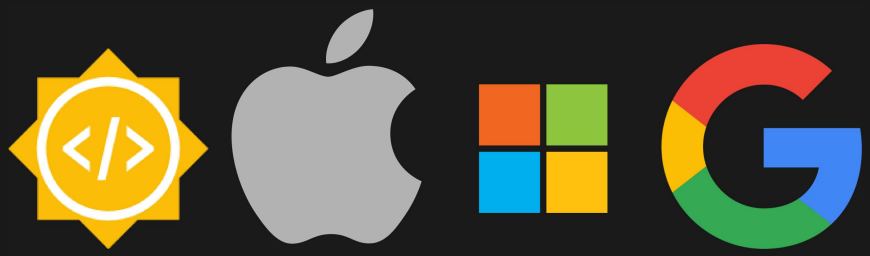
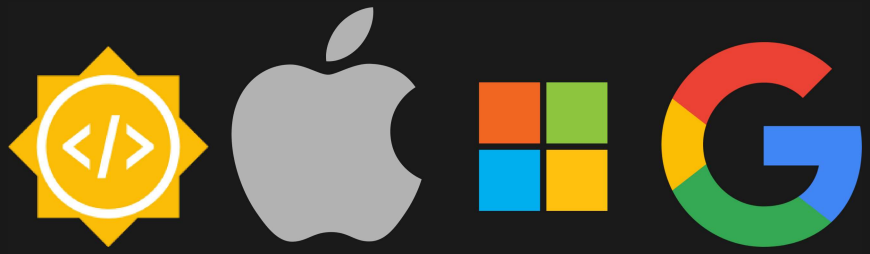
Algorithms from a Compiler Developer's Toolbox

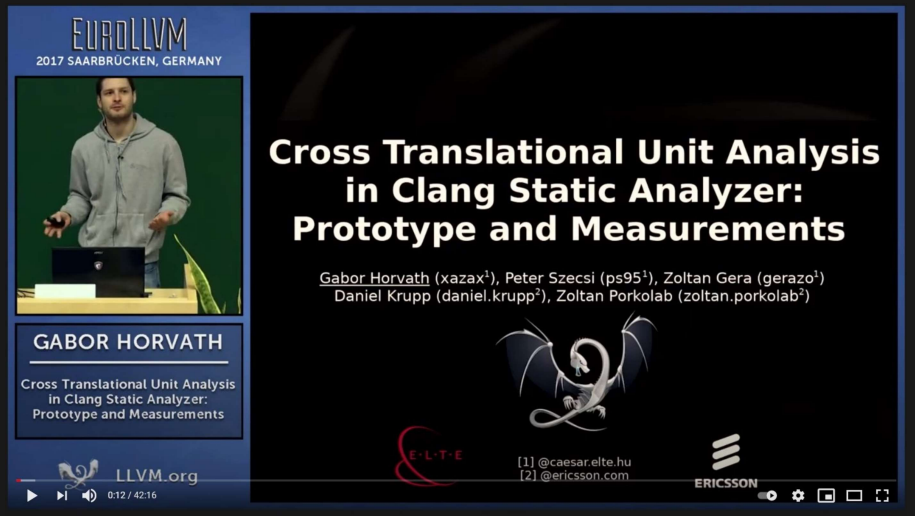
Gábor Horváth

ALGORITHMS FROM A COMPILER DEVELOPER'S TOOLBOX

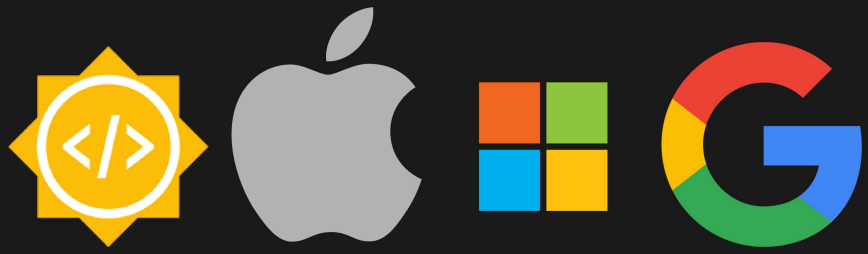
Gabor Horvath (He/Him)

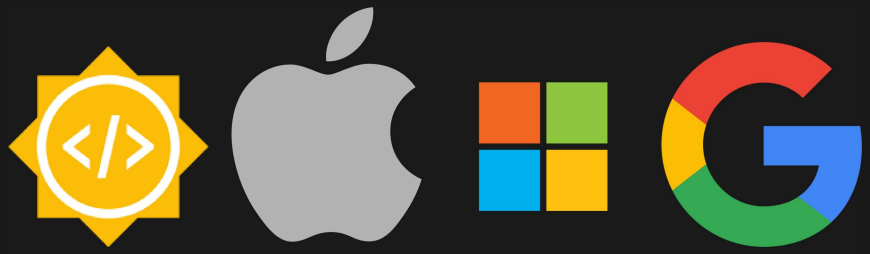


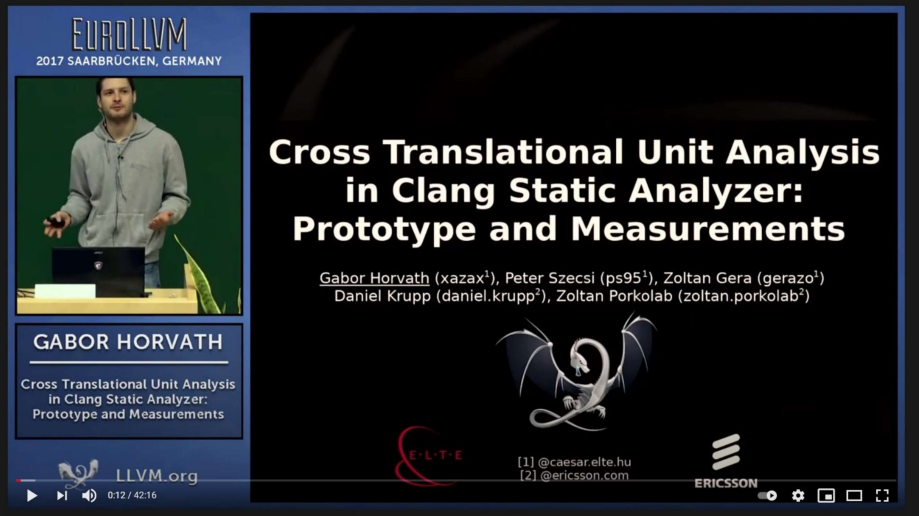




2017 EuroLLVM Developers' Meeting: G. Horvath "Cross Translational Unit Analysis in Clang ..."







2017 EuroLLVM Developers' Meeting: G. Horvath "Cross Translational Unit Analysis in Clang ..."

CppCon 2018: Herb Sutter "Thoughts on a more powerful and simpler C++ (5 of N)"

AGENDA

- Why study compilers?

AGENDA

- Why study compilers?
- Not just a constant factor!

AGENDA

- Why study compilers?
- Not just a constant factor!
- The beautiful algebra in our loops

AGENDA

- Why study compilers?
- Not just a constant factor!
- The beautiful algebra in our loops
- Sources of ~~redundant~~ redundancy

AGENDA

- Why study compilers?
- Not just a constant factor!
- The beautiful algebra in our loops
- Sources of ~~redundant~~ redundancy
- Counting values

AGENDA

- Why study compilers?
- Not just a constant factor!
- The beautiful algebra in our loops
- Sources of ~~redundant~~ redundancy
- Counting values
- Where to find more

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms
- Persistent data structures

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms
- Persistent data structures
- Finite-state automata

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms
- Persistent data structures
- Finite-state automata
- Scheduling

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms
- Persistent data structures
- Finite-state automata
- Scheduling
- SMT solvers

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms
- Persistent data structures
- Finite-state automata
- Scheduling
- SMT solvers
- Approximating NP-hard problems

WHY STUDY COMPILERS?

