Checked C: Adding Memory Safety to LLVM

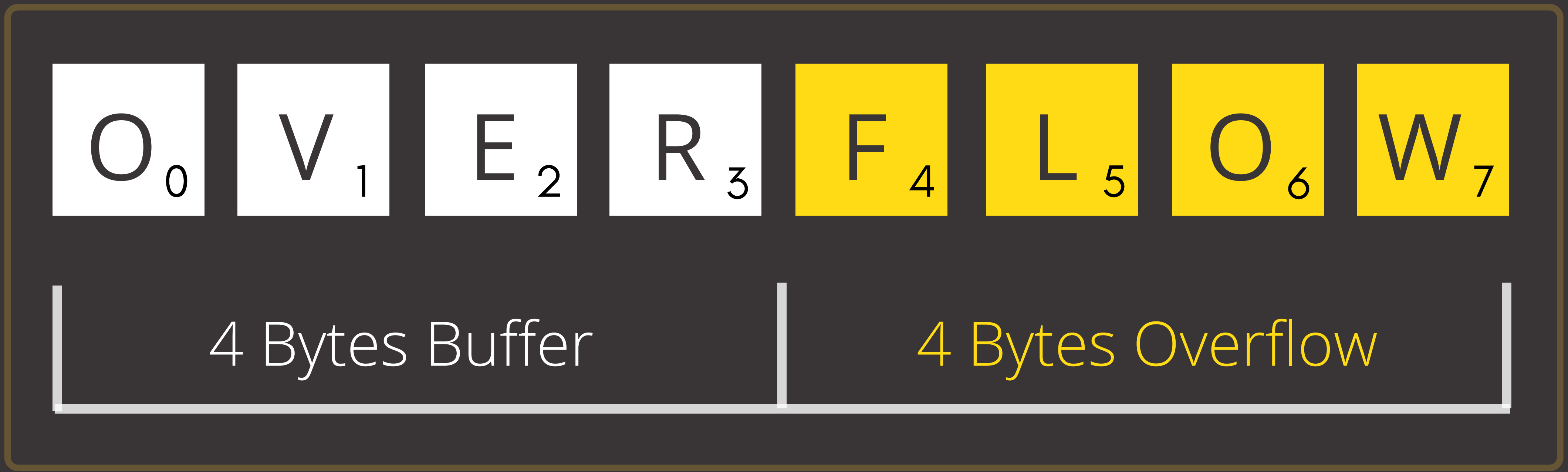
Mandeep Singh Grang
Katherine Kjeer

WHAT WE'LL DISCUSS

- What is Checked C?
- Implementation of Checked C in Clang
- Novel algorithm to widen bounds for null-terminated pointers
- Novel algorithm for comparison of expressions
- Conversion of legacy C code to Checked C
- Experimental evaluation
- Resources

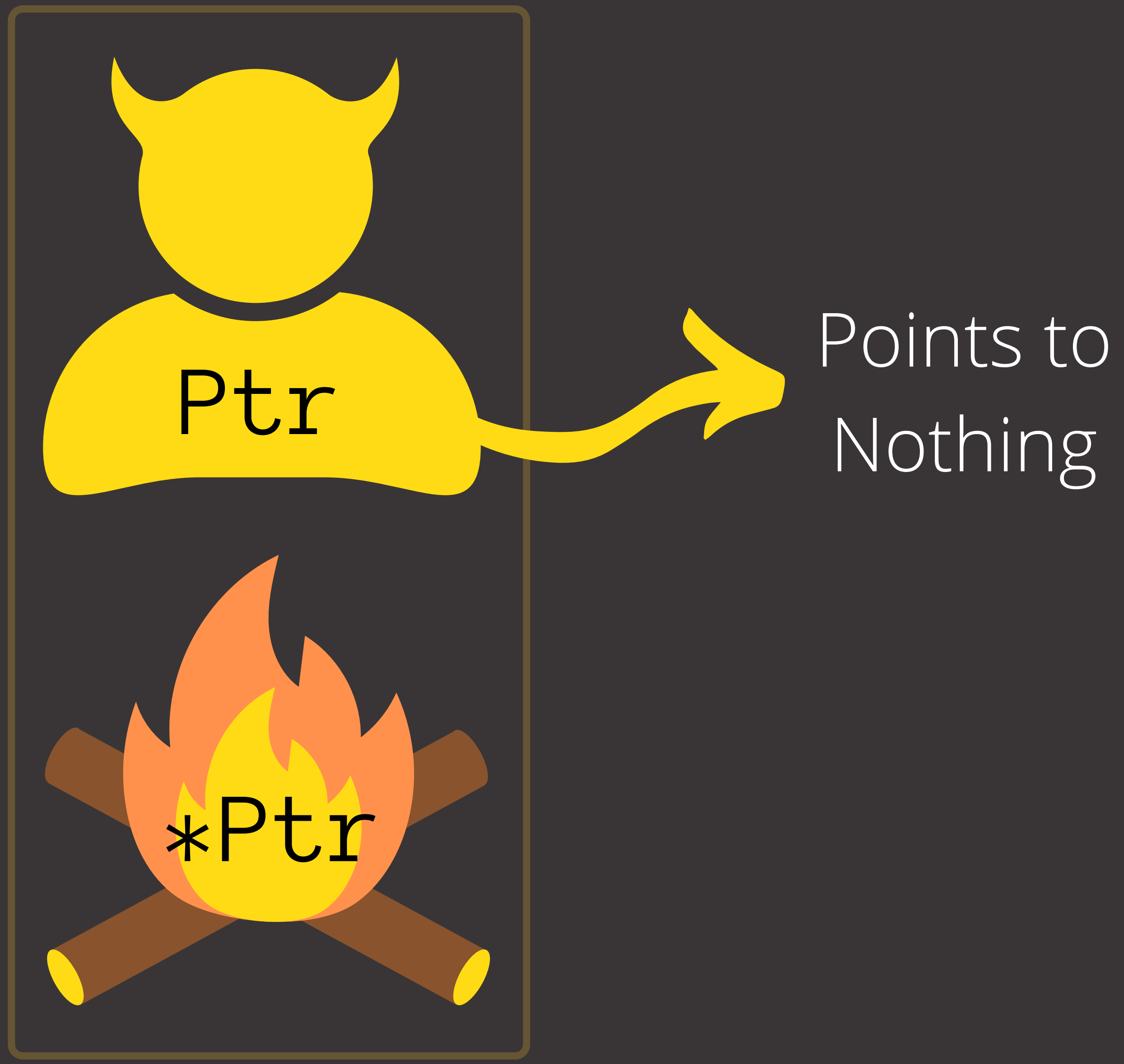
MEMORY SAFETY HAZARDS IN C

Buffer Overflow



O_0	V_1	E_2	R_3	F_4	L_5	O_6	W_7

Null Pointer Dereference



CHECKED C: IN A NUTSHELL

Extension to C

Supports spatial safety

New Pointer Types

Adds 3 new pointer types that are bounds-checked

Incremental Porting

Allows incremental porting from legacy C

Syntax like C++

Syntax for checked pointers is borrowed from C++ templates

Implemented in Clang

Checked C has been implemented in our fork of Clang

https://bit.ly/3kmepEp

_Ptr<T>

1

Points to a Single Object

Points to an object of type T

No Pointer Arithmetic

Pointer used for dereference only

⚠

Runtime Check for Non-nullness

Non-nullness checked at runtime, if necessary

_Ptr<T>

C	Checked C
`T *x;`	`_Ptr<T> x;`
`int *p;`	`_Ptr<int> p;`
`const int *p;`	`_Ptr<const int> p;`

`int x;`
`int *const p = &x;`
	`int x;`
	`const _Ptr<int> p = &x;`

_Array_ptr<T>

Pointer to Array

Points to an element of an array of type T

Pointer Arithmetic Allowed

Pointer arithmetic can be done on this pointer type

Runtime Check for Bounds

Non-nullness and bounds checked at runtime, if necessary

_Array_ptr<T>

C	Checked C

`T *x = "";`
`T x[] = {};`
	`_Array_ptr<T> x = "";`
	`T x _Checked[] = {};`

`const char *p = "abc";`	`_Array_ptr<const char> p = "abc";`
`char *foo(char p[]);`	`_Array_ptr<char> foo(char p _Checked[]);`

_Nt_array_ptr<T>

"abc\0"

Null Terminated Array

Points to a sequence of elements that ends with a null terminator

'\0'

Element Access

An element of the sequence can be read provided the preceding elements are not the null terminator

Automatic Bounds Widening

Bounds can be widened based on number of elements read

_Nt_array_ptr<T>

C	Checked C

T *x = "";
T x[] = {};
	_Nt_array_ptr<T> x = "";
	T x _Nt_checked[] = {};

const char *p = "abc";	_Nt_array_ptr<const char> p = "abc";
char *foo(char p[]);	_Nt_array_ptr<char> foo(char p _Nt_checked[]);

BOUNDS FOR ARRAY POINTERS

https://bit.ly/2F8W3YE

LIMIT MEMORY

Describe memory range pointer can access

LOW-LEVEL CONTROL

Programmer declares bounds that act as invariants

RUNTIME CHECKS

Check that memory accesses are within bounds

STATIC CHECKS

Check that bounds invariants are not violated

BOUNDS DECLARATIONS

COUNT

```
p : count(n)
```

p can access n array elements

RANGE

```
p : bounds(e1, e2)
```

p can access memory from e1 to e2

BYTE COUNT

```
p : byte_count(n)
```

p can access n bytes

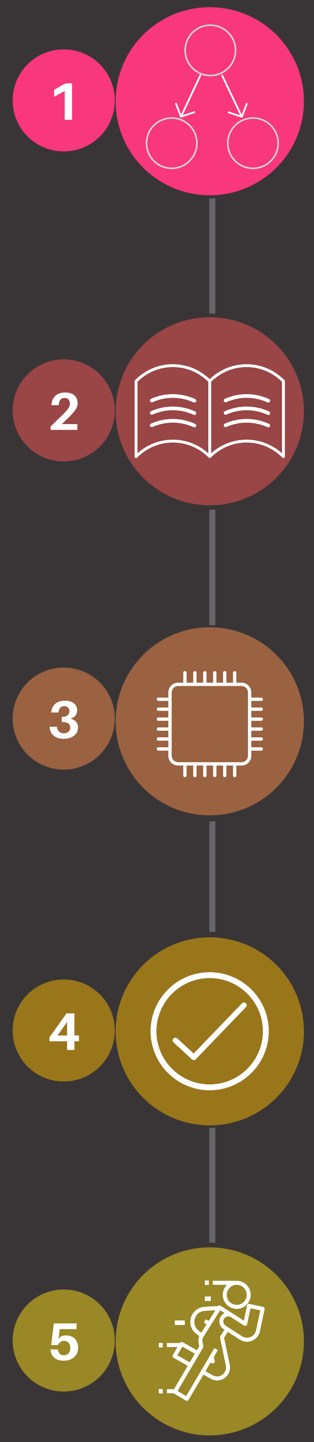
UNKNOWN

```
p : bounds(unknown)
```

p cannot be used to access memory

DYNAMICALLY CHECKING BOUNDS

```
void f(_Array_ptr<int> p : count(len),
       size_t len) {
  for (int i = 0; i <= len; ++i) {
    int n = p[i];
  }
}
```



Visit an expression that reads memory via a pointer

`p[i]`

DYNAMICALLY CHECKING BOUNDS

```
void f(_Array_ptr<int> p : count(len),
       size_t len) {
  for (int i = 0; i <= len; ++i) {
    int n = p[i];
  }
}
```

1. Visit an expression that reads memory via a pointer

   `p[i]`

2. Get the pointer-typed expression that reads memory

   `p[i] == *(p + i) => p + i (pointer p)`

3.

4.

5.

DYNAMICALLY CHECKING BOUNDS

```
void f(_Array_ptr<int> p : count(len),
       size_t len) {
  for (int i = 0; i <= len; ++i) {
    int n = p[i];
  }
}
```

1 Visit an expression that reads memory via a pointer

`p[i]`

2 Get the pointer-typed expression that reads memory

`p[i] == *(p + i) => p + i (pointer p)`

3 Get the bounds of the pointer-typed expression

`count(len) => bounds(p, p + len)`

4

5

DYNAMICALLY CHECKING BOUNDS

```
void f(_Array_ptr<int> p : count(len),
       size_t len) {
  for (int i = 0; i <= len; ++i) {
    int n = p[i];
  }
}
```

1. Visit an expression that reads memory via a pointer
   `p[i]`
2. Get the pointer-typed expression that reads memory
   `p[i] == *(p + i) => p + i (pointer p)`
3. Get the bounds of the pointer-typed expression
   `count(len) => bounds(p, p + len)`
4. Insert a dynamic check for the expression and bounds
   `p[i], bounds(p, p + len)`
5.

DYNAMICALLY CHECKING BOUNDS

```
void f(_Array_ptr<int> p : count(len),
          size_t len) {
   for (int i = 0; i <= len; ++i) {
      int n = p[i];
   }
}
```

1. Visit an expression that reads memory via a pointer
   `p[i]`
2. Get the pointer-typed expression that reads memory
   `p[i] == *(p + i) => p + i (pointer p)`
3. Get the bounds of the pointer-typed expression
   `count(len) => bounds(p, p + len)`
4. Insert a dynamic check for the expression and bounds
   `p[i], bounds(p, p + len)`
5. At runtime, check that the pointer is within bounds
   `0 <= (p + i) < (p + len)`
   `0 <= i < len`

DYNAMICALLY CHECKING BOUNDS

```
void f(_Array_ptr<int> p : count(len),
         size_t len) {
   for (int i = 0; i <= len; ++i) {
      int n = p[i];
   }
}
```

When i == len:

runtime error!

1. Visit an expression that reads memory via a pointer
   `p[i]`
2. Get the pointer-typed expression that reads memory
   `p[i] == *(p + i) => p + i (pointer p)`
3. Get the bounds of the pointer-typed expression
   `count(len) => bounds(p, p + len)`
4. Insert a dynamic check for the expression and bounds
   `p + i, bounds(p, p + len)`
5. At runtime, check that the pointer is within bounds
   `0 <= (p + i) < (p + len)`
   `0 <= i < len`

STATICALLY CHECKING BOUNDS DECLARATIONS

INFER

Bounds for pointer-typed expressions

CONVERT

Inferred and declared bounds to ranges

CHECK

Declared range is within inferred range

```
LHS = RHS;
```

ASSIGNMENTS:

Check RHS bounds contain LHS bounds

```
void f(param);
f(arg);
```

FUNCTION CALLS:

Check arg bounds contain param bounds

BOUNDS INFERENCE FOR EXPRESSIONS

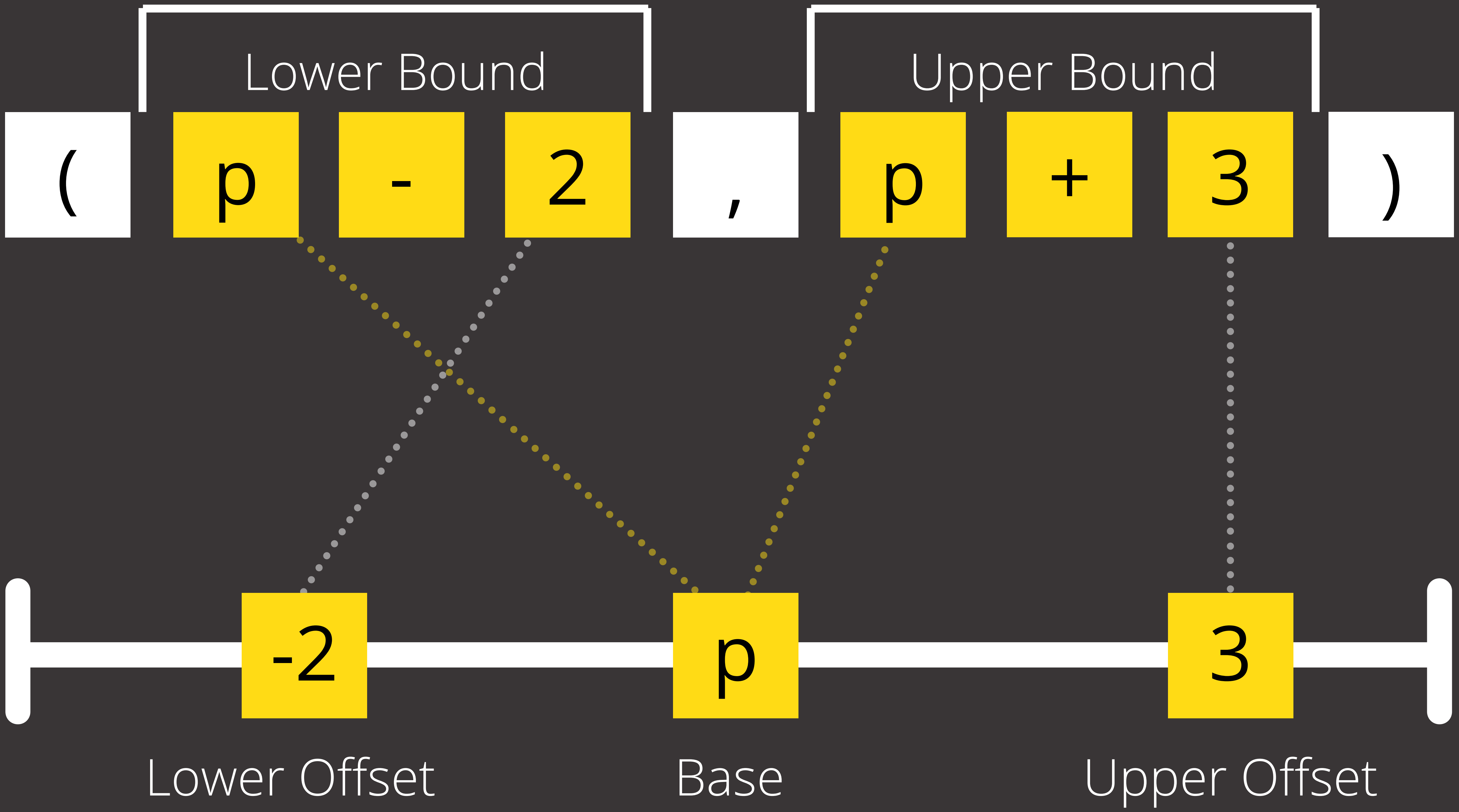
Declared `(p, p + x)`

	Inferred

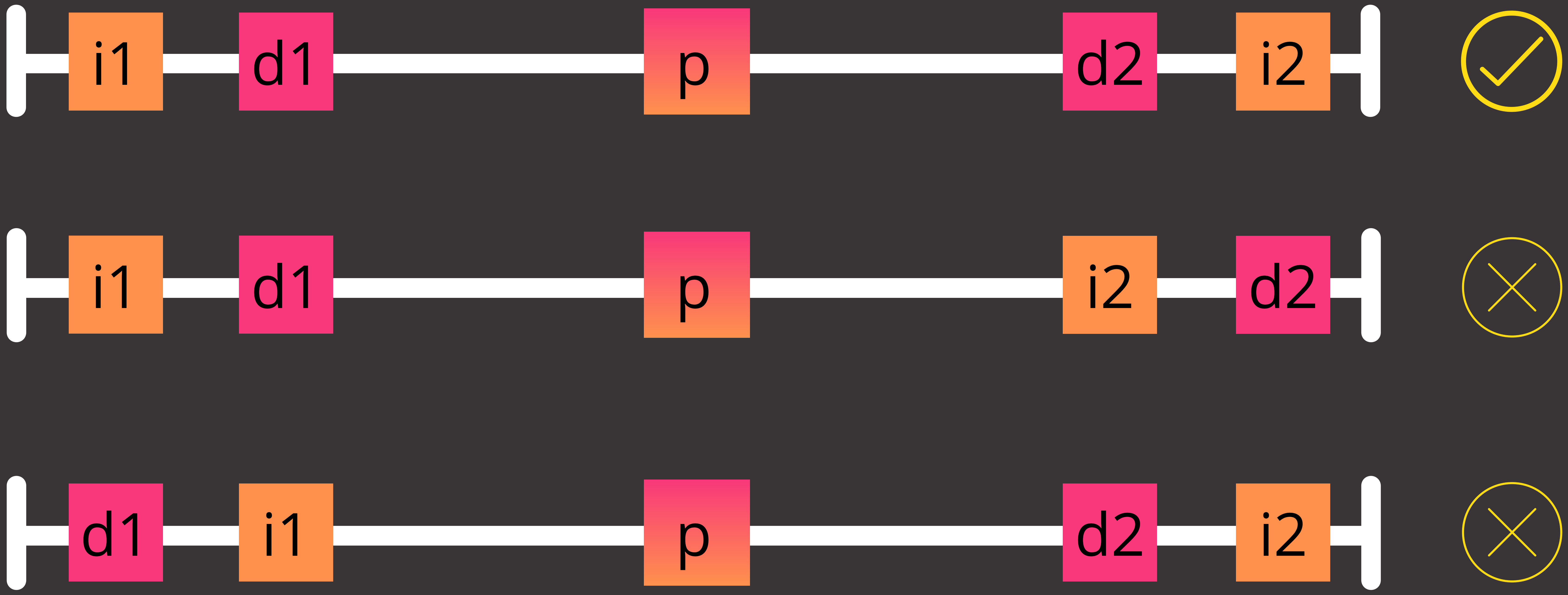
`void f(_Array_ptr<int> p : count(x), int x,`
`       _Array_ptr<int> q : count(3)) {`	

