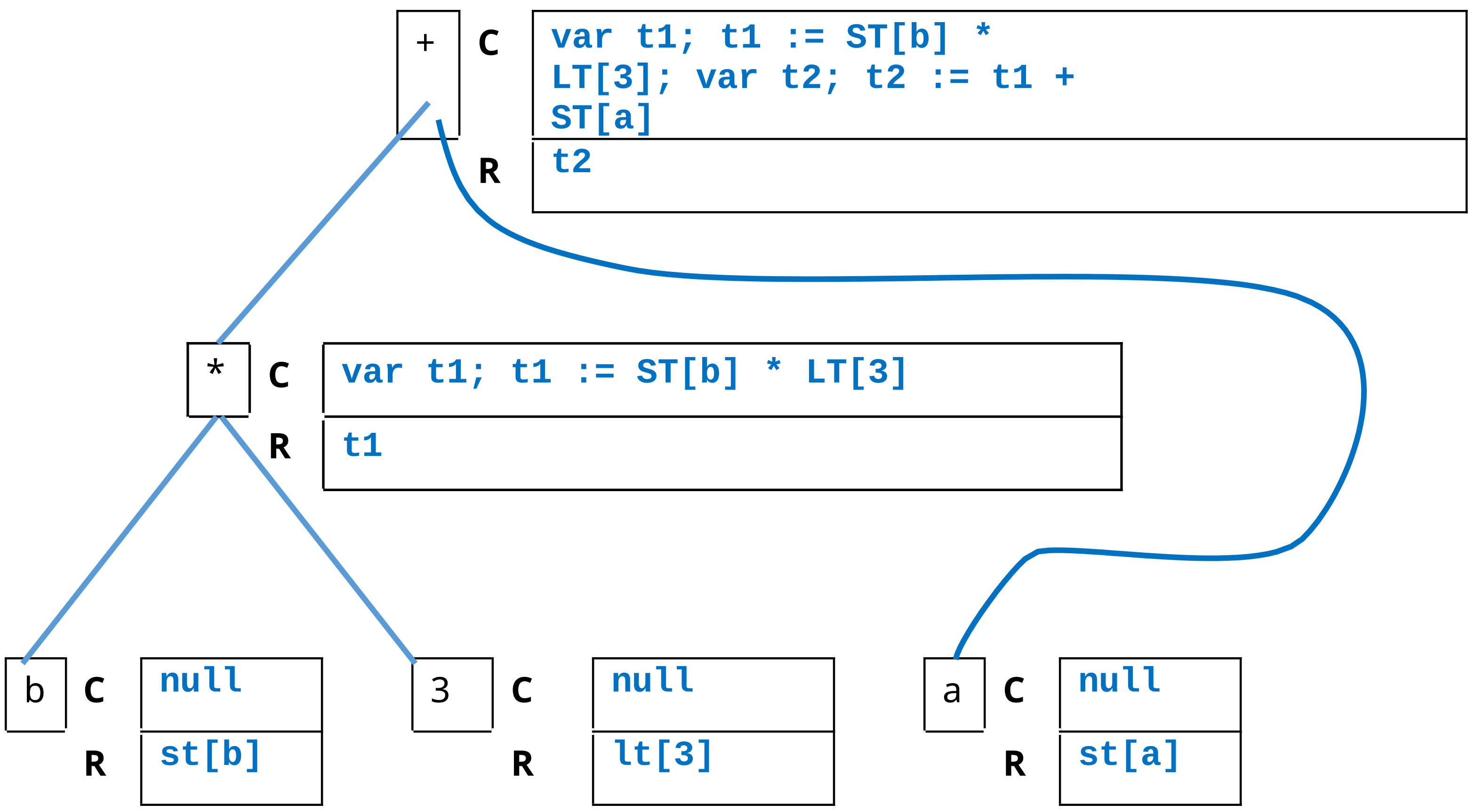


Apply the bin-op algorithm again.

- f) set res.result to “**t2**”. (The temp. variable t2 holds the result of res.code).