They use beautiful algorithms and data structures

- Graph algorithms
- Persistent data structures
- Finite-state automata
- Scheduling
- SMT solvers
- Approximating NP-hard problems
- Fixed-point iteration

WHY STUDY COMPILERS?

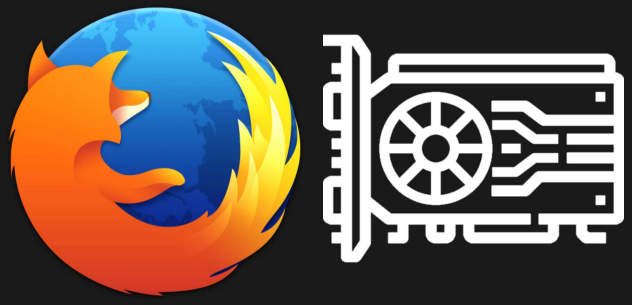
They are everywhere (and awesome)

WHY STUDY COMPILERS?

They are everywhere (and awesome)

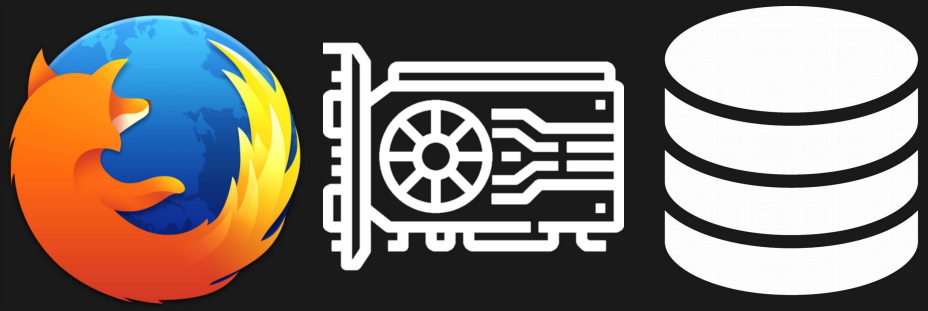
WHY STUDY COMPILERS?

They are everywhere (and awesome)



WHY STUDY COMPILERS?

They are everywhere (and awesome)



WHY STUDY COMPILERS?

They are everywhere (and awesome)

WHY STUDY COMPILERS?

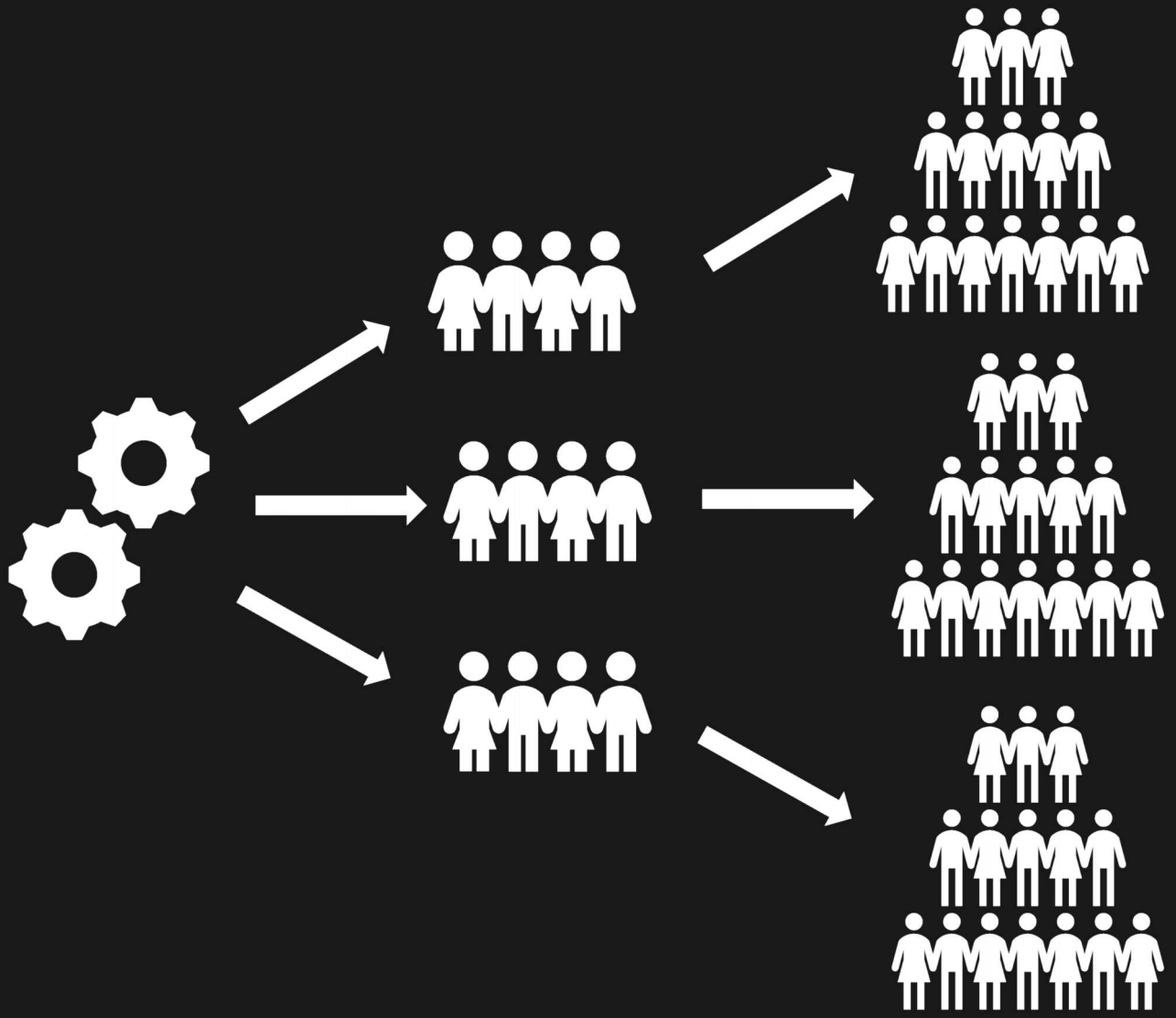
They are everywhere (and awesome)

WHY STUDY COMPILERS?

They are everywhere (and awesome)