`  p = q;`	`(q, q + 3)`
`  p = (int _Checked[]){ 0, 1 };`	`{ 0, 1 }, { 0, 1 } + 2)`
`  p++; // Original value of p: p - 1.`	`(p - 1, p - 1 + x)`
`  x = x * 2; // No original value for x.`	unknown
`}`	

CONVERT BOUNDS TO RANGE

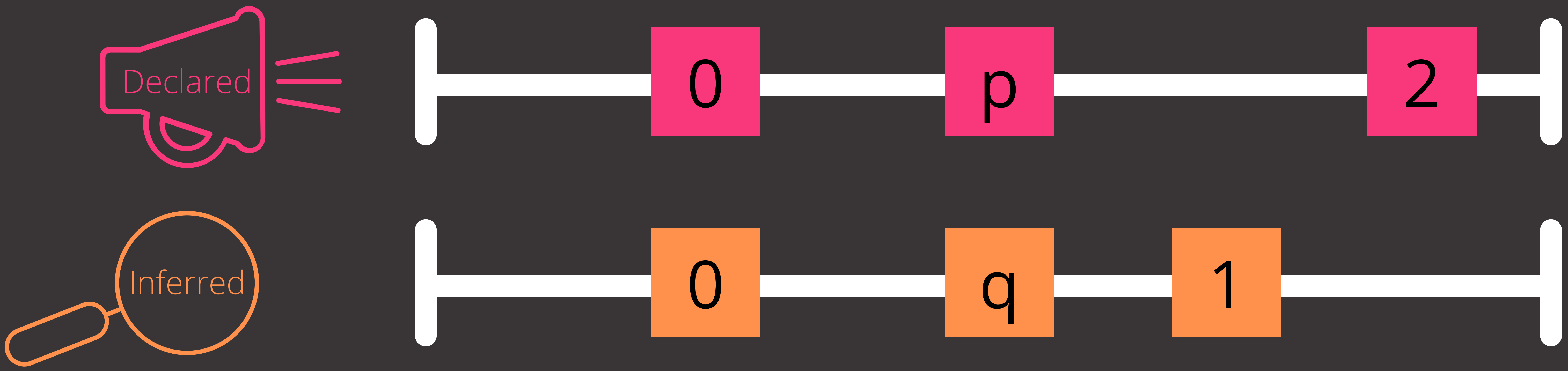


CHECK RANGES



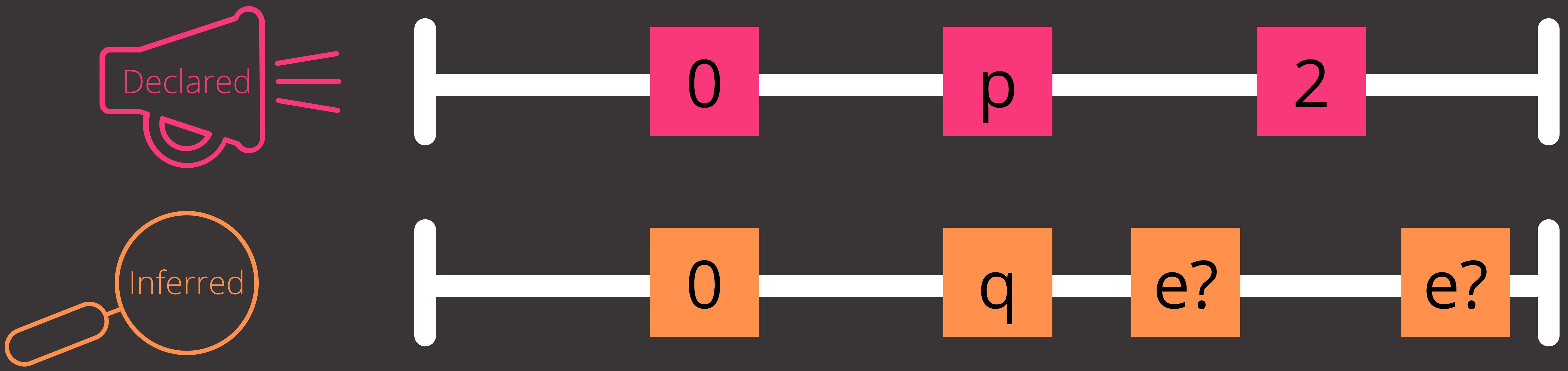
BOUNDS DECLARATIONS ERRORS

```
_Array_ptr<char> p : count(2) = 0;
_Array_ptr<char> q : count(1) = 0;
// Error: declared bounds for 'p' are invalid
// after assignment.
p = q;
```



BOUNDS DECLARATIONS WARNINGS

```
_Array_ptr<char> p : count(2) = 0;
_Array_ptr<char> q : count(e) = 0;
// Warning: cannot prove declared bounds for 'p'
// are valid after assignment.
p = q;
```



BOUNDS WIDENING

FOR NULL TERMINATED POINTERS

Lower bound Upper bound

```
_Nt_array_ptr<T> p : bounds(p, p) = "";
```

```
if (*p)
```

Ptr deref is at upper bound. Widen the bounds. New bounds: (p, p + 1)

```
  if (*(p + 1))
```

Ptr deref is at upper bound. Widen the bounds. New bounds: (p, p + 2)

```
    if (*(p + 3))
```

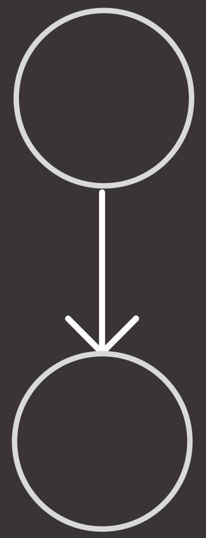
Ptr deref is NOT at upper bound. No bounds widening. Flag ERROR!

```
error: out-of-bounds memory access: if (*(p + 3))
note: accesses memory at or above the upper bound
note: inferred bounds are 'bounds(p, p + 2)'
```

BOUNDS WIDENING

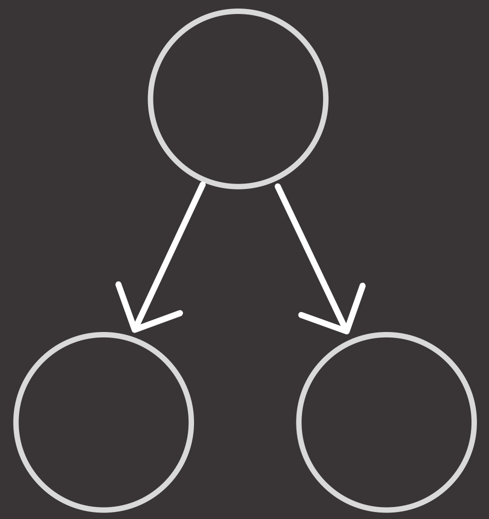
DATAFLOW PROPERTIES

https://bit.ly/35lm4yx



Forward

A basic block is visited before its successors



Path-Sensitive

Dataflow analysis generates different facts on the then and *else* branches

Flow-Sensitive

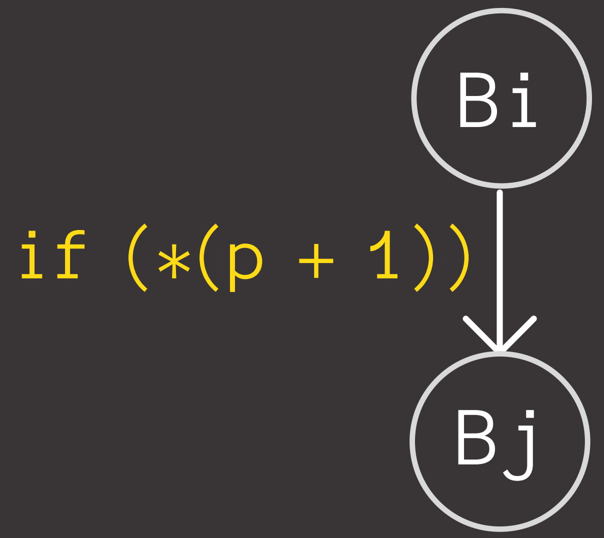
Dataflow analysis depends on the order of statements in a basic block

Intra-Procedural

Dataflow analysis is done on one function at a time

BOUNDS WIDENING

DATAFLOW EQUATIONS

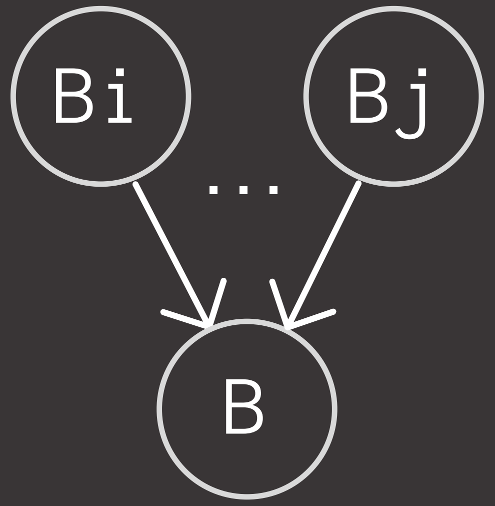


Gen[Bi->Bj]

Gen[Bi->Bj] ∪ {p:1},
where p ∈ _Nt_array_ptr

Kill[B]

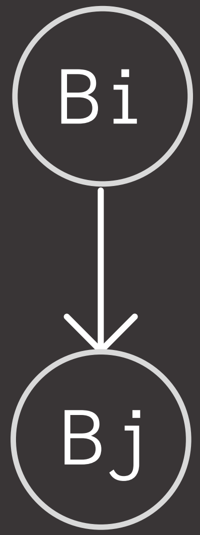
Kill[B] ∪ {p},
where p ∈ _Nt_array_ptr
and i ∈ decl_bounds(p)



In[B]

∩ Out[Bi->B],
where Bi ∈ pred(B)

Init: In[Entry] = ∅
In[B] = Top



Out[Bi->Bj]

(In[Bi] - Kill[Bi]) ∪
Gen[Bi->Bj]

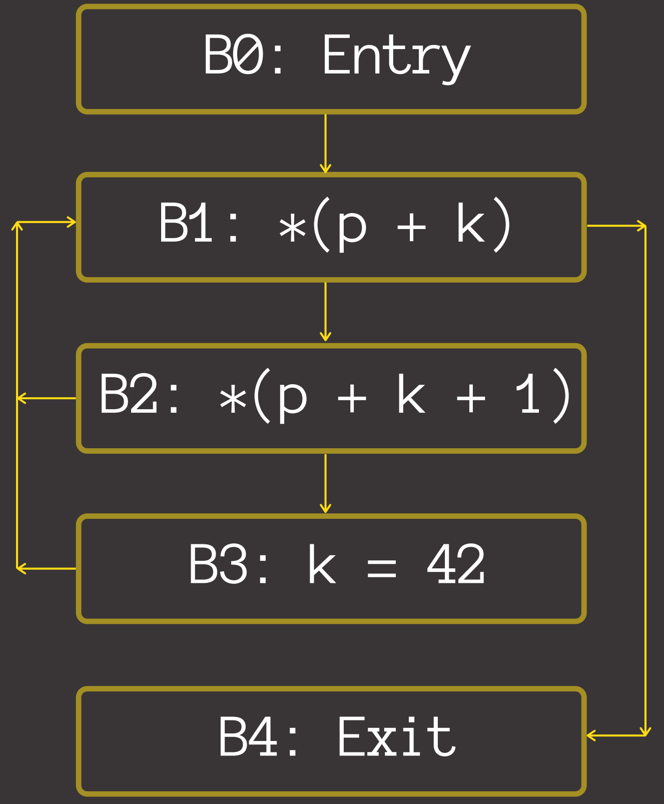
Init: Out[Entry->Bj] = ∅
Out[Bi->Bj] = Top

BOUNDS WIDENING

DATAFLOW ANALYSIS

```
int k = 0;
_Nt_array_ptr<T> p : bounds(p, p + k);

while (*(p + k))
  if (*(p + k + 1))
    k = 42;
```



Blocks	Pred	Succ	Gen	Kill	In (Init)	Out (Init)	In	Out	In	Out
B0	∅	{B1}	{B0->B1: ∅}	∅	∅	{B0->B1: ∅}	∅	{B0->B1: ∅}	∅	{B0->B1: ∅}
B1	{B0, B2, B3}	{B2, B4}	{B1->B2: {p:0}, B1->B4: ∅}	∅	{p:1}	{B1->B2: {p:1}, B1->B4: {p:1}}	∅	{B1->B2: {p:0}, B1->B4: ∅}	∅	{B1->B2: {p:0}, B1->B4: ∅}
B2	{B1}	{B3, B1}	{B2->B3: {p:1}, B2->B1: ∅}	∅	{p:1}	{B2->B3: {p:1}, B2->B1: {p:1}}	{p:1}	{B2->B3: {p:1} B2->B1: {p:1}}	{p:0}	{B2->B3: {p:1}, B2->B1: ∅}
B3	{B2}	{B1}	{B3->B1: ∅}	{p}	{p:1}	{B3->B1: {p:1}}	{p:1}	{B3->B1: ∅}	{p:1}	{B3->B1: ∅}
B4	{B1}	∅	∅	∅	{p:1}	∅	{p:1}	∅	∅	∅

BOUNDS WIDENING

THE NEED TO COMPARE EXPRESSIONS

```
_Nt_array_ptr<T> p : bounds(p, p + i + j + 4);
```

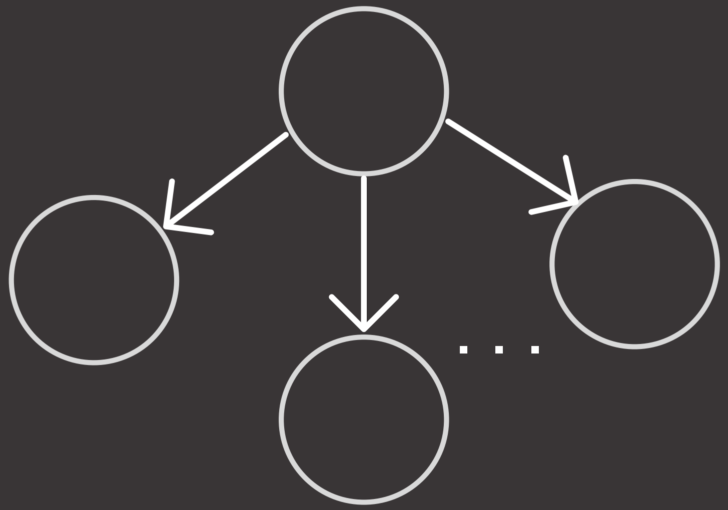
Should Widen Bounds	Should Not Widen Bounds
`if (*(p + i + j + 1 + 3))`	`if (*(p + i + j + 3))`
`if (*(2 + i + p + j + 2 + 0)`	`if (*(p + (i * j) + 4))`
`if (*(p + 5 + i − 1 + j))`	`if (*(p + i + 4))`
`if (*(j + p + i + (2 * 2)))`	`if (*(p + i + j + 4 + k))`

"We need a mechanism to determine if two expressions are equivalent"