Finished



+
C
var t1; t1 := ST[b] * LT[3]; var t2; t2 := t1 + ST[a]
R
t2
*
C
var t1; t1 := ST[b] * LT[3]
R
t1
b
C
null
R
st[b]
3
C
null
R
lt[3]
a
C
null
R
st[a]

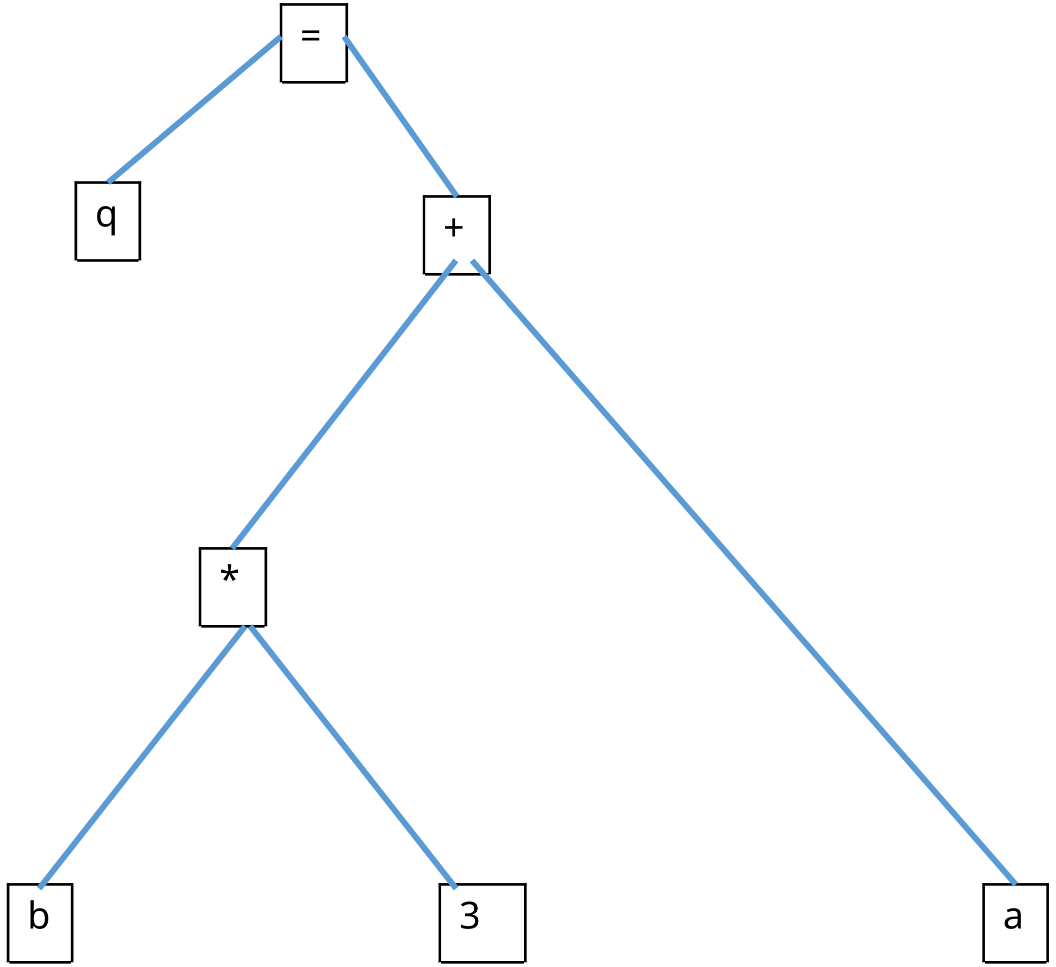
Finished

- We have now generated the TAC code that implements the expression.
- It is all in the C/R block at the root of the parse tree.
- ```
  var t1;
  t1 := ST[b] * LT[3];
  var t2;
  t2 := t1 + ST[a]
  ```
  ```

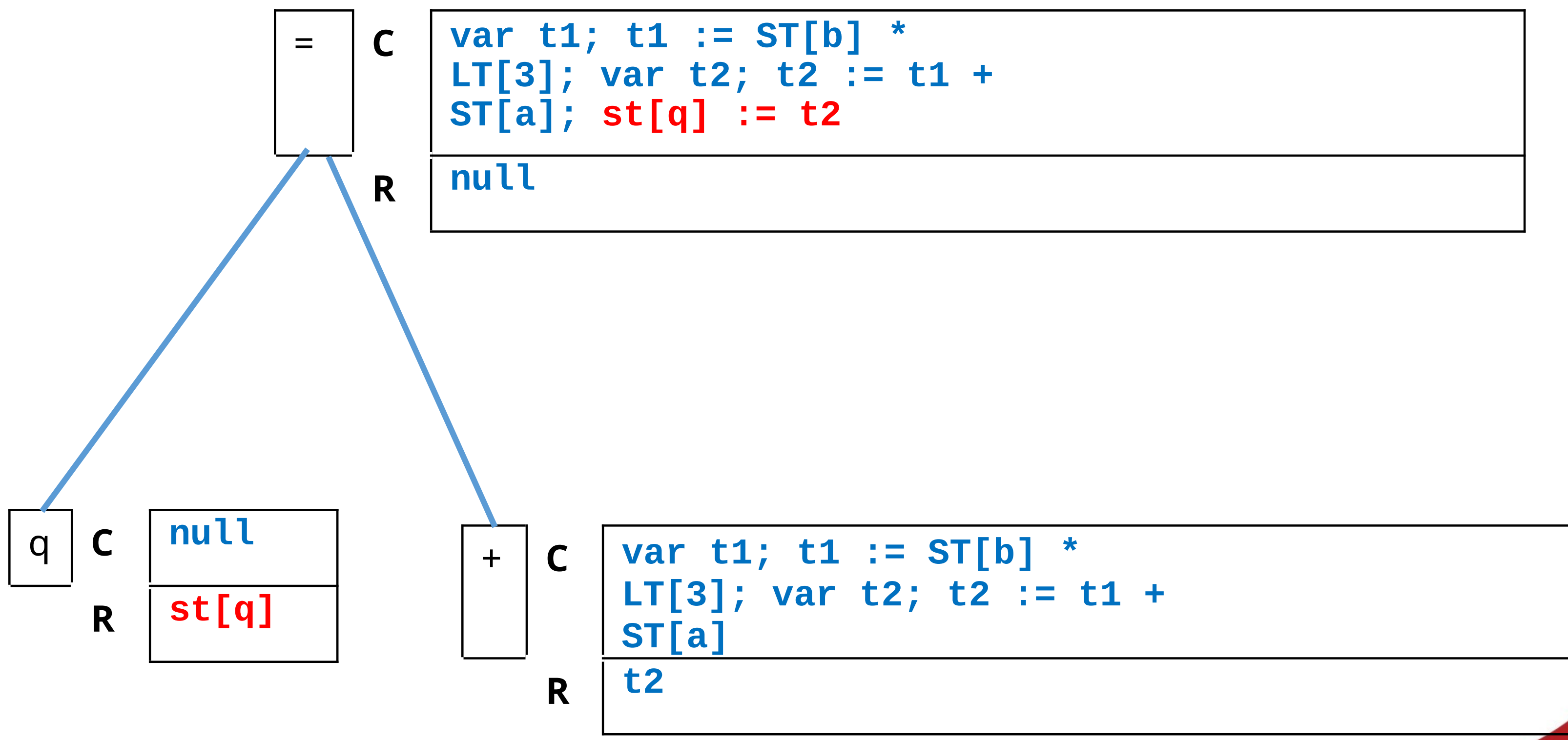
One more step in the example

- This is the expression we processed : b * 3 + a
- Let’s make this the RHS of an assignment statement.
- q = b * 3 + a

The Parse Tree : q = b * 3 + a



- Let’s look at the parse tree after processing of the RHS expression is complete.



Improving the Code

- We could
  - Store the literal **3** directly (instead of the reference via the literal table)
  - Remove the temporary variable T2 and store the result of the addition directly in "q"

=	C	var t1; t1 := ST[b] * LT[3]; var t2; t2 := t1 + ST[a]; st[q] := t2
	R	null

=	C	var t1; t1 := ST[b] * 3; st[q] := t1 + ST[a];
	R	null

Optimisation Phase

- These improvements are part of the optimisation phase
  - These improvements are independent of the specific machine code generated
  - There could be further optimization when the parser inspects the actual code generated

- One of the advantages of the TAC representation is that it is easier to optimize than the parse tree representation

Learning Outcomes

- You should now have an understanding of how code can be generated by the compiler …