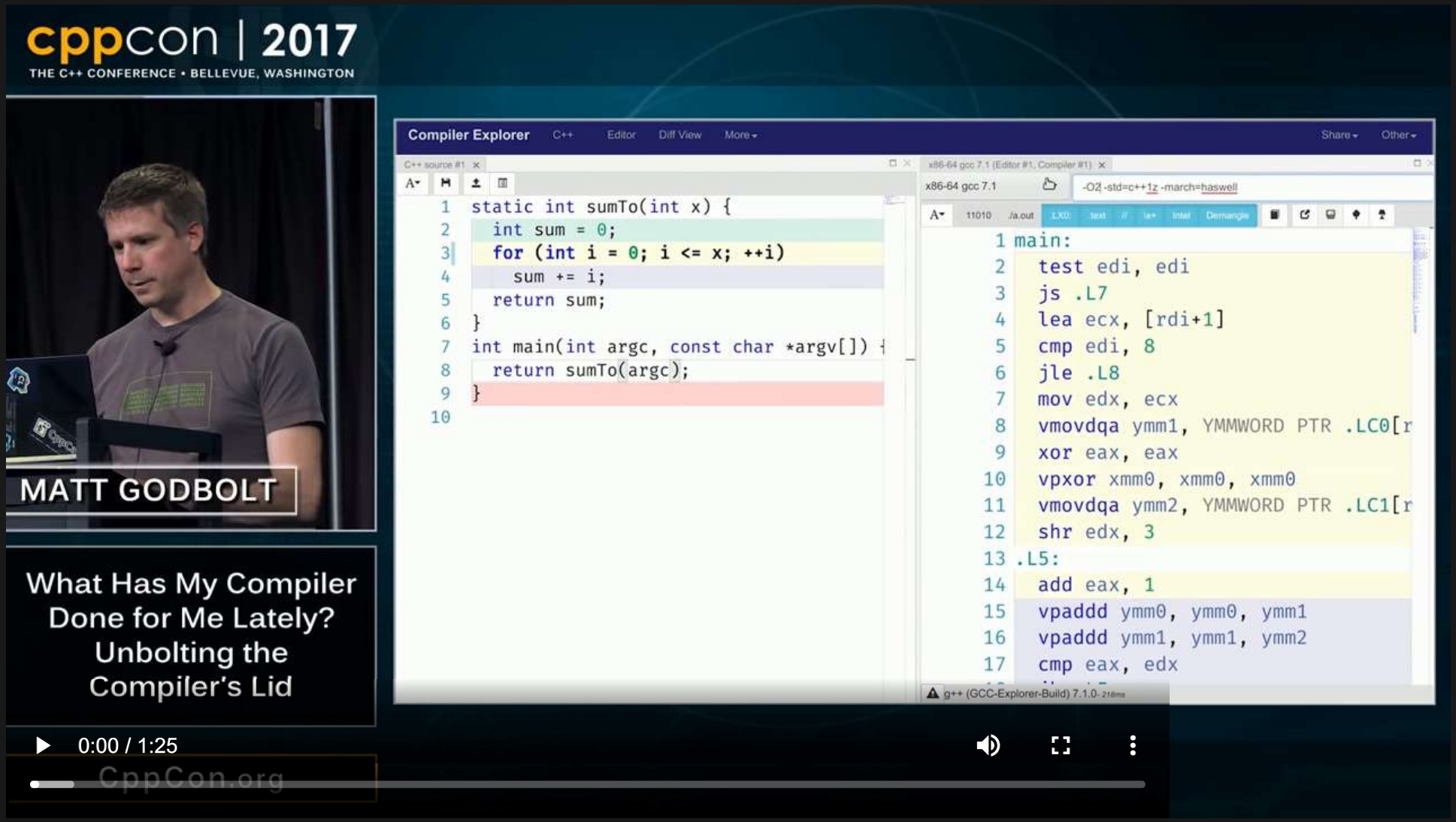
WHY STUDY COMPILERS?

WHY STUDY COMPILERS?



LOOP STRENGTH REDUCTION

CONSTANT FACTORS ONLY?



IS IT ROBUST?

LET'S UNBOLT THE COMPILER'S LID!

IT'S MATH

CHAINS OF RECURRENCES

RECURSIVE FUNCTIONS 101

RECURSIVE FUNCTIONS 101

RECURSIVE FUNCTIONS 101

$$n! = \begin{cases} 1, \text{ for } n = 0 \\ n * (n-1)! \text{ otherwise} \end{cases}$$

RECURSIVE FUNCTIONS 101

$$n! = \begin{cases} 1, & \text{for } n = 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

TERSE NOTATION

TERSE NOTATION

$$n! = \begin{cases} 1, \text{ for } n = 0 \\ n * (n-1)! \text{ otherwise} \end{cases}$$

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

$$f_v(i) = \{initial, +, increment\}$$

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

$$f_v(i) = \{initial, +, increment\}$$

Iter. increment

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

$$f_v(i) = \{initial, +, increment\}$$

Iter.		increment
0	initial	

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

$$f_v(i) = \{initial, +, increment\}$$

Iter.	increment
0	initial
1	initial + increment

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

$$f_v(i) = \{initial, +, increment\}$$

Iter.	increment
0	initial
1	initial + increment
2	initial + 2*increment

... BUT WHY?

```
int v = initial;
for(int i = 0; i < max; ++i)
    v += increment;
```

$$f_v(i) = \{initial, +, increment\}$$

Iter.	increment
0	initial
1	initial + increment
2	initial + 2*increment
3	initial + 3*increment

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, ?, ?\}$$

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, ?, ?\}$$

0.

1.

2.

3.

4.

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, ?, ?\}$$

0. 0
1. 1
2. 4
3. 9
4. 16

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, +, ?\}$$

0. 0
1. 1
2. 4
3. 9
4. 16

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, +, g(n)\}$$

$$g(n) = \{?, ?, ?\}$$

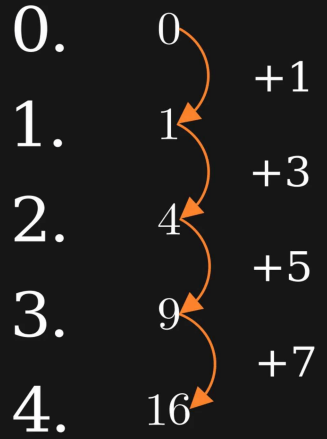
0. 0
1. 1
2. 4
3. 9
4. 16

... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, +, g(n)\}$$

$$g(n) = \{?, ?, ?\}$$

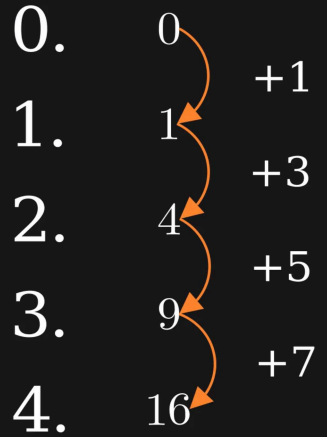


... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, +, g(n)\}$$

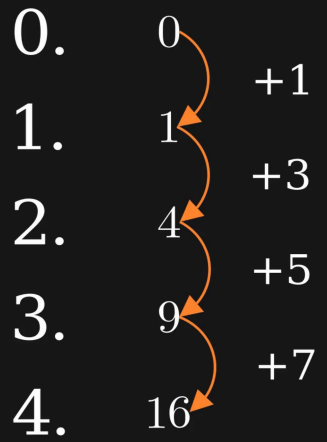
$$g(n) = \{1, +, 2\}$$



... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

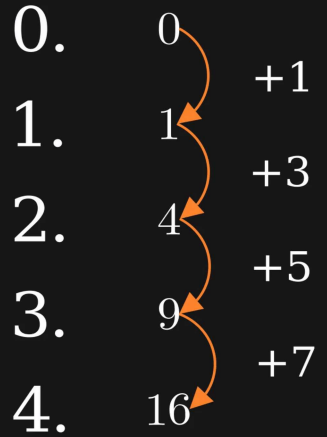
$$f_{i*i}(n) = \{0, +, \{1, +, 2\}\}$$



... MORE COMPLEX VALUES?

```
for(int i = 0; i < max; ++i)
    compute(i*i);
```

$$f_{i*i}(n) = \{0, +, 1, +, 2\}$$



INTERLUDE: WHAT IS AN ALGEBRA?

INTERLUDE: WHAT IS AN ALGEBRA?

From Arabic al-jabr: reunion of broken parts.

INTERLUDE: WHAT IS AN ALGEBRA?