SEMANTIC COMPARISON OF EXPRESSIONS

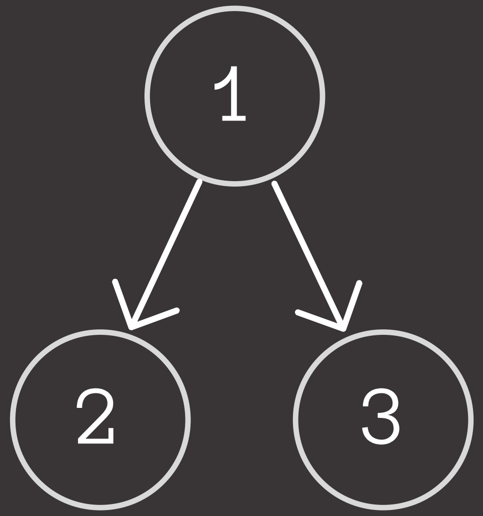
THE PREORDER AST

https://bit.ly/3bLT4kx



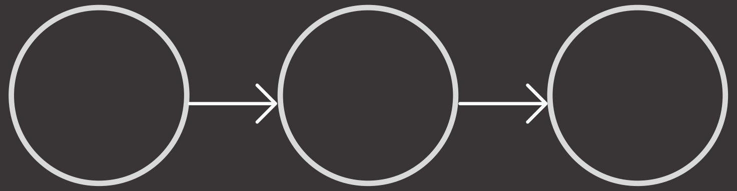
N-ary

The preorder AST is an n-ary tree



Preorder

It represents an expression in the preoder form



Flattened

The tree is flattened at each level by coalescing nodes with their parents

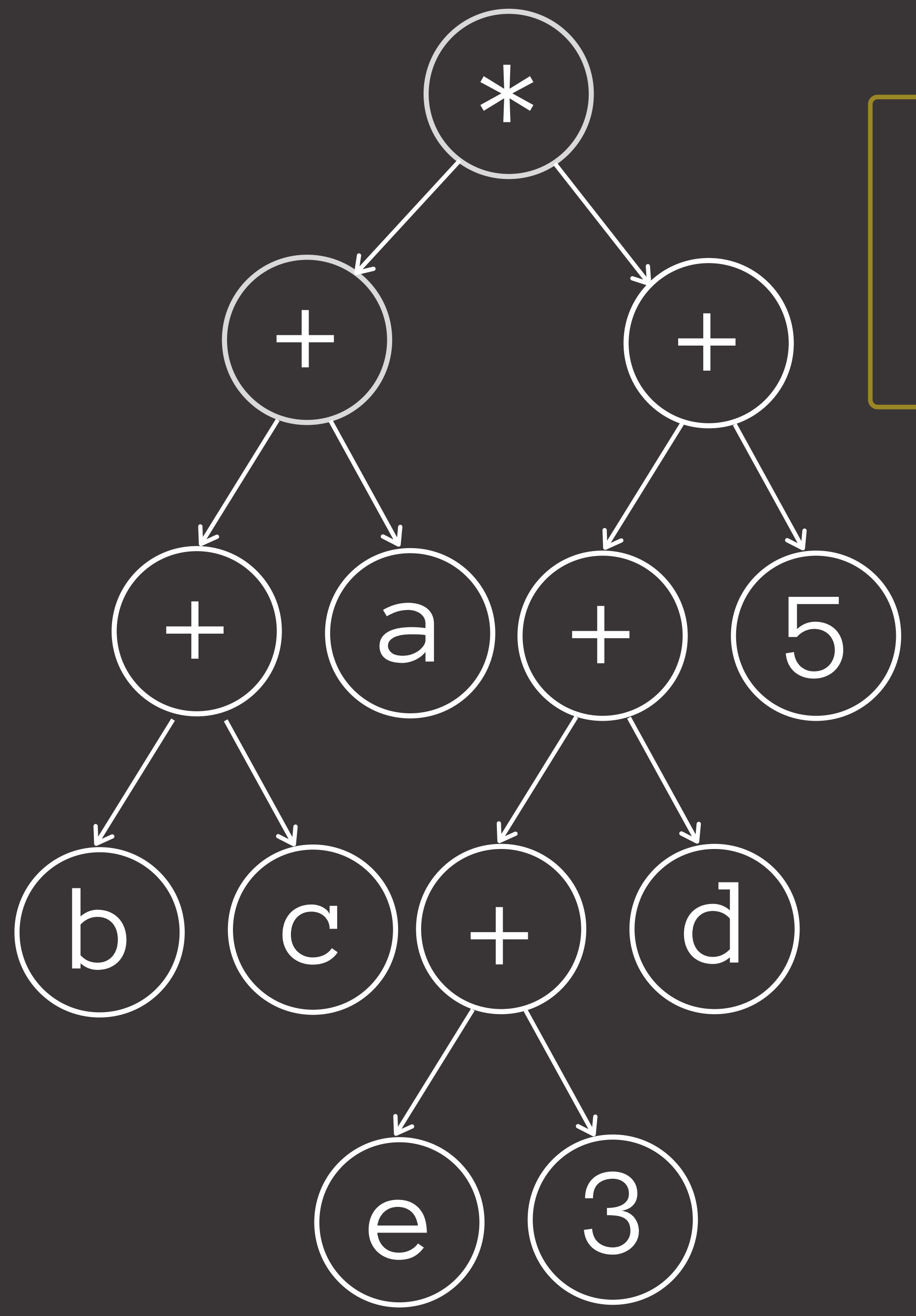
$E1 \equiv E2$

Normalized

The underlying expression is normalized by constant-folding and sorting the nodes of the tree

SEMANTIC COMPARISON OF EXPRESSIONS

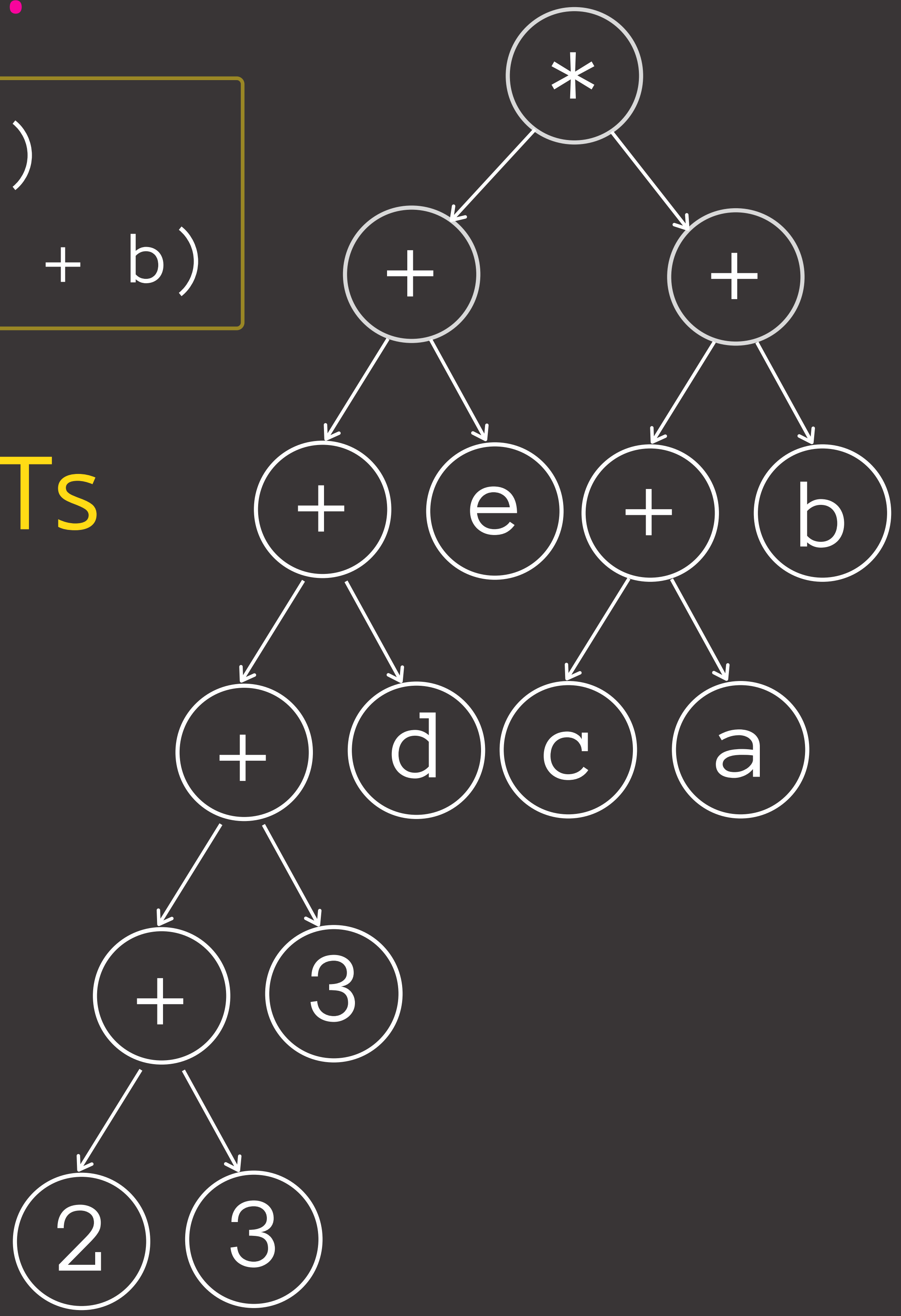
USING THE PREORDER AST



Are E1 and E2 equivalent?

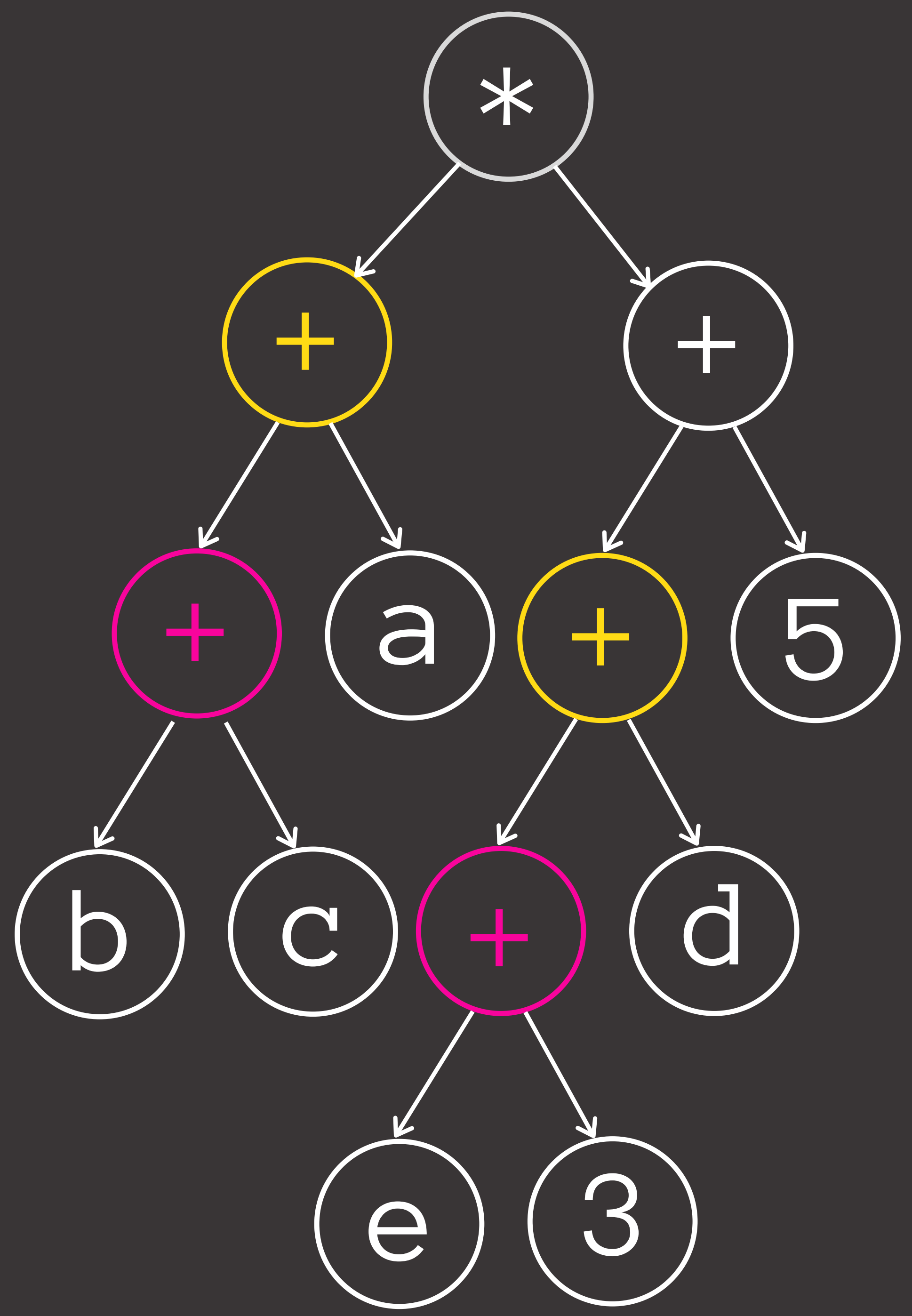
```
E1 = (b + c + a) * (e + 3 + d + 5)
E2 = (2 + 3 + 3 + d + e) * (c + a + b)
```