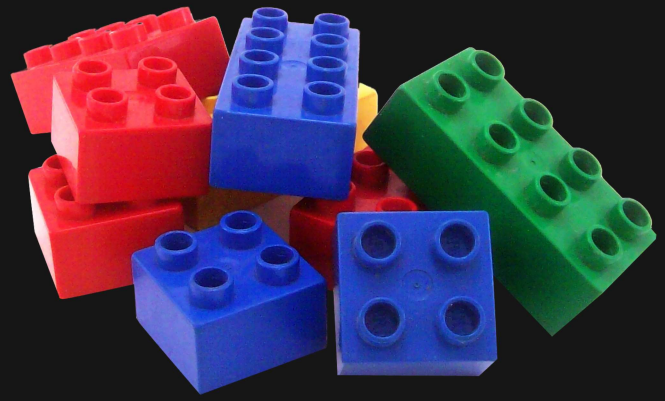
From Arabic al-jabr: reunion of broken parts.

Study of symbols and the rules for manipulating symbols.

INTERLUDE: WHAT IS AN ALGEBRA?

From Arabic al-jabr: reunion of broken parts.

Study of symbols and the rules for manipulating symbols.



THE LEGO ALGEBRA

THE LEGO ALGEBRA

- Closed

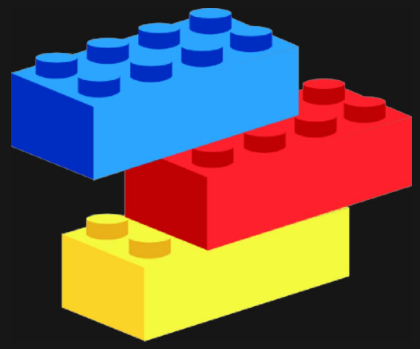
```
stack :: lego -> lego -> lego
```

THE LEGO ALGEBRA

- Closed

  ```
  stack :: lego -> lego -> lego
  ```

- Rules

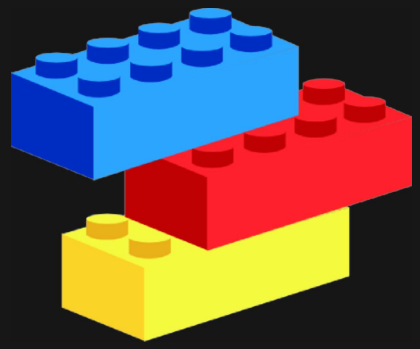


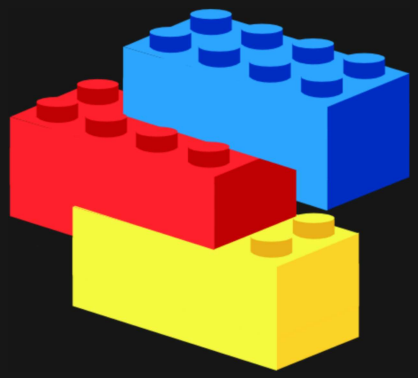
THE LEGO ALGEBRA

- Closed

  ```
  stack :: lego -> lego -> lego
  ```

- Rules





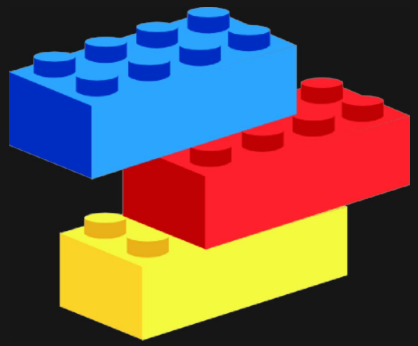
THE LEGO ALGEBRA

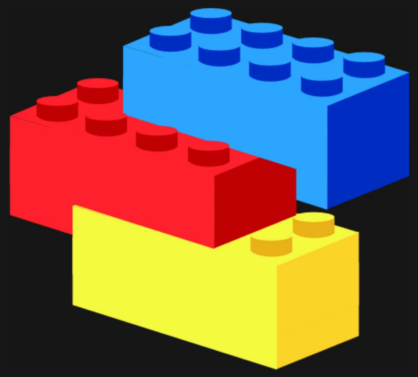
- Closed

  ```
  stack :: lego -> lego -> lego
  ```

- Rules
- Identities

  ```
  stack Y (stack R B) == stack (stack Y R) B
  ```





OPERATIONS, IDENTITIES

OPERATIONS, IDENTITIES

$$c + \{a, \ + \ , b\} =$$

OPERATIONS, IDENTITIES

$$c + \{a, \ + \ , b\} = \{a + c, \ + \ , b\}$$

OPERATIONS, IDENTITIES

$$c + \{a, +, b\} = \{a + c, +, b\}$$

$$c * \{a, +, b\} =$$

OPERATIONS, IDENTITIES

$$c + \{a, +, b\} = \{a + c, +, b\}$$

$$c * \{a, +, b\} = \{a * c, +, b * c\}$$

OPERATIONS, IDENTITIES

$$c + \{a, +, b\} = \{a + c, +, b\}$$

$$c * \{a, +, b\} = \{a * c, +, b * c\}$$

$$\{a, +, b\} + \{e, +, f\} =$$

OPERATIONS, IDENTITIES

$$c + \{a, \ + \ , b\} = \{a + c, \ + \ , b\}$$

$$c * \{a, \ + \ , b\} = \{a * c, \ + \ , b * c\}$$

$$\{a, \ + \ , b\} + \{e, \ + \ , f\} = \{a + e, \ + \ , b + f\}$$

OPERATIONS, IDENTITIES

$$c + \{a, \ + \ , b\} = \{a + c, \ + \ , b\}$$

$$c * \{a, \ + \ , b\} = \{a * c, \ + \ , b * c\}$$

$$\{a, \ + \ , b\} + \{e, \ + \ , f\} = \{a + e, \ + \ , b + f\}$$

$$\{a, \ + \ , b\} * \{e, \ + \ , f\} =$$

OPERATIONS, IDENTITIES

$$c + \{a, +, b\} = \{a + c, +, b\}$$

$$c * \{a, +, b\} = \{a * c, +, b * c\}$$

$$\{a, +, b\} + \{e, +, f\} = \{a + e, +, b + f\}$$

$$\{a, +, b\} * \{e, +, f\} = \{a * e, +, a * f + e * b + b * f, +$$

OPERATIONS, IDENTITIES

$$c + \{a, +, b\} = \{a + c, +, b\}$$

$$c * \{a, +, b\} = \{a * c, +, b * c\}$$

$$\{a, +, b\} + \{e, +, f\} = \{a + e, +, b + f\}$$

$$\{a, +, b\} * \{e, +, f\} = \{a * e, +, a * f + e * b + b * f, +$$

$$c = \{c, +, 0\} = \{c\}$$

OPERATIONS, IDENTITIES

$$\{a, +, b\} + \{e, +, f\} = \{a + e, +, b + f\}$$

$$\{a, +, b\} * \{e, +, f\} = \{a * e, +, a * f + e * b + b * f, +$$

$$c = \{c, +, 0\} = \{c\}$$

... MORE COMPLEX VALUES, REVISITED!

```
for(i=0; i < m; ++i)
  compute(i*i);
```

... MORE COMPLEX VALUES, REVISITED!

```
for(i=0; i < m; ++i)
  compute(i*i);
```