Step 1: Create preorder ASTs for E1 and E2

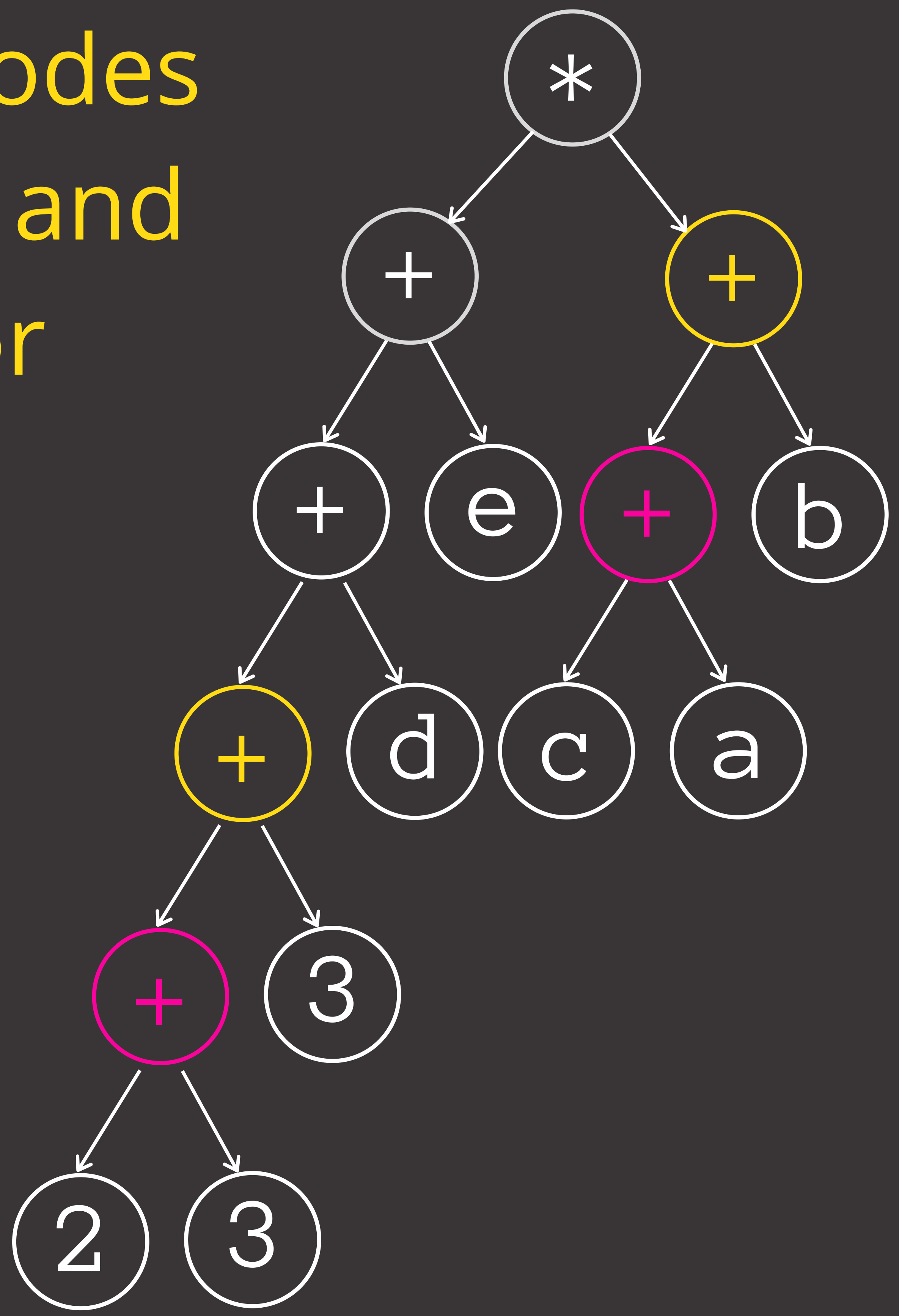


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

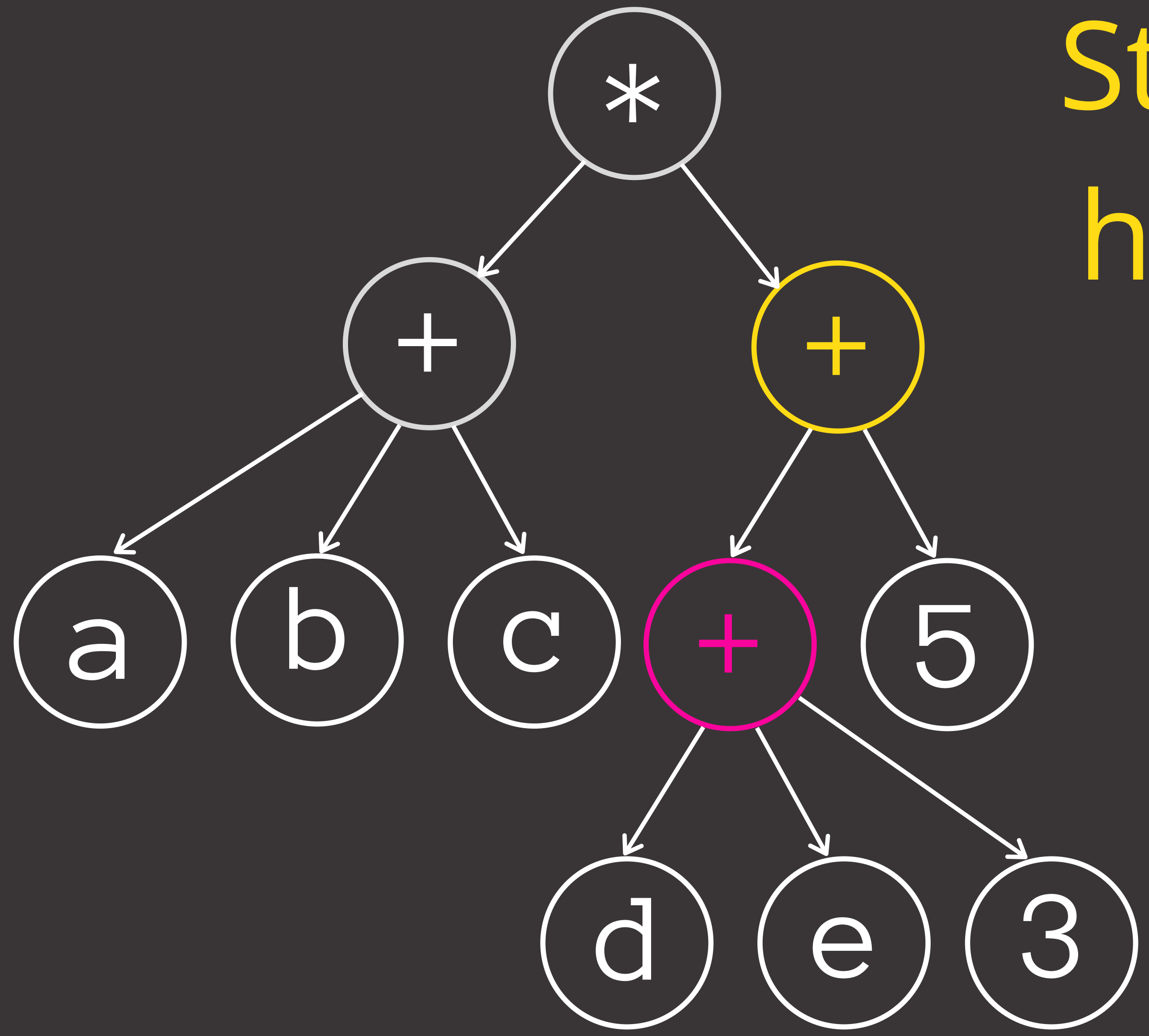


Step 2: Coalesce leaf nodes having a commutative and associative operator

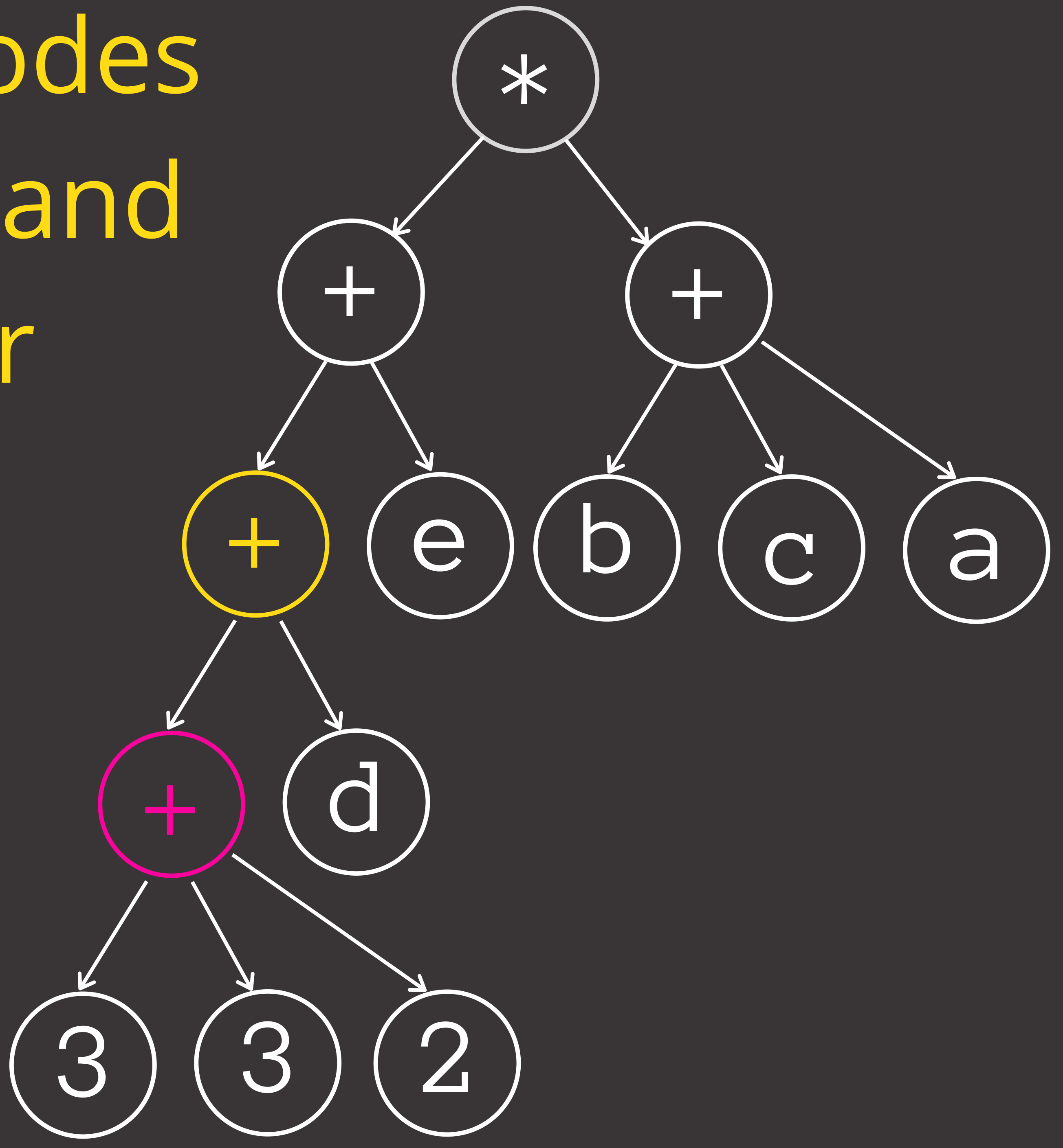


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

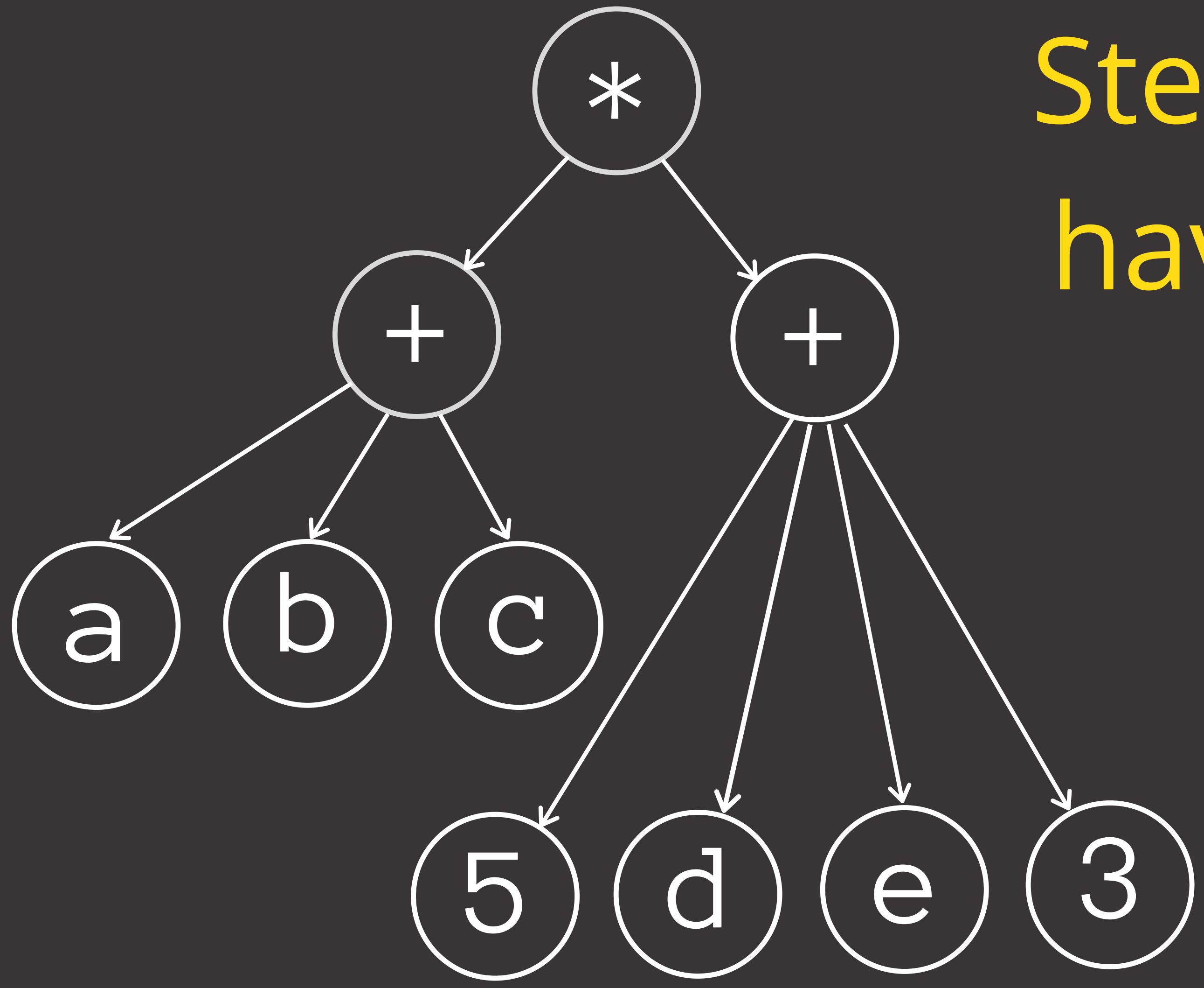


Step 2: Coalesce leaf nodes having a commutative and associative operator

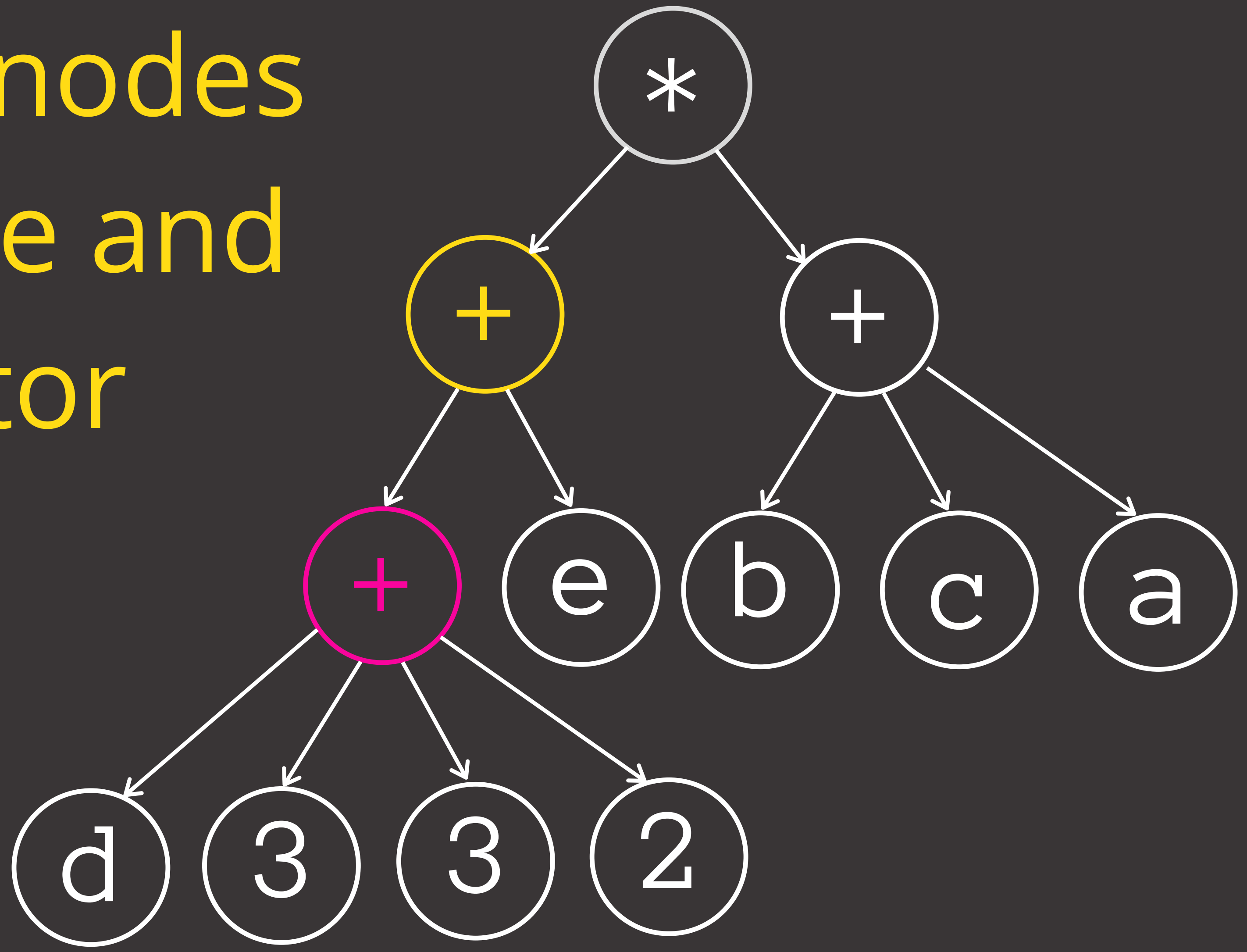


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

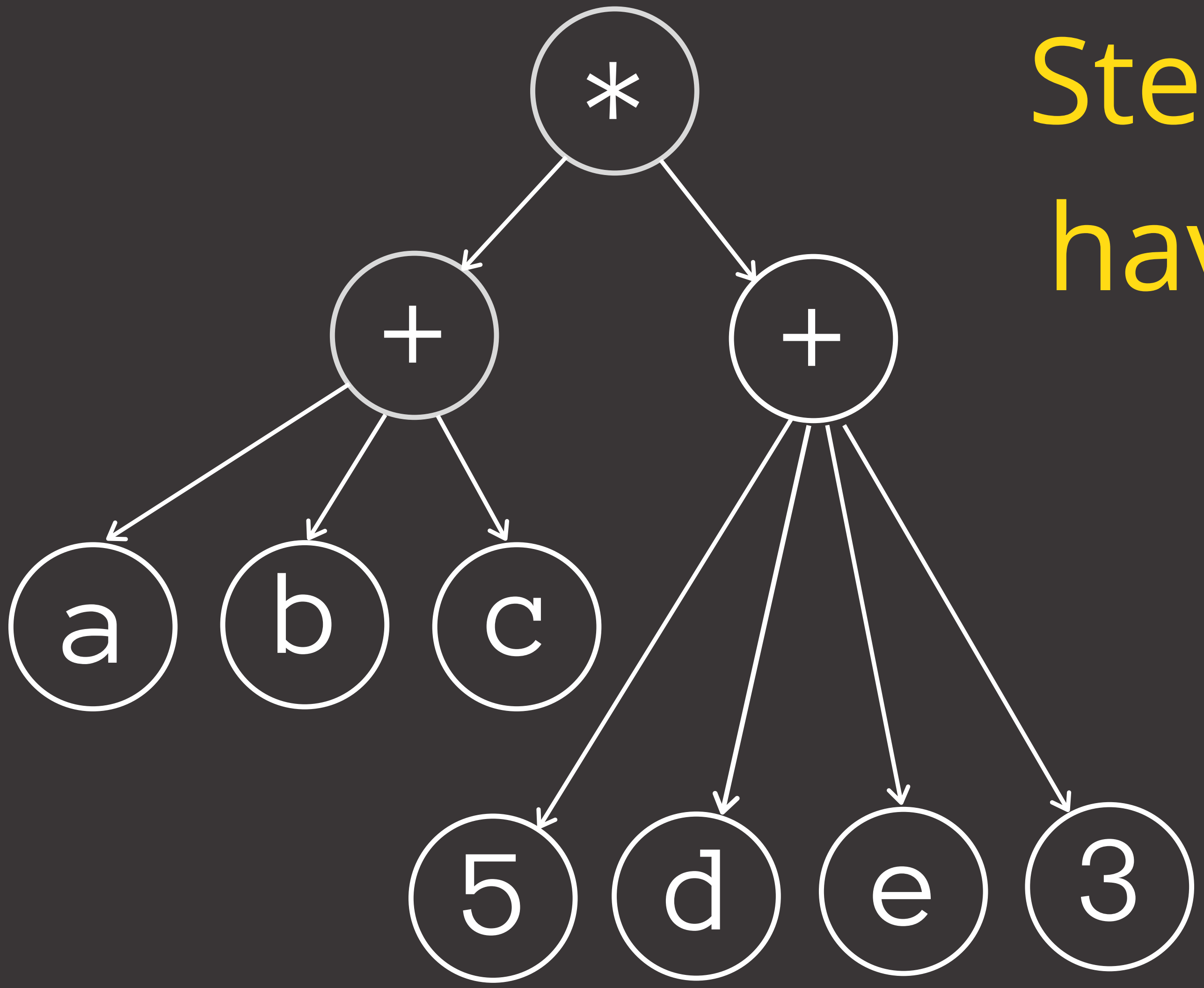


Step 2: Coalesce leaf nodes having a commutative and associative operator

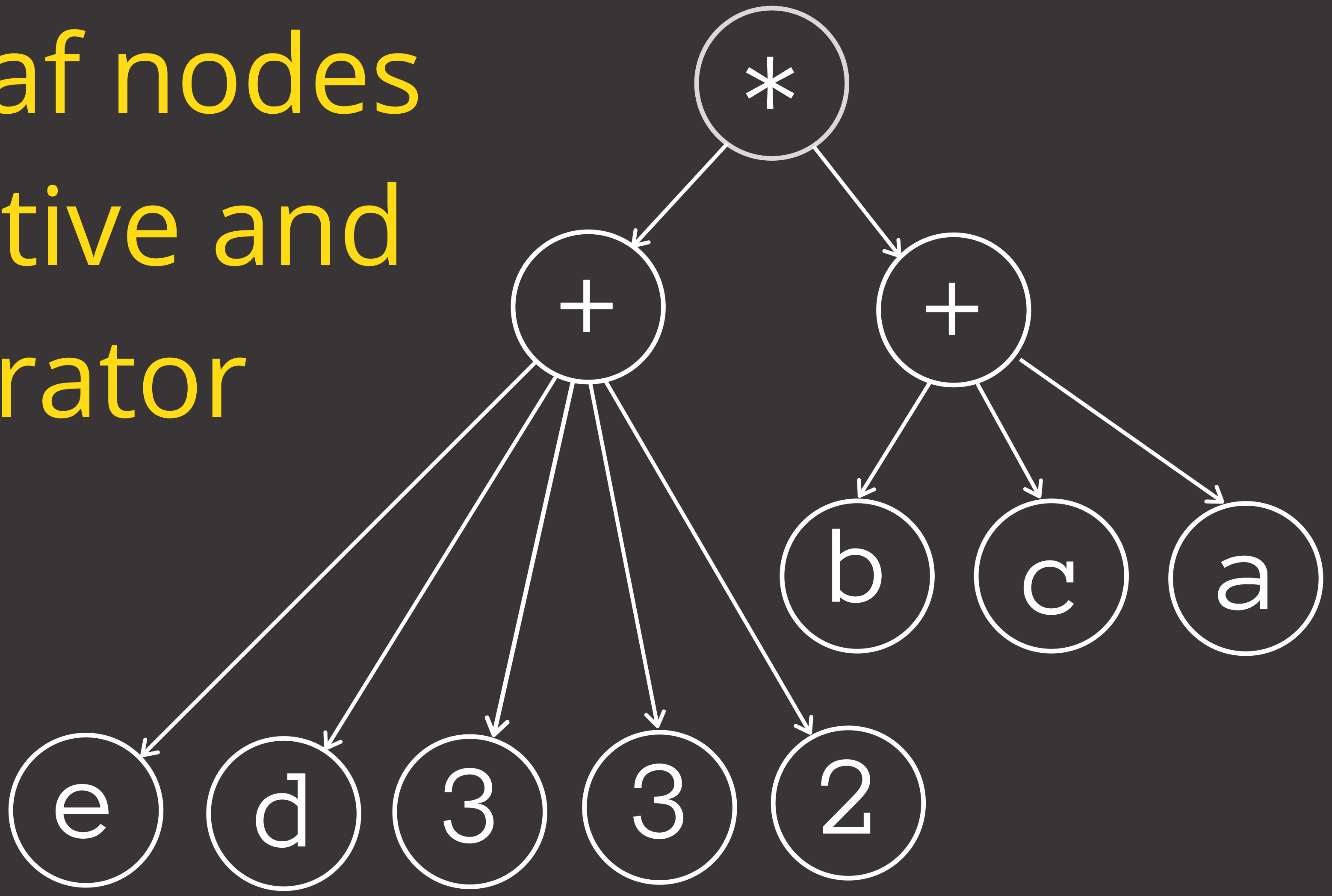


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

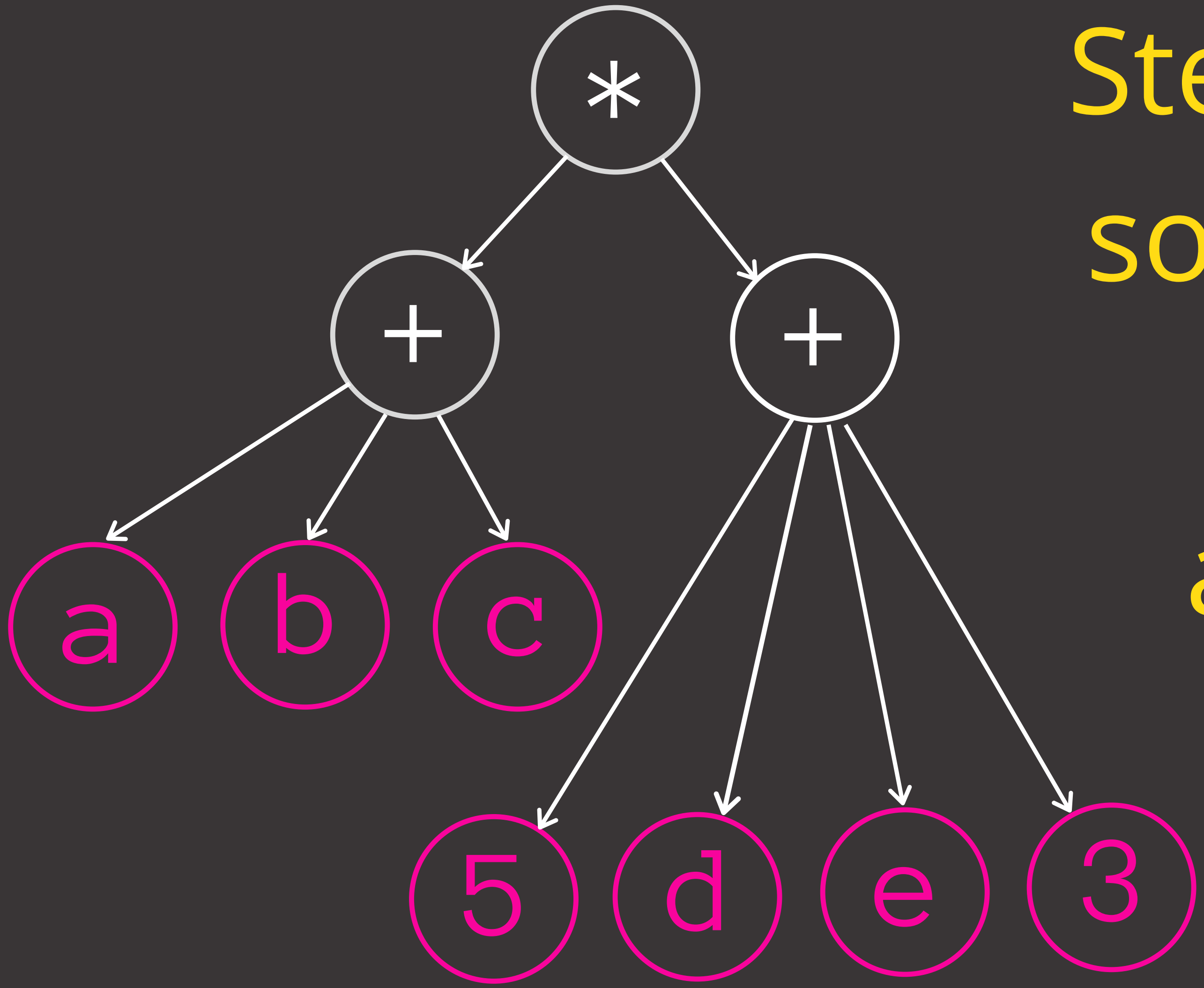


Step 2: Coalesce leaf nodes having a commutative and associative operator

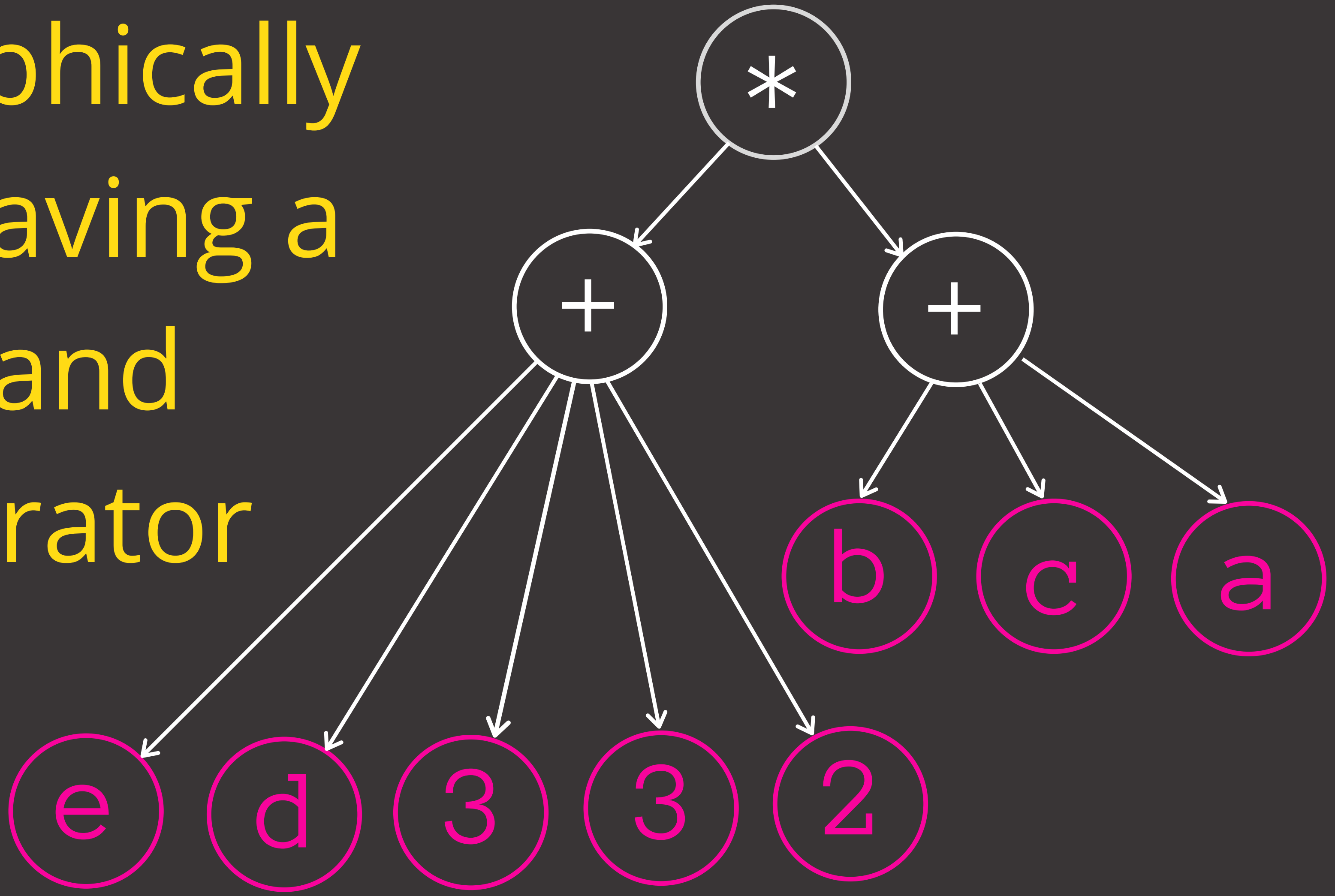


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

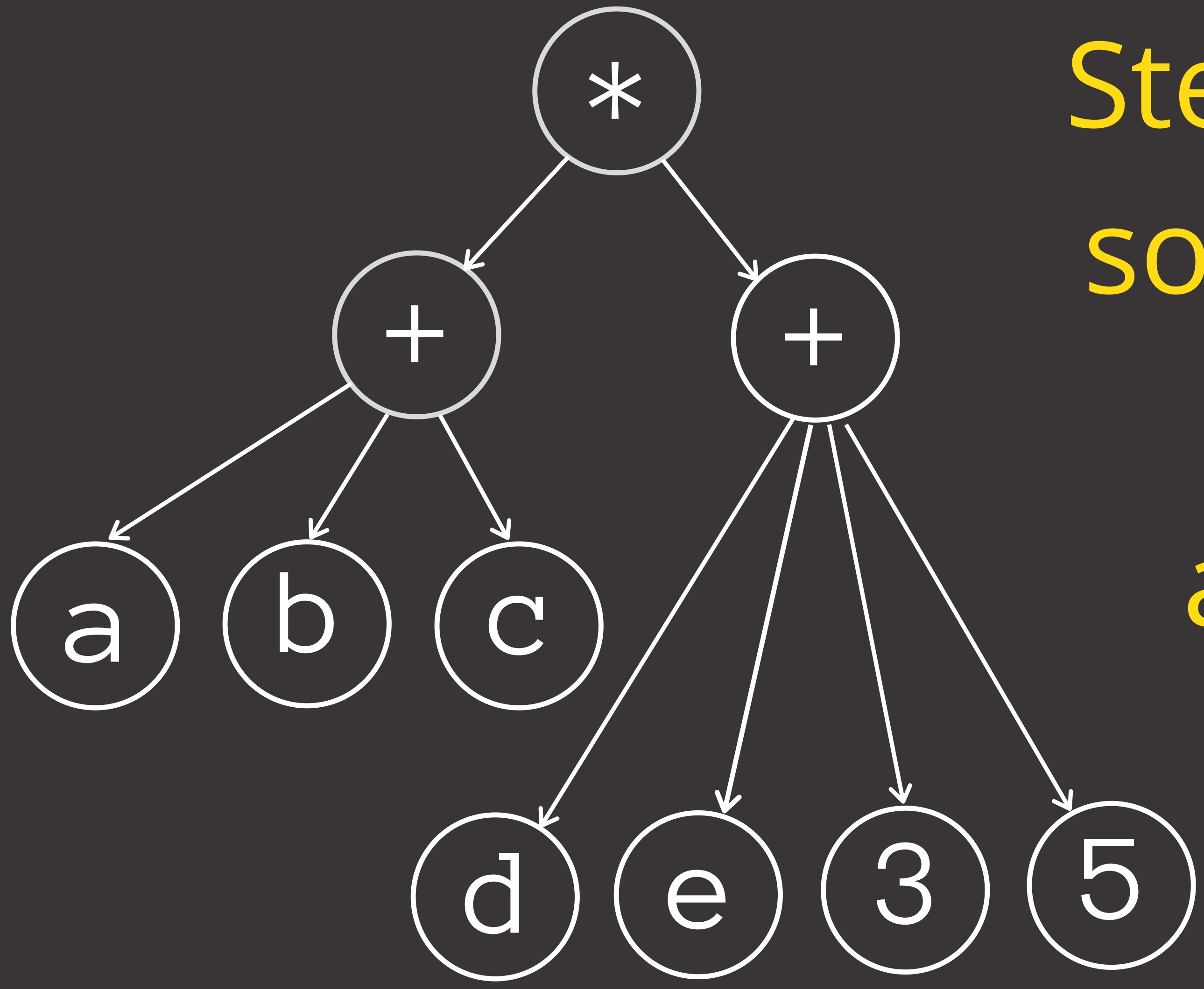


Step 3: Lexicographically sort leaf nodes having a commutative and associative operator

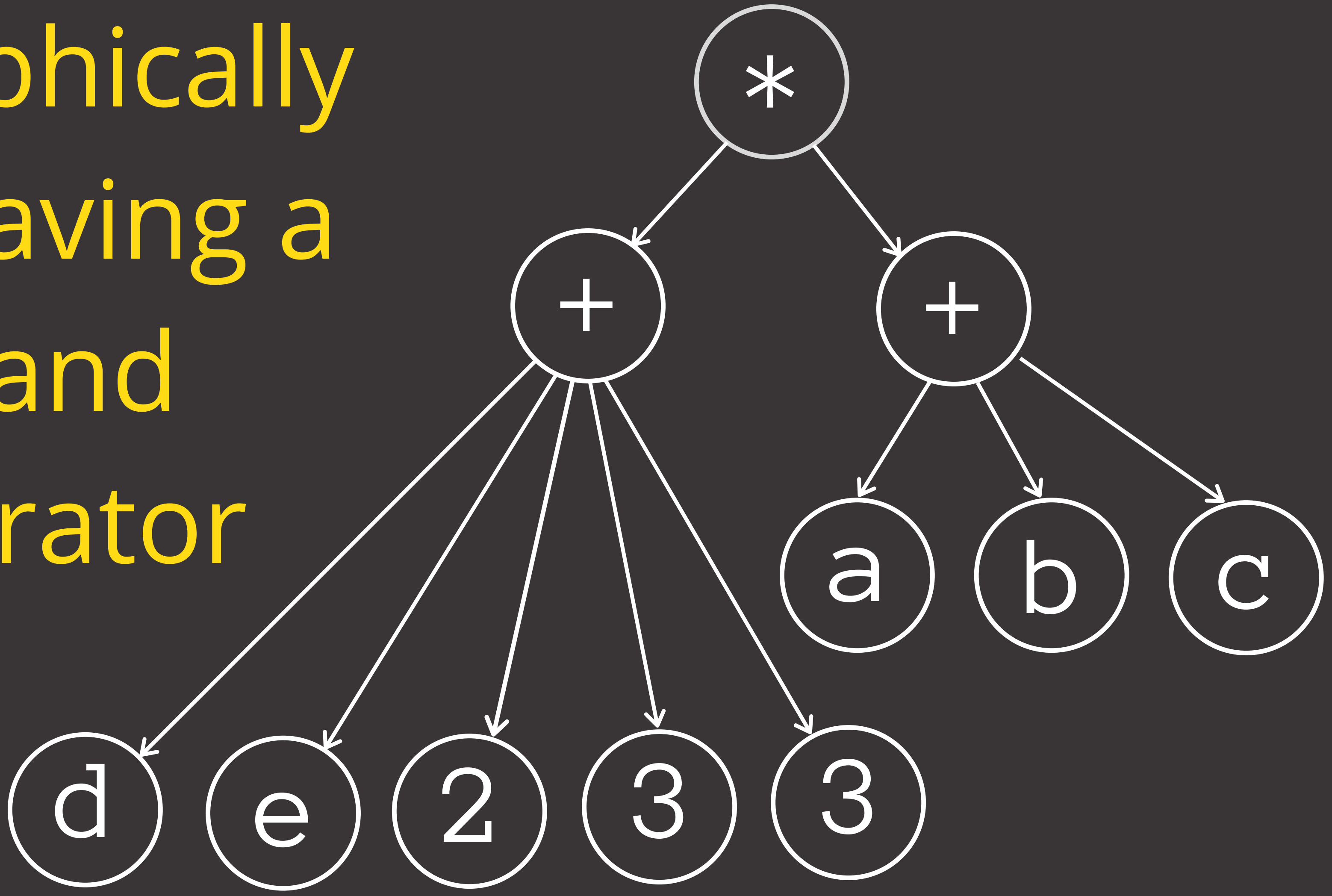


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

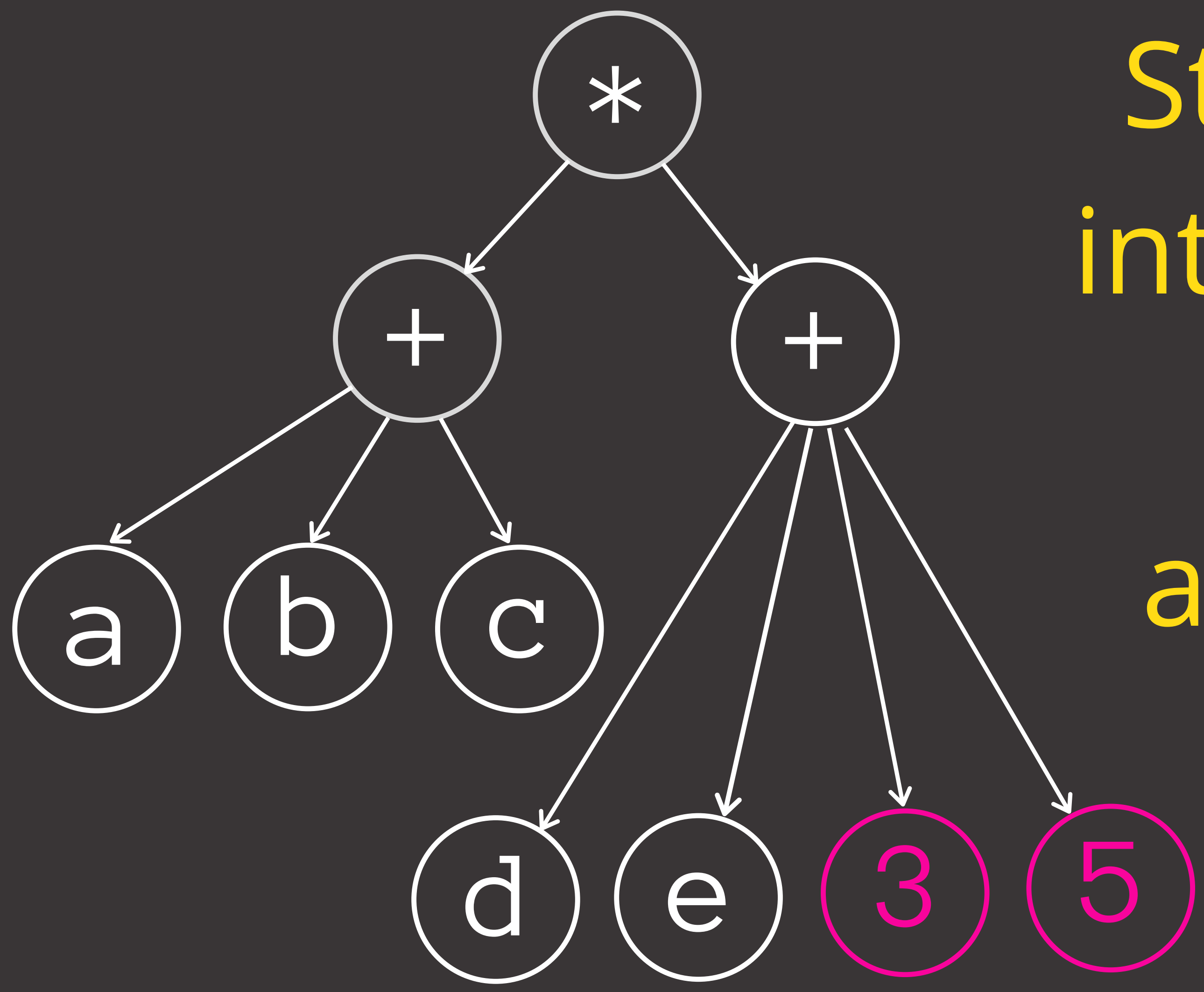


Step 3: Lexicographically sort leaf nodes having a commutative and associative operator