$f_i(n)$

f_i

... MORE COMPLEX VALUES, REVISITED!

```
for(i=0; i < m; ++i)
  compute(i*i);
```

$$f_i(n)$$

$$f_{i*i}(n) = \{0$$

... MORE COMPLEX VALUES, REVISITED!

```
for(i=0; i < m; ++i)
  compute(i*i);
```

$$f_i(n)$$

$$f_{i*i}(n) = \{0$$

$$\{a, +, b\} * \{e, +, f\} = \{a * e$$

... MORE COMPLEX VALUES, REVISITED!

```
for(i=0; i < m; ++i)
  compute(i*i);
```

$$f_i(n)$$

$$a \leftarrow 0 b \leftarrow 1 e \leftarrow 0 f \leftarrow 1$$

$$f_{i*i}(n) = \{0$$

$$\{a, +, b\} * \{e, +, f\} = \{a * e$$

$$\{0, +, 1\} * \{0, +, 1\} = \{0 * 0,$$

... MORE COMPLEX VALUES, REVISITED!

```
for(i=0; i < m; ++i)
  compute(i*i);
```

$$f_i(n)$$

$$a \leftarrow 0 b \leftarrow 1 e \leftarrow 0 f \leftarrow 1$$

$$f_{i*i}(n) =$$

$$\{a, +, b\} * \{e, +, f\} = \{a * e$$

$$\{0, +, 1\} * \{0, +, 1\} = \{0 * 0,$$

$$\{0, +, 1\} * \{0, +$$

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_i(n) = \{0, +, 1\}$$

$$f_i^2(n) = \{0, +, 1, +, 2\}$$

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_i(n) = \{0, +, 1\}$$

$$f_i{}^2(n) = \{0, +, 1, +, 2\}$$

$$f_{2i}(n) = ?$$

$$f_{i+1}(n) = ?$$

$$f_{(i+1)^2}(n) = ?$$

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_i(n) = \{0, +, 1\}$$

$$f_i{}^2(n) = \{0, +, 1, +, 2\}$$

$$f_{2i}(n) = \{0, +, 2\}$$

$$f_{i+1}(n) = \{1, +, 1\}$$

$$f_{(i+1)^2}(n) = \{1, +, 3, +, 2\}$$

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_i(n) = \{0, +, 1\}$$

$$f_i{}^2(n) = \{0, +, 1, +, 2\}$$

$$f_{2i}(n) = \{0, +, 2\}$$

$$f_{i+1}(n) = \{1, +, 1\}$$

$$f_{(i+1)^2}(n) = \{1, +, 3, +, 2\}$$

$$f_{(i+1)^2 - i^2}(n) = ?$$

$$f_{(i+1)^2-i^2-2i}(n) = ?$$

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_i(n) = \{0, +, 1\}$$

$$f_{i^2}(n) = \{0, +, 1, +, 2\}$$

$$f_{2^i}(n) = \{0, +, 2\}$$

$$f_{i+1}(n) = \{1, +, 1\}$$

$$f_{(i+1)^2}(n) = \{1, +, 3, +, 2\}$$

$$f_{(i+1)^2 - i^2}(n) = \{1, +, 2, +, 0\}$$

$$f_{(i+1)^2-i^2-2i}(n) = \{1, +, 0\}$$

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_{(i+1)^2-i^2-2i}(n) = \{1\}$$

```
1 for(i=0; i < m; ++i)
2   v[i] = 1;
```

THE POWER OF ALGEBRAS

```
for(i=0; i < m; ++i)
  v[i] = (i + 1)*(i + 1) - i*i - 2*i;
```

$$f_{(i+1)^2-i^2-2i}(n) = \{1\}$$

```
1 for(i=0; i < m; ++i)
2   v[i] = 1;
```

$$(i+1)^2 = i^2 + 2i + 1$$

CLOSED FROMS

$$\{a, +, b\}$$

CLOSED FROMS

$$\{a, \ + \ , b\}$$

$$a, a + b, a + 2b, a + 3b, \ldots$$

CLOSED FROMS

$$\{a, \ + \ , b\}$$

$$a, a + b, a + 2b, a + 3b, \ldots$$

Arithmetic series with the difference of b!

CLOSED FROMS

$$\{a, \ +, b\}$$

$$a, a + b, a + 2b, a + 3b, \ldots$$

Arithmetic series with the difference of b!

$$f(n) = \{c, \ +, a, \ +, b\}$$

CLOSED FROMS

$$\{a, \ + \ , b\}$$

$$a, a + b, a + 2b, a + 3b, \ldots$$

Arithmetic series with the difference of b!

$$f(n) = \{c, \ + \ , a, \ + \ , b\}$$

$$f(n) = c + \sum_{i=0}^{n} \{a, \ + \ , b\}$$

CLOSED FROMS

$$\{a, \ +, b\}$$

$$a, a + b, a + 2b, a + 3b, \ldots$$

Arithmetic series with the difference of b!

$$f(n) = \{c, \ +, a, \ +, b\}$$

$$f(n) = c + \frac{n[2a + b(n-1)]}{2}$$

CLOSED FROMS

$$\{a, \ +, b\}$$

$$a, a + b, a + 2b, a + 3b, \ldots$$

Arithmetic series with the difference of b!

$$f(n) = \{c, \ +, a, \ +, b\}$$

$$f(n) = c + \frac{n[2a + b(n-1)]}{2}$$

There are more...

STRENGTH REDUCTION REVISITED

```
void g(int);
void f() {
  for(int i = 0; i < 20; i += 1
  {
    g(5 * i);
  }
}
```

STRENGTH REDUCTION REVISITED

```
void g(int);
void f() {
  for(int i = 0; i < 20; i +=
  {
    g(5 * i);
  }
}
```

$$f_{5i}(n) = 5 * \{0, +, 1\} = \{0, +$$

STRENGTH REDUCTION REVISITED

```
void g(int);
void f() {
  for(int i = 0; i < 20; i +=
  {
    g(5 * i);
  }
}
```

$$f_{5i}(n) = 5 * \{0, +, 1\} = \{0, +$$

We have the strength reduc
format for free.

STRENGTH REDUCTION REVISITED

```
void g(int);
void f() {
  for(int i = 0; i < 100; i +=
  {
    g(i);
  }
}
```

$$f_{5i}(n) = 5 * \{0, +, 1\} = \{0, +$$

We have the strength reduc
format for free.

STRENGTH REDUCTION REVISITED

```
void g(int *);
void f() {
  int t[20];
  for(int i = 0; i < 20; i += 1
  {
    t[i] = (i + 1) * (i + 1) +
                3*i - 5;
  }
  g(t);
}
```

STRENGTH REDUCTION REVISITED

```
void g(int *);
void f() {
  int t[20];
  for(int i = 0; i < 20; i +=
  {
    t[i] = (i + 1) * (i + 1)
                3*i - 5;
  }
  g(t);
}
```

Initially: 4 additions, 2 multiplications

STRENGTH REDUCTION REVISITED

```
void g(int *);
void f() {
  int t[20];
  for(int i = 0; i < 20; i +=
  {
    t[i] = (i + 1) * (i + 1)
                3*i - 5;
  }
  g(t);
}
```

Initially: 4 additions, 2 multiplications

$$f_{(i+1)^2+3i-5}(n) = \{-4, +, 6,$$

STRENGTH REDUCTION REVISITED

```
void g(int *);
void f() {
  int t[20];
  a = -4;
  b = 6;
  for(int i = 0; i < 20; i +=
  {
    t[i] = a;
    a += b;
    b += 2;
  }
  g(t);
}
```

Initially: 4 additions, 2 multiplications

$$f_{(i+1)^2+3i-5}(n) = \{-4, +, 6,$$

After: 2 additions

CLOSED FORM SUM REVISITED

$$f_{result}(n) = \{0, +, 0, +, 1\}$$

LLVM SCEV DEMO

WHERE TO LEARN MORE?