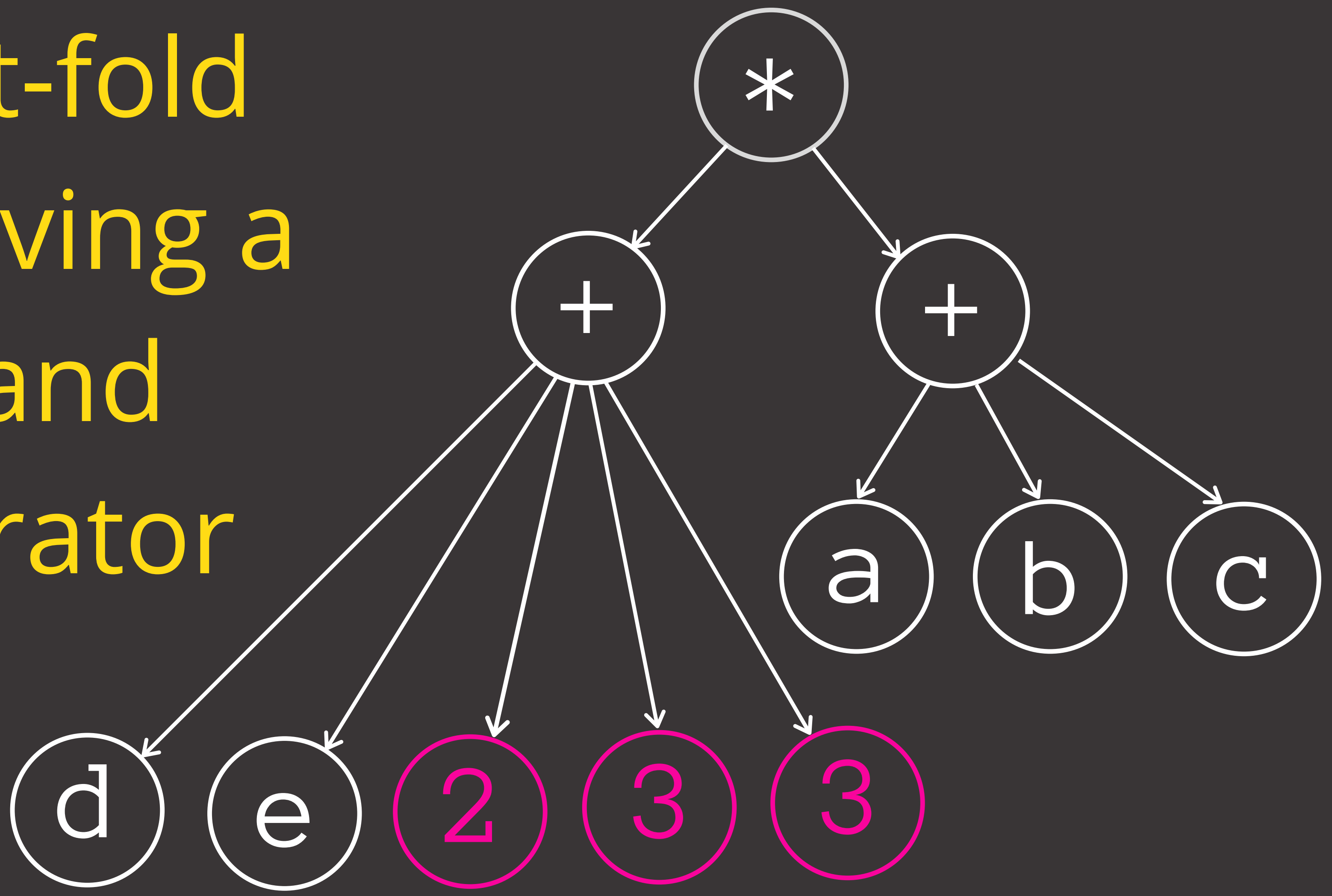


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

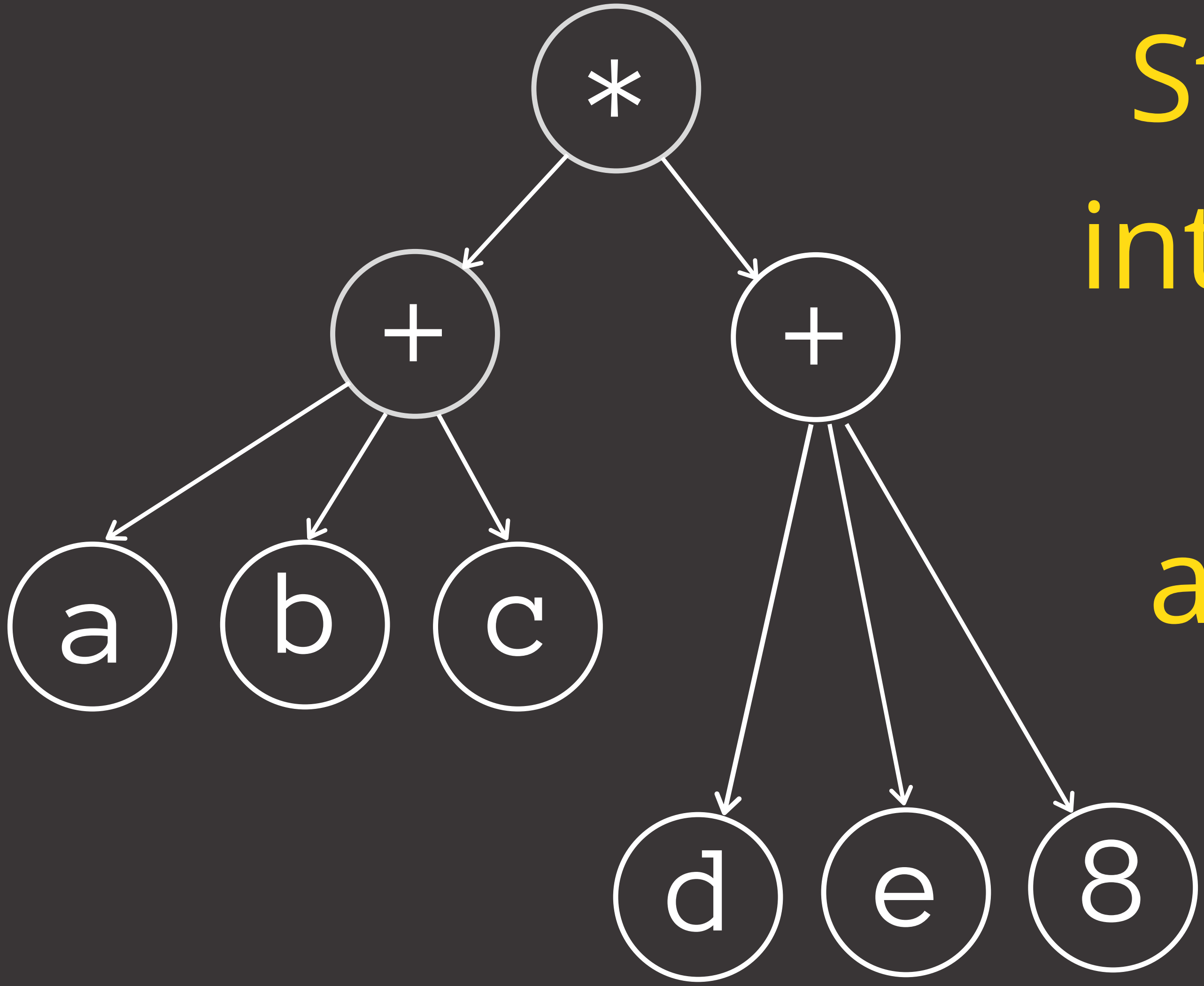


Step 4: Constant-fold integer nodes having a commutative and associative operator

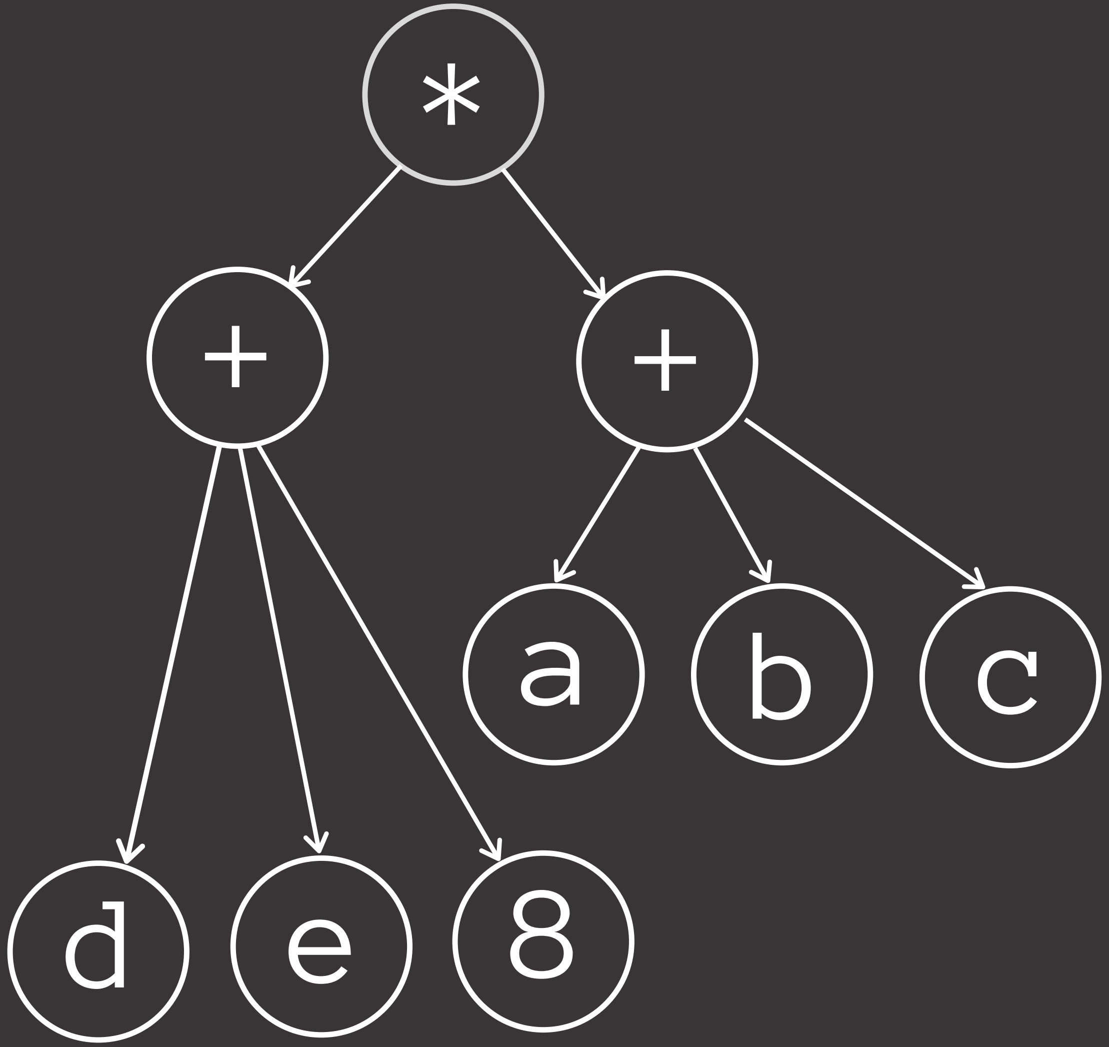


SEMANTIC COMPARISON OF EXPRESSIONS

USING THE PREORDER AST

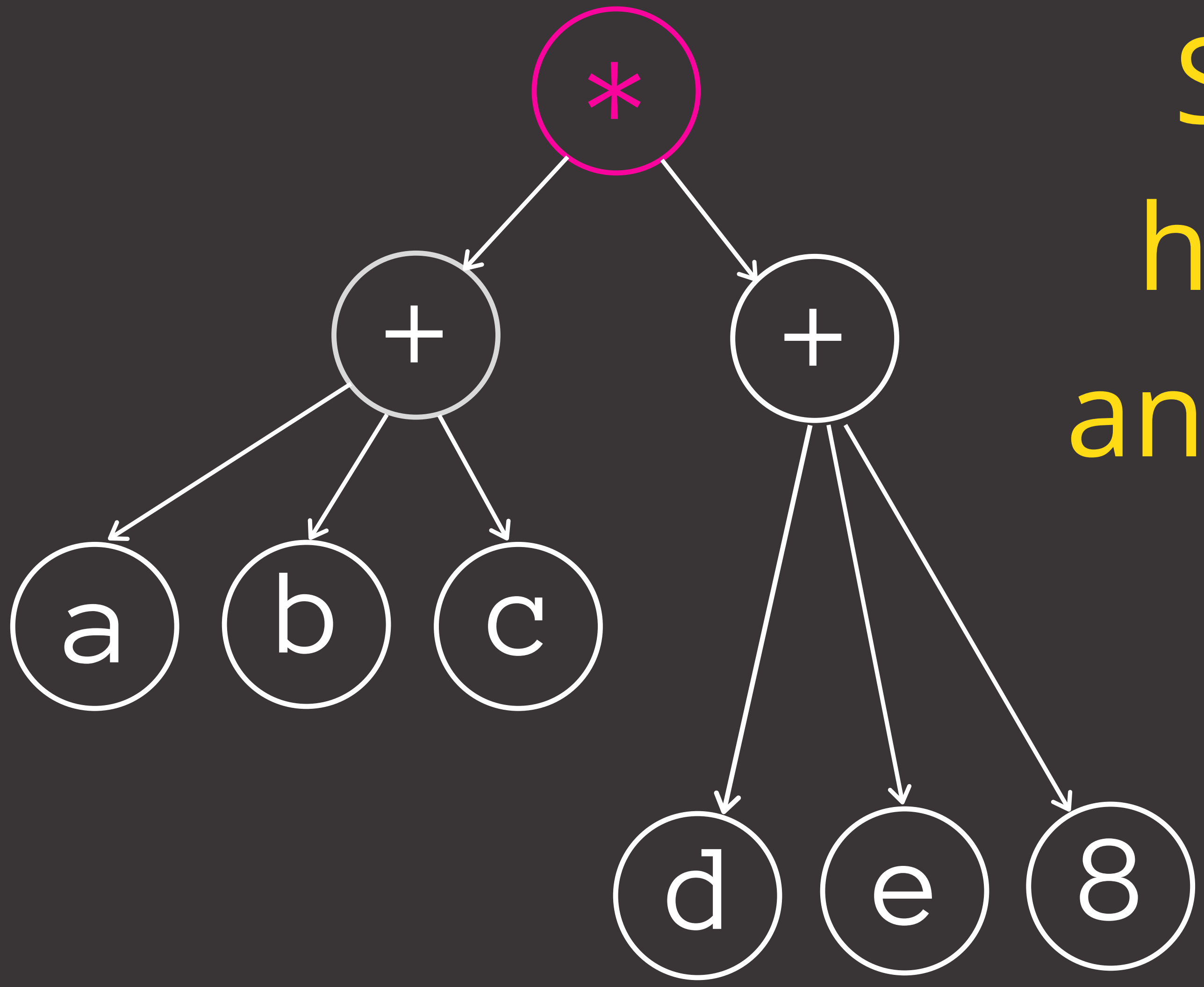


Step 4: Constant-fold integer nodes having a commutative and associative operator