GCC

Academic papers

`llvm::ScalarEvolution`

RECAPPING CHAINS OF RECURRENCES

RECAPPING CHAINS OF RECURRENCES

- Great to model some loop variant values

RECAPPING CHAINS OF RECURRENCES

- Great to model some loop variant values
- Algebra of simple recursive functions

RECAPPING CHAINS OF RECURRENCES

- Great to model some loop variant values
- Algebra of simple recursive functions
- Algebraic simplifications

RECAPPING CHAINS OF RECURRENCES

- Great to model some loop variant values
- Algebra of simple recursive functions
- Algebraic simplifications
- Strength reduction

RECAPPING CHAINS OF RECURRENCES

- Great to model some loop variant values
- Algebra of simple recursive functions
- Algebraic simplifications
- Strength reduction
- Closed forms

RECAPPING CHAINS OF RECURRENCES

- Great to model some loop variant values
- Algebra of simple recursive functions
- Algebraic simplifications
- Strength reduction
- Closed forms
- Many more...

VALUE NUMBERING

SOURCES OF REDUNDANCY

```
1 int calculate(int a, int b) {
2         int result = a * b + 2;
3         if (a % 2 == 0) {
4                 result += a * b;
5         }
6         return result;
7 }
```

SOURCES OF REDUNDANCY

```
1 int calculate(int a, int b) {
2         int tmp = a * b;
3         int result = tmp + 2;
4         if (a % 2 == 0) {
5                 result += tmp;
6         }
7         return result;
8 }
```

SOURCES OF REDUNDANCY

```
1 int foo() {
2     int matrix[5][5] = {...};
3
4     matrix[1][2] = bar();
5     matrix[1][3] = baz();
6
7     calculate(matrix);
8 }
```

SOURCES OF REDUNDANCY

```
1 int foo() {
2     int matrix[5][5] = {...};
3
4     matrix[1][2] = bar();
5     matrix[1][3] = baz();
6
7     calculate(matrix);
8 }
```

```
*((int*)matrix + ROW * sizeof(int) * 1 + sizeof(int) * 2) = ba
*((int*)matrix + ROW * sizeof(int) * 1 + sizeof(int) * 3) = ba
```

SOURCES OF REDUNDANCY

```c
int dead_code() {
    int a;
    a = 5; // dead
    a = 7;
    return a;
}

int copy_propagation() {
    int x = get_value();
    int y = x;
    int z = y;
    return z;
}
```

SOURCES OF REDUNDANCY

```c
int dead_code() {
    int a;
    a = 7;
    return a;
}

int copy_propagation() {
    int x = get_value();
    return x;
}
```

PASSES OF A COMPILER

BRIL

https://github.com/sampsyo/bril

Bril (the Big Red Intermediate Language) is a compiler IR made for teaching CS 6120, a grad compilers course. It is an extremely simple instruction-based IR that is meant to be extended. Its canonical representation is JSON, which makes it easy to build tools from scratch to manipulate it.

BRIL EXAMPLE

BRIL EXAMPLE

```
> cat add.bril
@main {
  v0: int = const 1;
  v1: int = const 2;
  v2: int = add v0 v1;
  print v2;
}
```

BRIL EXAMPLE

```
> cat add.bril | bril2json
{
  "functions": [
    {
      "instrs": [
        {
          "dest": "v0",
          "op": "const",
          "type": "int",
          "value": 1
        },
        {
          "dest": "v1",
          "op": "const",
          "type": "int",
          "value": 2
        },
        {
          "args": [
            "v0",
            "v1"
          ],
          "dest": "v2",
          "op": "add",
          "type": "int"
        },
        {
          "args": [
            "v2"
          ],
          "op": "print"
        }
      ],
      "name": "main"
    }
  ]
}
```

BRIL EXAMPLE

```
> cat add.bril | bril2json | brili
3
```

BRIL EXAMPLE

```
> cat negate_add.py
import json
import sys

ir = json.load(sys.stdin)

for func in ir["functions"]:
    for instr in func["instrs"]:
        if instr["op"] == "add":
            instr["op"] = "sub"

print(json.dumps(ir, indent=4))
```

BRIL EXAMPLE

```
> cat add.bril | bril2json | python3 negate_add.py | bril2txt
@main {
  v0: int = const 1;
  v1: int = const 2;
  v2: int = sub v0 v1;
  print v2;
}
```

BRIL EXAMPLE

```
> cat add.bril | bril2json | python3 negate_add.py | brili
-1
```

BRIL EXAMPLE

```
> cat add.bril | bril2json | python3 negate_add.py | python3 negate_add.py | python3 negate_add.py | brili
-1
```

BRIL RECAP

BRIL RECAP

- No compilation, fast experimenting

BRIL RECAP

- No compilation, fast experimenting
- Small core language, trivial API

BRIL RECAP

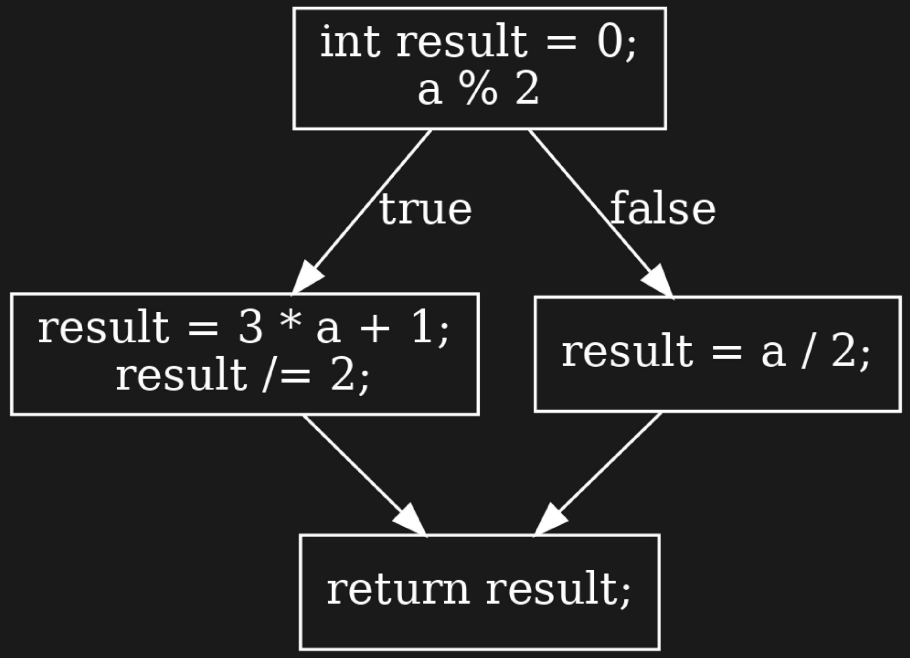
- No compilation, fast experimenting
- Small core language, trivial API
- Easily composable pipeline

BRIL RECAP

- No compilation, fast experimenting
- Small core language, trivial API
- Easily composable pipeline
- Tooling included