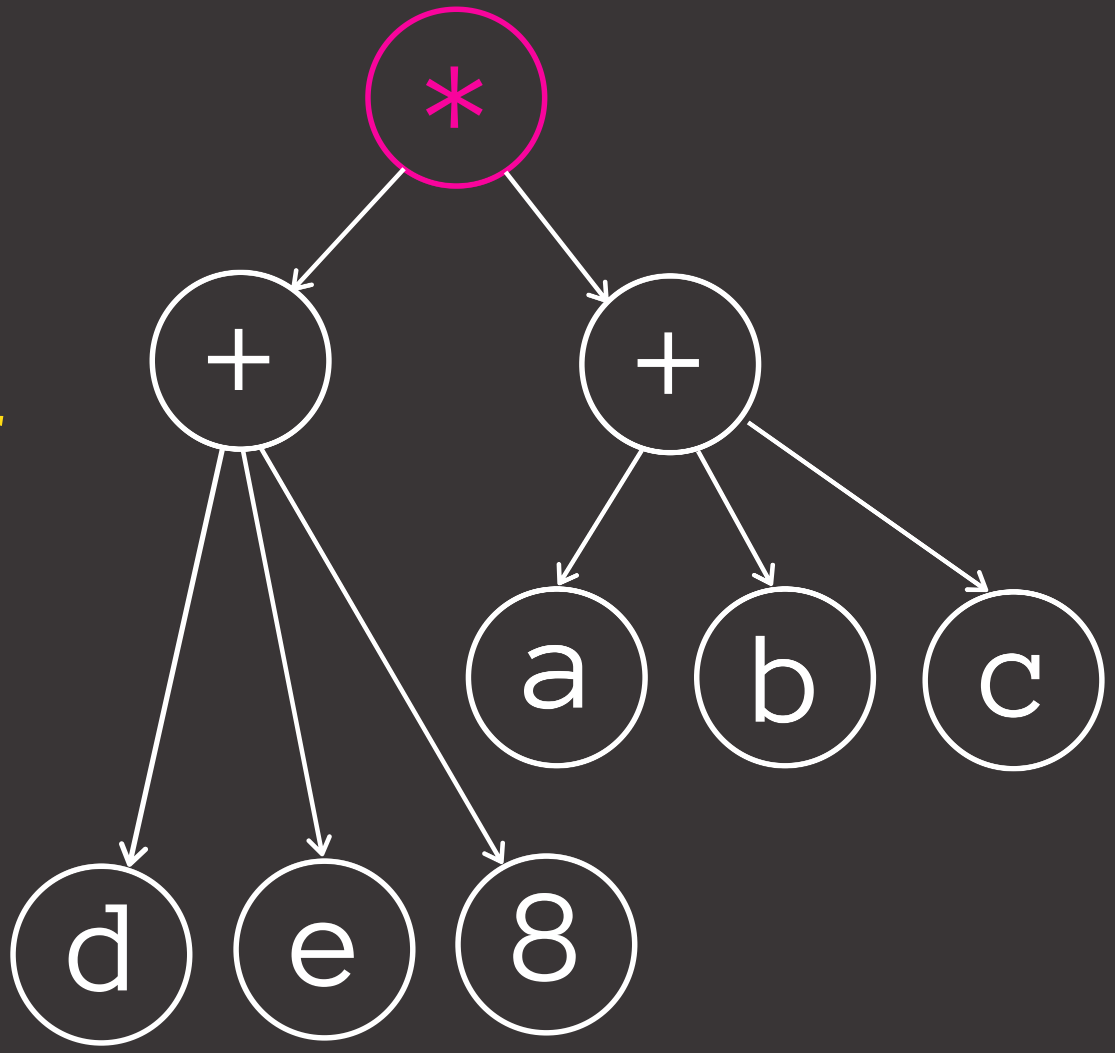


SEMANTIC COMPARISON OF EXPRESSIONS

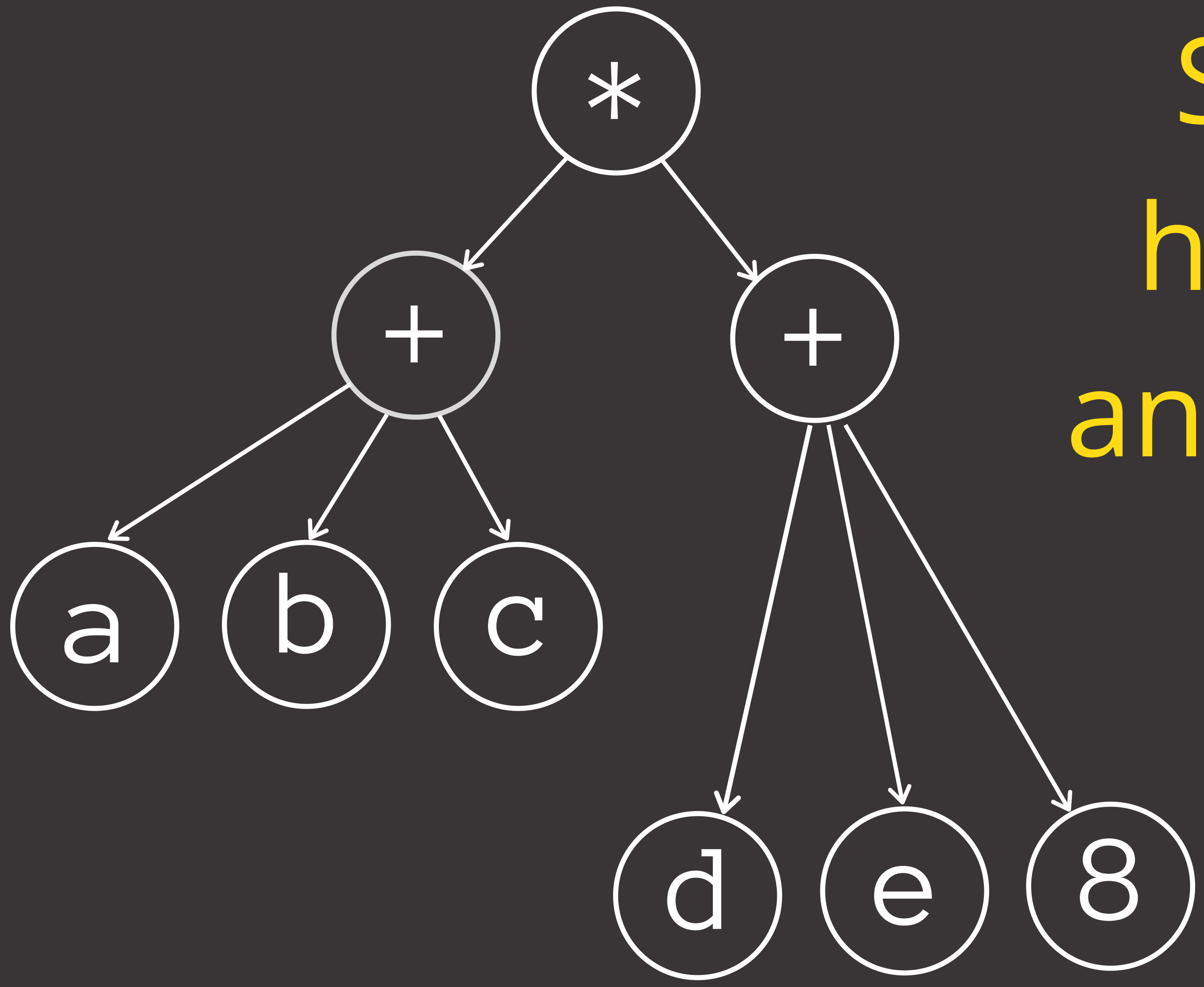
USING THE PREORDER AST



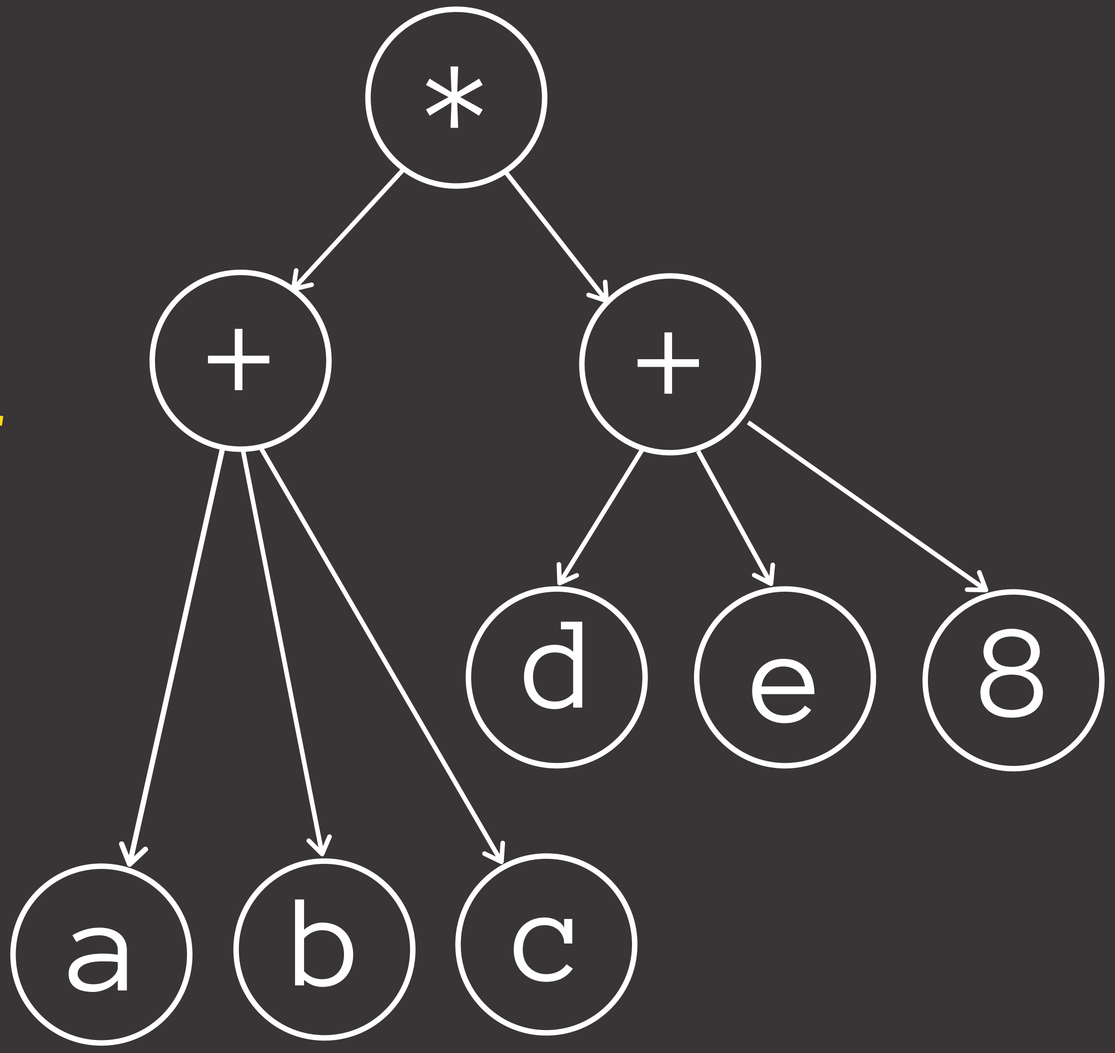
Step 5: Sort subtrees having a commutative and associative operator



SEMANTIC COMPARISON OF EXPRESSIONS
USING THE PREORDER AST



Step 5: Sort subtrees having a commutative and associative operator



Now compare the two ASTs node-by-node to check if the underlying expressions are equivalent.

CHALLENGES

USING THE PREORDER AST

Integer overflow due to re-association of expressions

```
(e1 + e2) + e3
```

Original expression may not overflow

```
e1 + (e2 + e3)
```

Expression may overflow after re-association!

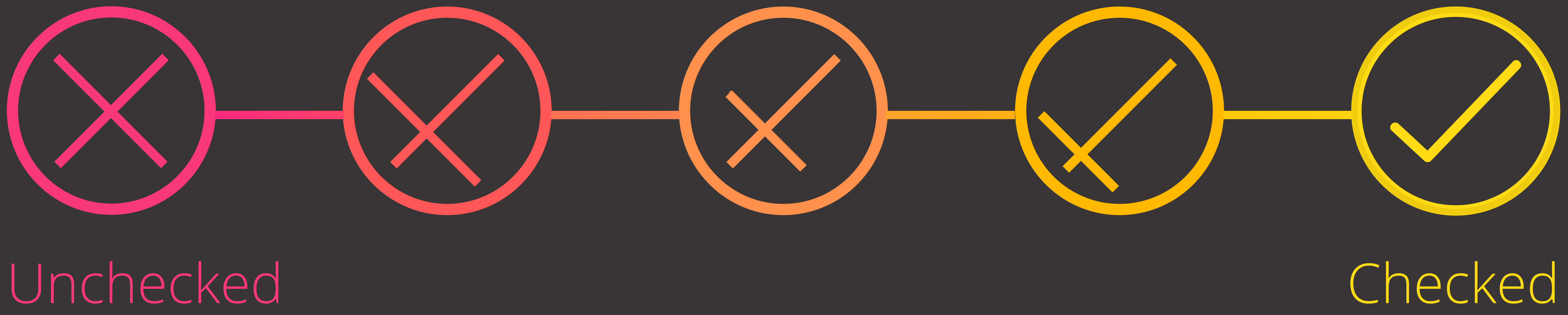
Possible Solution

`-fwrapv` Treat signed integer overflow as two's complement

What about pointer arithmetic overflow?

`-fwrapv-pointer` GCC has this flag

INCREMENTAL CONVERSION



WHAT ABOUT COMPATIBILITY?

Conversion without breaking compatibility?

Bounds-safe interface

BOUNDS-SAFE INTERFACES

https://bit.ly/35qXht0

CONVERSION SUPPORT

Port from legacy C a few lines at a time

ALTERNATE TYPES

Specify types for checked parameters

OPTIONAL BOUNDS

Checked arguments must meet bounds

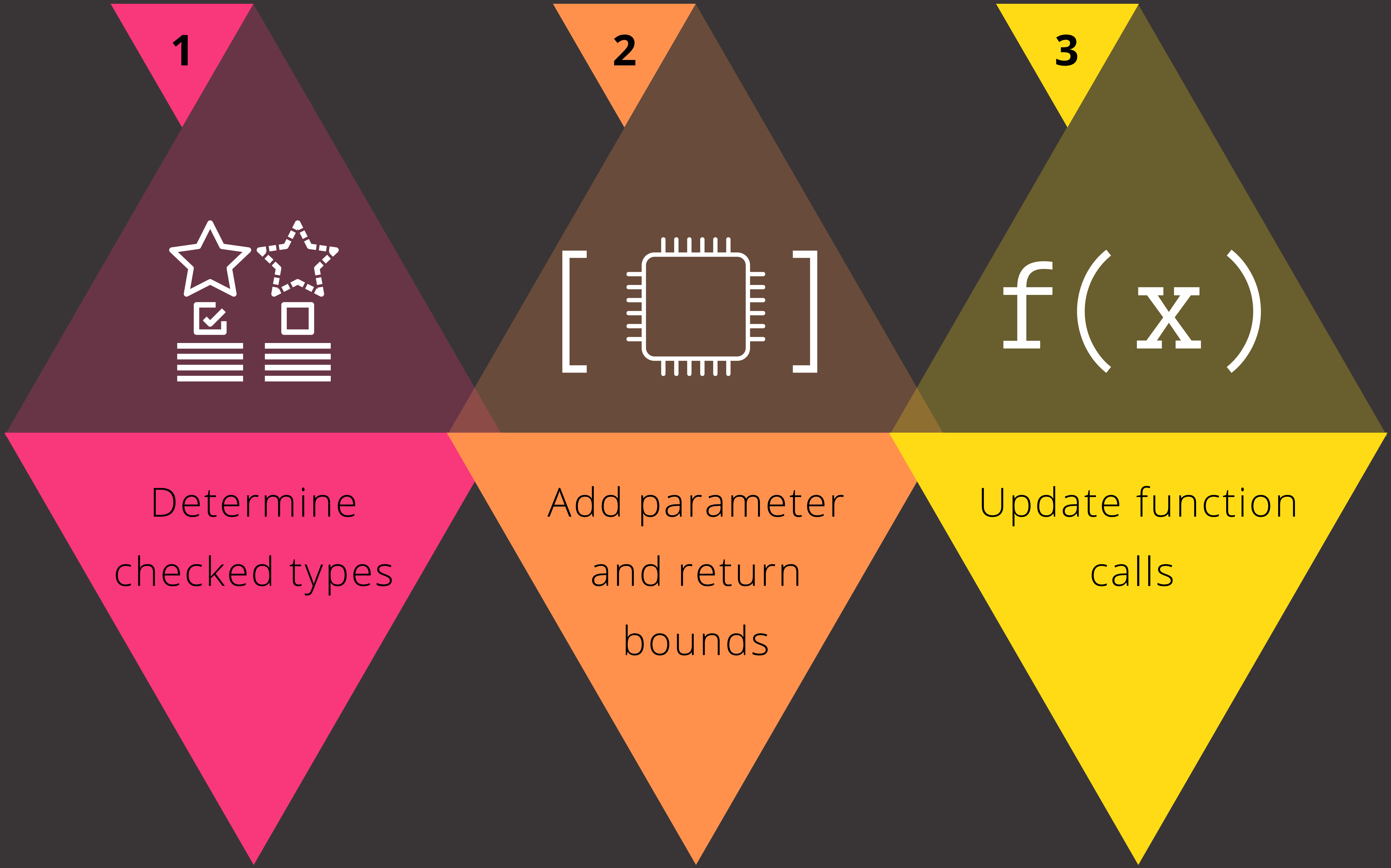
BACKWARDS COMPATIBLE

Accept unchecked pointer arguments

BOUNDS CHECKING

Check bounds for checked arguments

CONVERTING A FUNCTION

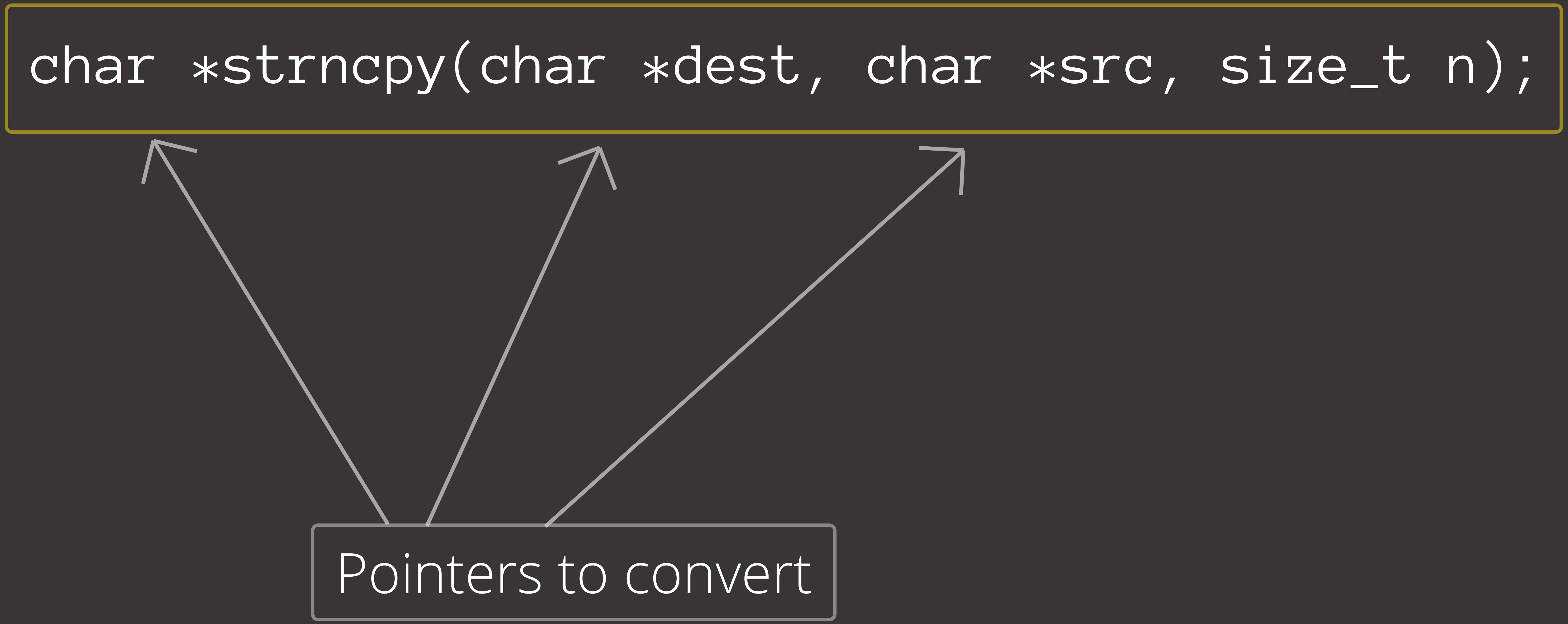


1 Determine checked types

2 Add parameter and return bounds

3 Update function calls

CONVERSION EXAMPLE



```
char *strncpy(char *dest, char *src, size_t n);
```

Pointers to convert

DETERMINE CHECKED TYPES

```
char *strncpy(
        char *dest : itype(_Nt_array_ptr<char>),
        char *src : itype(_Nt_array_ptr<char>),
        size_t n
    ) : itype(_Nt_array_ptr<char>);
```

ADD BOUNDS

```
// strncpy copies the first n characters of src into dest.
char *strncpy(
        char *dest : itype(_Nt_array_ptr<char>) count(n),
        char *src : itype(_Nt_array_ptr<char>) count(n),
        size_t n
    ) : itype(_Nt_array_ptr<char>) count(n);
```

UPDATE FUNCTION CALLS

```c
void unchecked_pointers() {
  // dest points to 3 characters including null terminator.
  char *dest = "12\0";
  // src points to 2 characters including null terminator.
  char *src = "1\0";
  // Fine - there is no bounds checking for dest and src.
  strncpy(dest, src, 3);
}
```

UPDATE FUNCTION CALLS

```
void checked_pointers() {
  // dest points to 3 characters including null terminator.
  _Nt_array_ptr<char> dest : count(3) = "12\0";
  // src points to 2 characters including null terminator.
  _Nt_array_ptr<char> src : count(2) = "1\0";
  // Fine - dest and src both point to at least 2 characters.
  strncpy(dest, src, 2);
  // Error: src points to 2 characters, expected to
  // point to at least 3.
  strncpy(dest, src, 3);
}
```

CHALLENGE: STRING LENGTHS

```
char *strupr(char *str);
```

```
char *strupr(char *str : itype(_Nt_array_ptr<char>));
```

```
char *strupr(char *str : itype(_Nt_array_ptr<char>) count(?));
```

CHALLENGE: STRING LENGTHS

This would be great...

```
char *strupr(
        char *str : itype(_Nt_array_ptr<char>)
                    count(strlen(str))
    );
```

...but it's not possible

WHY CAN'T WE USE STRLEN?

No modifying expressions are allowed in bounds

Function calls may modify memory

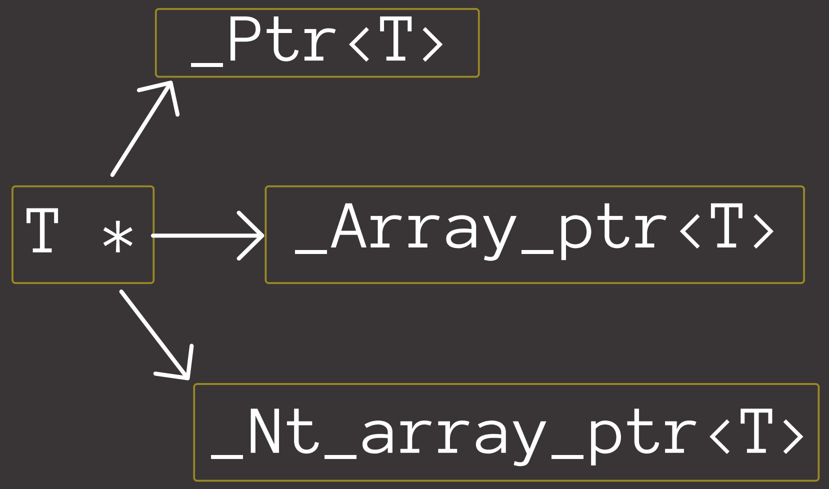
```
char *strupr(
        char *str : itype(_Nt_array_ptr<char>)
                    count(len),
        size_t len
    );
```

AUTOMATIC CONVERSION

CHECKEDC-CONVERT

https://bit.ly/32hTXOP

CONVERT POINTERS



UMD

Developed at the
University of Maryland

CONVERTING MUSL

2 interns

5 weeks

string subdirectory

network subdirectory

MUSL STRING LIBRARY

31

FUNCTIONS CONVERTED

316

LINES OF CODE CONVERTED

72

TOTAL FUNCTIONS

1574

TOTAL LINES OF CODE

MUSL NETWORK LIBRARY

51

FUNCTIONS CONVERTED

729

LINES OF CODE CONVERTED

65

TOTAL FUNCTIONS

3524

TOTAL LINES OF CODE

EVALUATION

LNT TESTS

Olden and Ptrdist benchmakrs

CODE SIZE

Impact on generated code

RUNTIME

Overhead introduced by dynamic checks

COMPILING

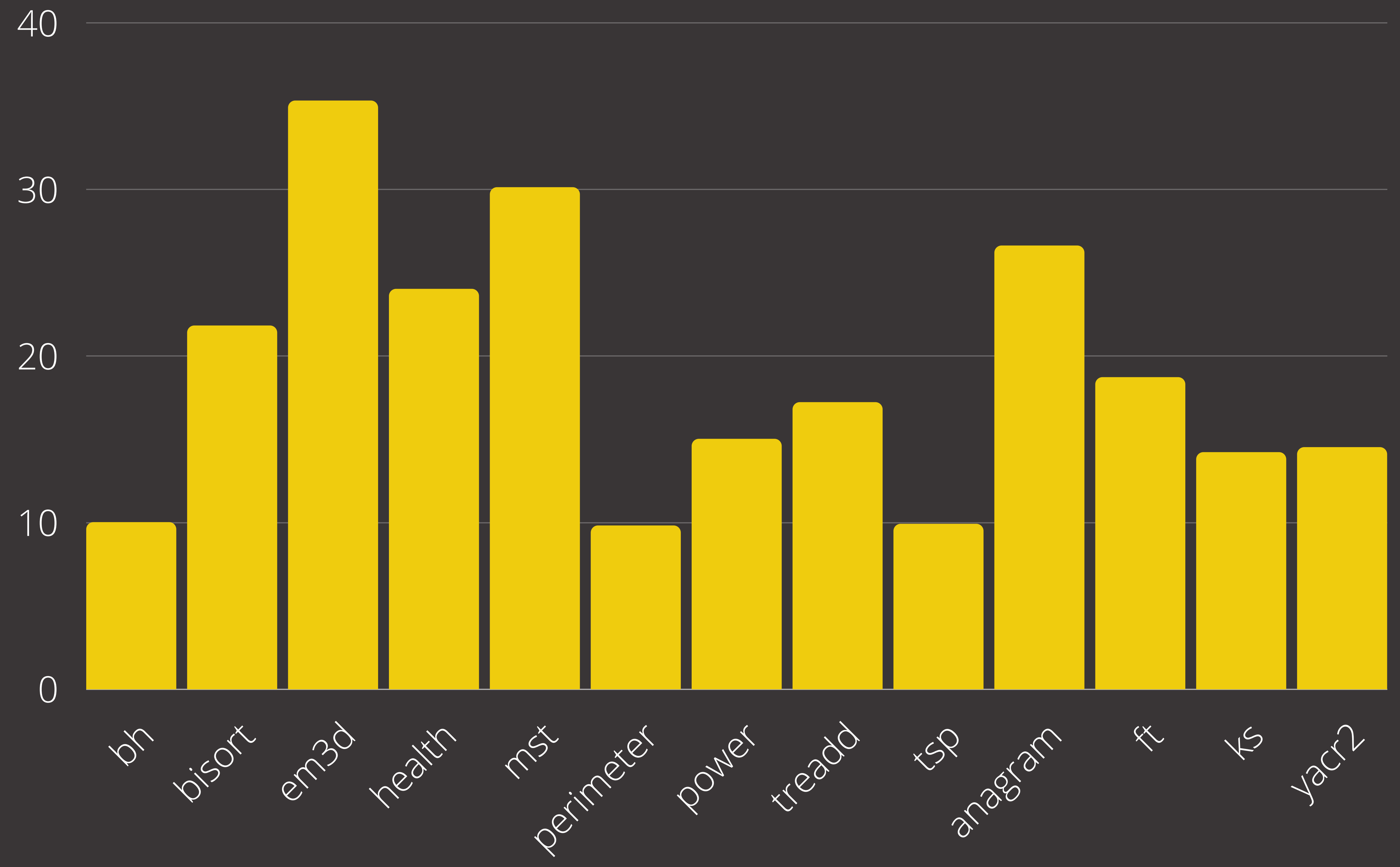
Impact on compilation time

LINES OF CODE MODIFIED

17.5%

Average LOC modified

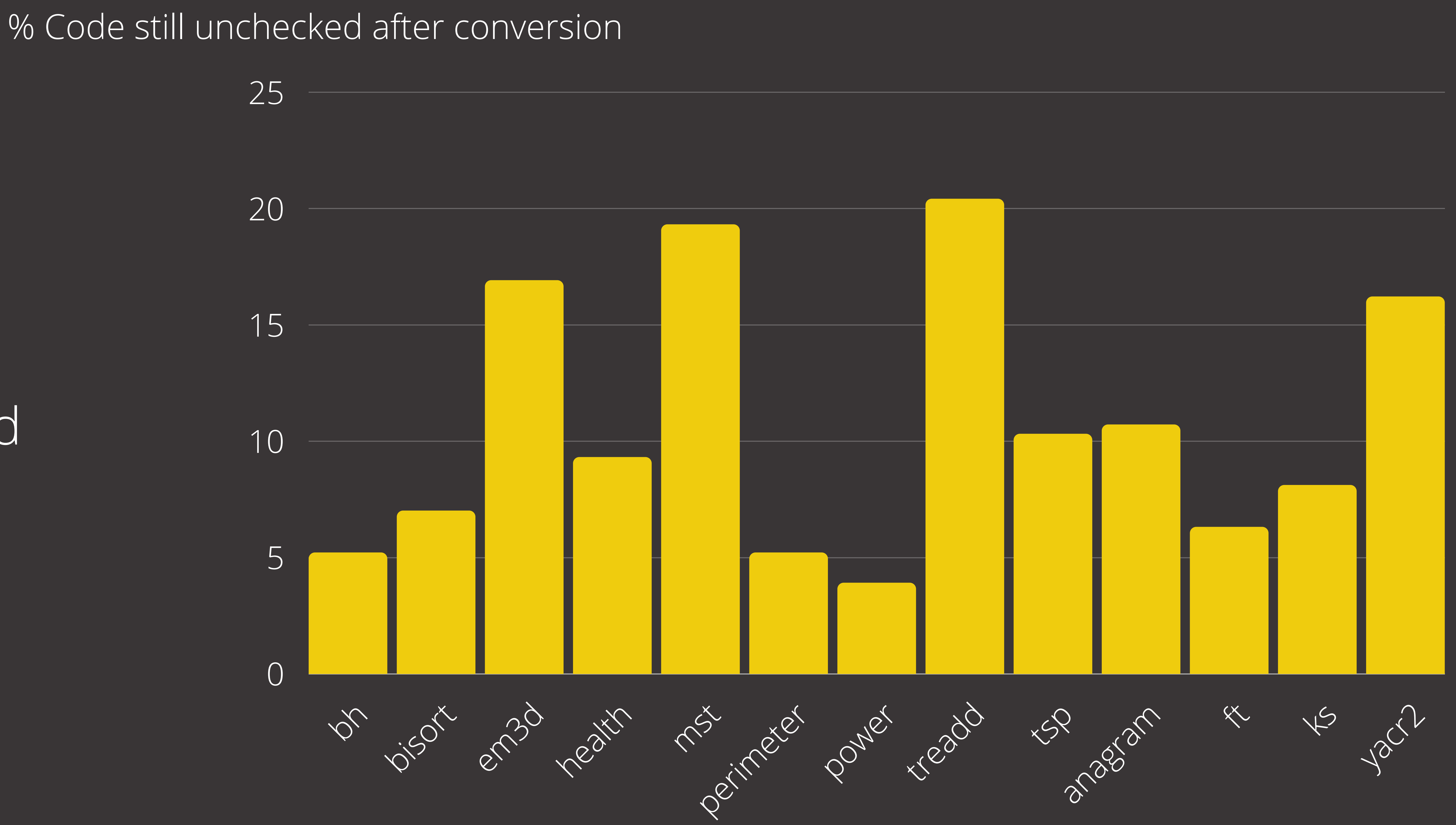
% Lines of code modified in converted test



REMAINING UNCHECKED CODE

9.3%

Average unchecked

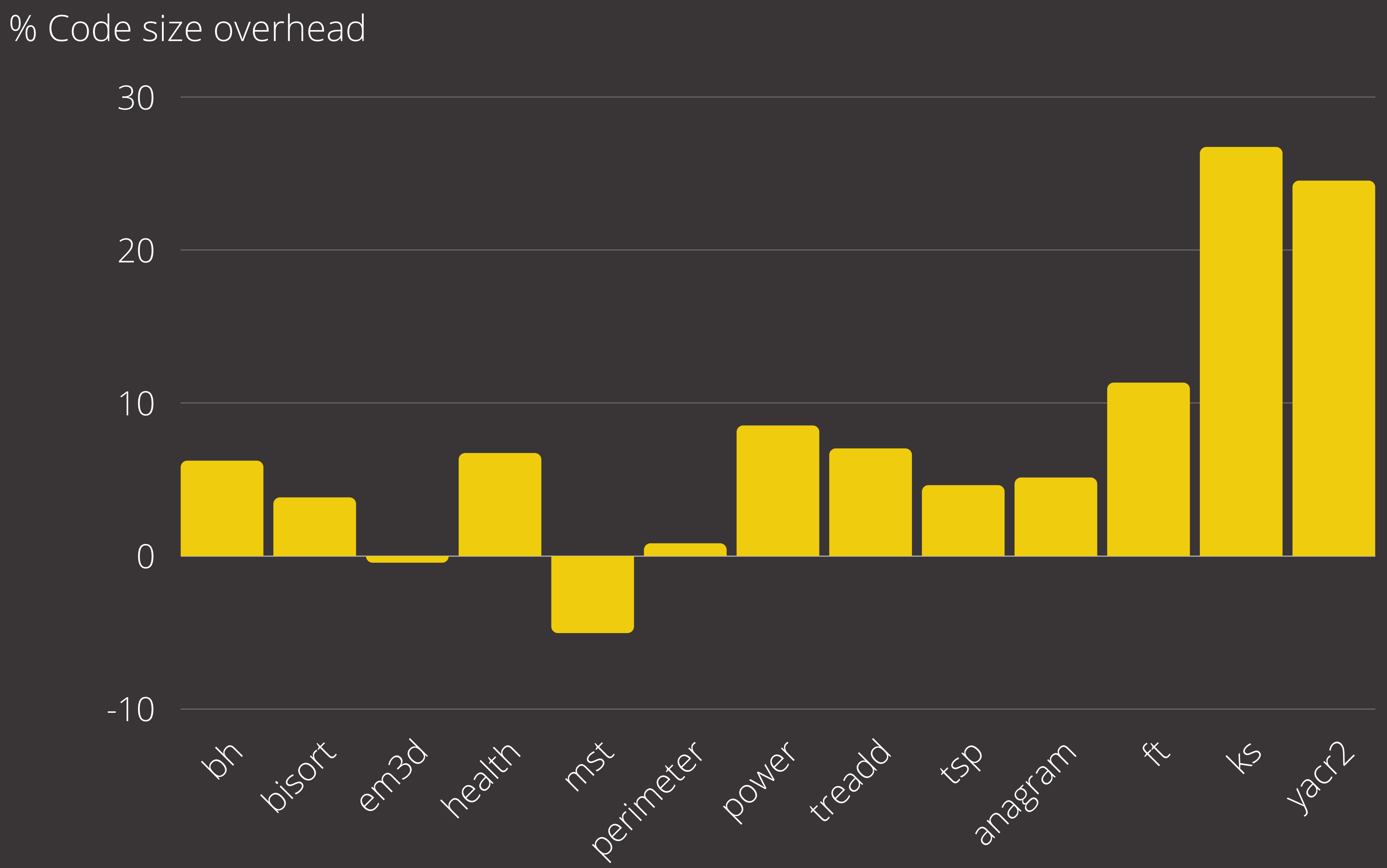


CODE SIZE

7.4%

Average overhead

Lower is better

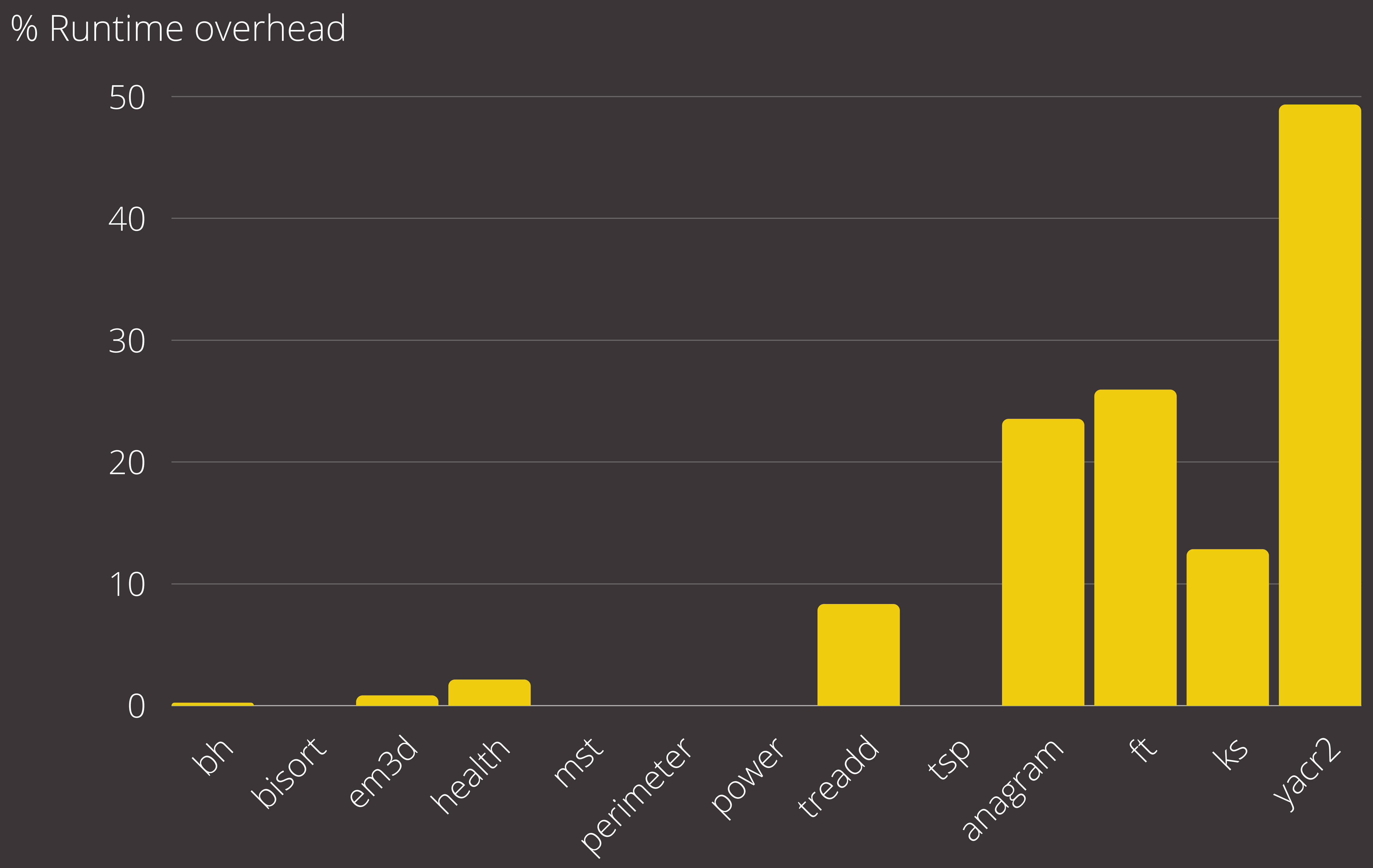


RUNTIME

8.6%

Average overhead

Lower is better

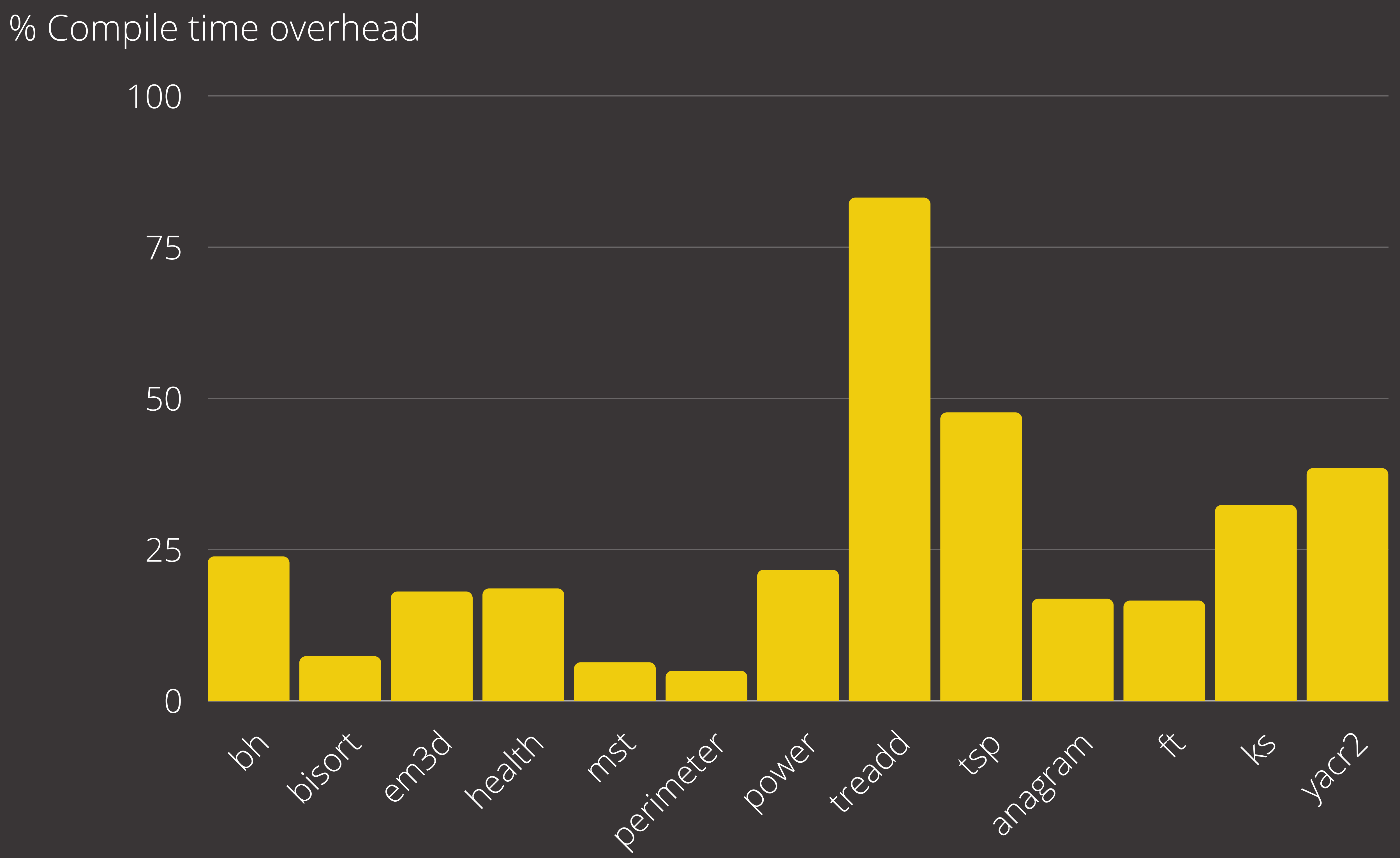


COMPILE TIME

24.3%

Average overhead

Lower is better



RESOURCES

Code Repository

https://bit.ly/2FrHkbh

Language Specification

https://bit.ly/2FmPyRO

SecDev 2018 Paper

https://bit.ly/2Zt2k8g