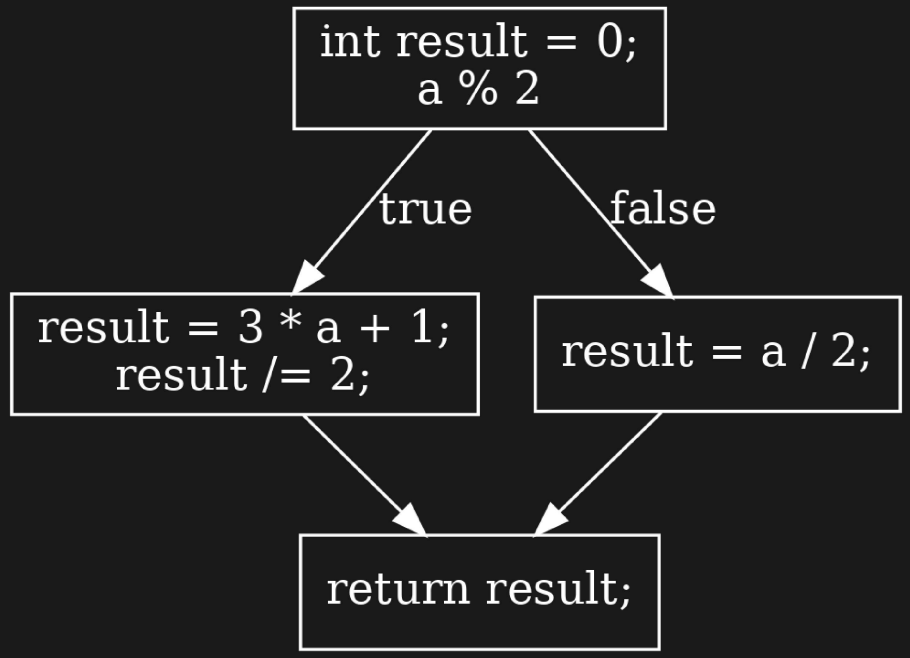
OPTIMIZATION TERMINOLOGY 101

```
int collatz(int a) {
  int result = 0;
  if (a % 2) {
    result = 3 * a + 1;
    result /= 2;
  } else {
    result = a/2;
  }
  return result;
}
```



OPTIMIZATION TERMINOLOGY 101

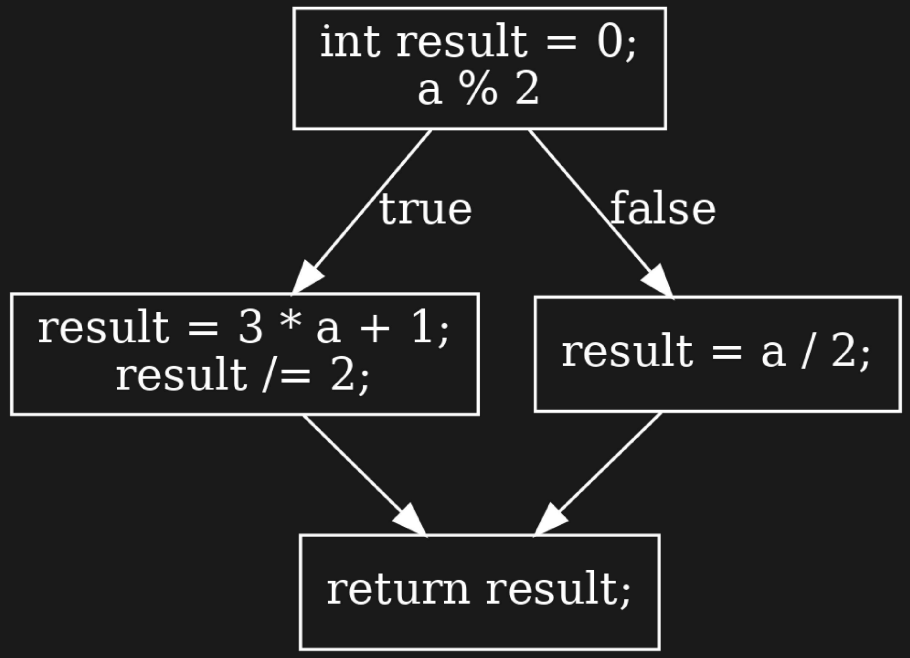
```
int collatz(int a) {
  int result = 0;
  if (a % 2) {
    result = 3 * a + 1;
    result /= 2;
  } else {
    result = a/2;
  }
  return result;
}
```



- CFG, Basic Block

OPTIMIZATION TERMINOLOGY 101

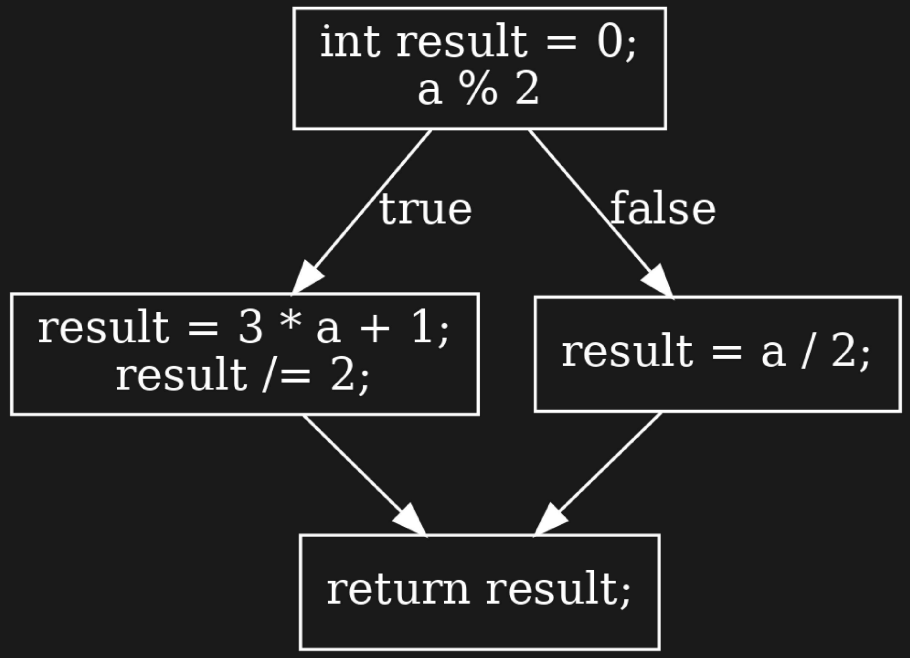
```
int collatz(int a) {
  int result = 0;
  if (a % 2) {
    result = 3 * a + 1;
    result /= 2;
  } else {
    result = a/2;
  }
  return result;
}
```



- CFG, Basic Block
- Local, super-local (loops, EBBs), global

OPTIMIZATION TERMINOLOGY 101

```
int collatz(int a) {
  int result = 0;
  if (a % 2) {
    result = 3 * a + 1;
    result /= 2;
  } else {
    result = a/2;
  }
  return result;
}
```



- CFG, Basic Block
- Local, super-local (loops, EBBs), global
- Intra-, inter-procedural

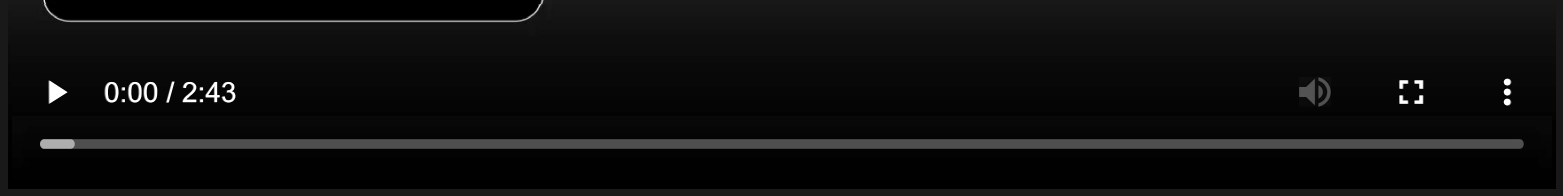
LOCAL VALUE NUMBERING

- Reasons about values, not variables
- Helps eliminating dead code
- Does common subexpression elimination
- Does copy propagation
- And more...

THE LVN ALGORITHM

Env # Val Home

```
int f(int v) {
  int a = 5;
  a = v;
  int x = 3;
  int y = x;
  int b = y;
  int s = a + b;
  int q = b + a;
  int p = s * q;
  return p;
}
```



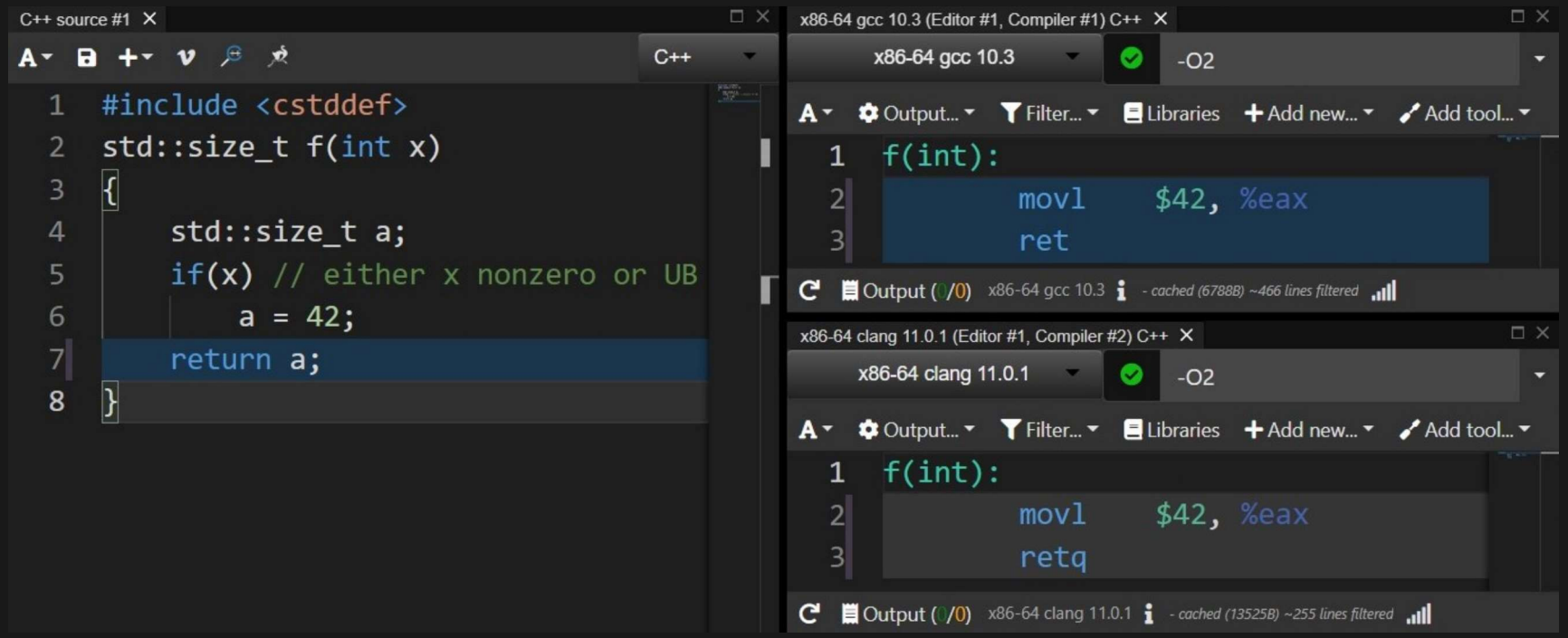
ALGEBRAIC IDENTITIES

DEAD CODE ELIMINATION

CONSTANT FOLDING

OPTIMIZATIONS AND SEMANTICS

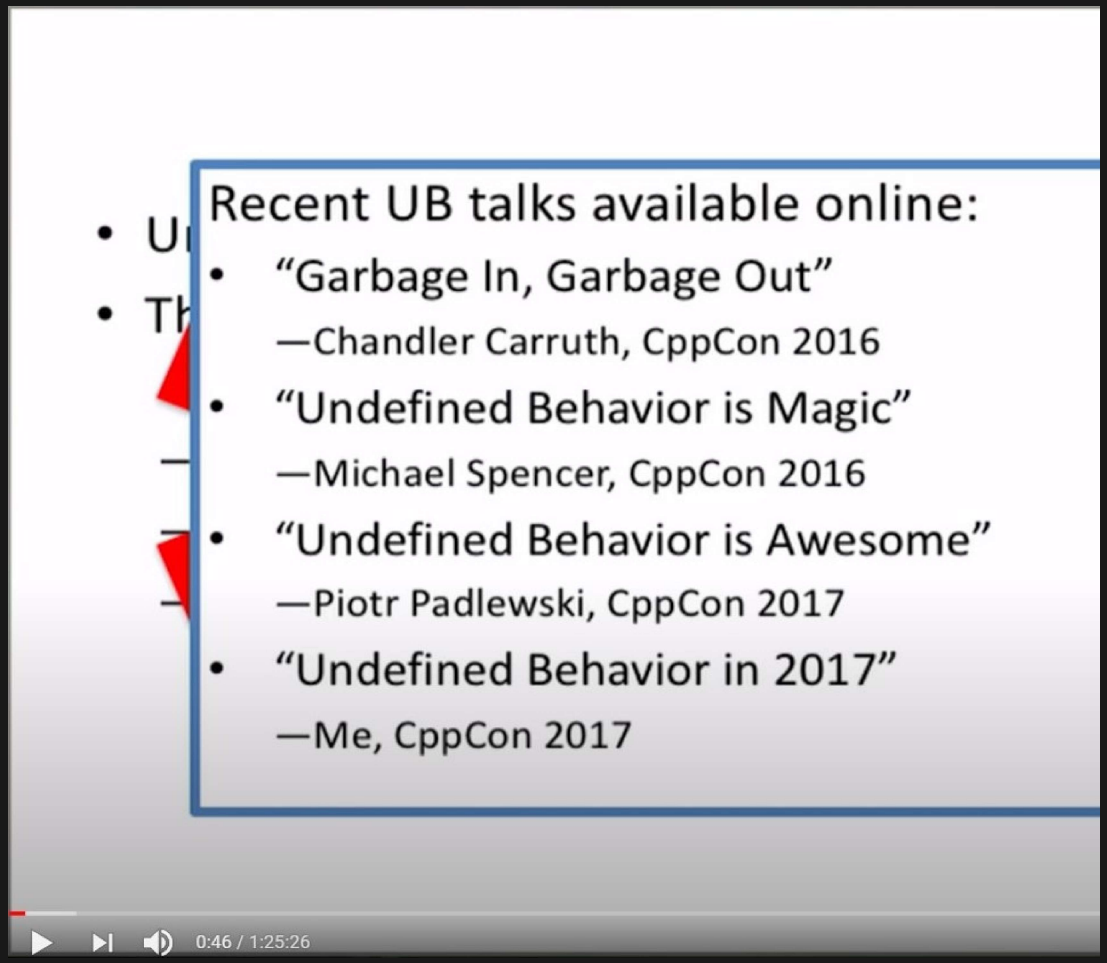
Cppreference UB examples



OPTIMIZATIONS AND SEMANTICS

REFINE

OPTIMIZATIONS AND SEMANTICS



LVN DEMO

```
> cat tests/lvn_dce.bril
# CMD: bril2json < {filename} | python3 ../lvn.py | python3 ../dce.py | bril2txt
@main {
  a: int = const 6;
  a: int = const 4;
  b: int = const 2;
  sum1: int = add a b;
  sum2: int = add b a;
  prod: int = mul sum1 sum2;
  print prod;
  x: int = const 4;
  copy1: int = id x;
  copy2: int = id copy1;
  copy3: int = id copy2;
  print copy3;
  c: int = const 5;
  sum1: int = add c b;
  prod2: int = mul sum1 sum2;
}
```

LVN DEMO

```
> cat tests/lvn_dce.bril | bril2json | python3 lvn.py | python3 dce.py | bril2txt
@main {
  a#1: int = const 4;
  prod: int = const 36;
  print prod;
  print a#1;
}
```

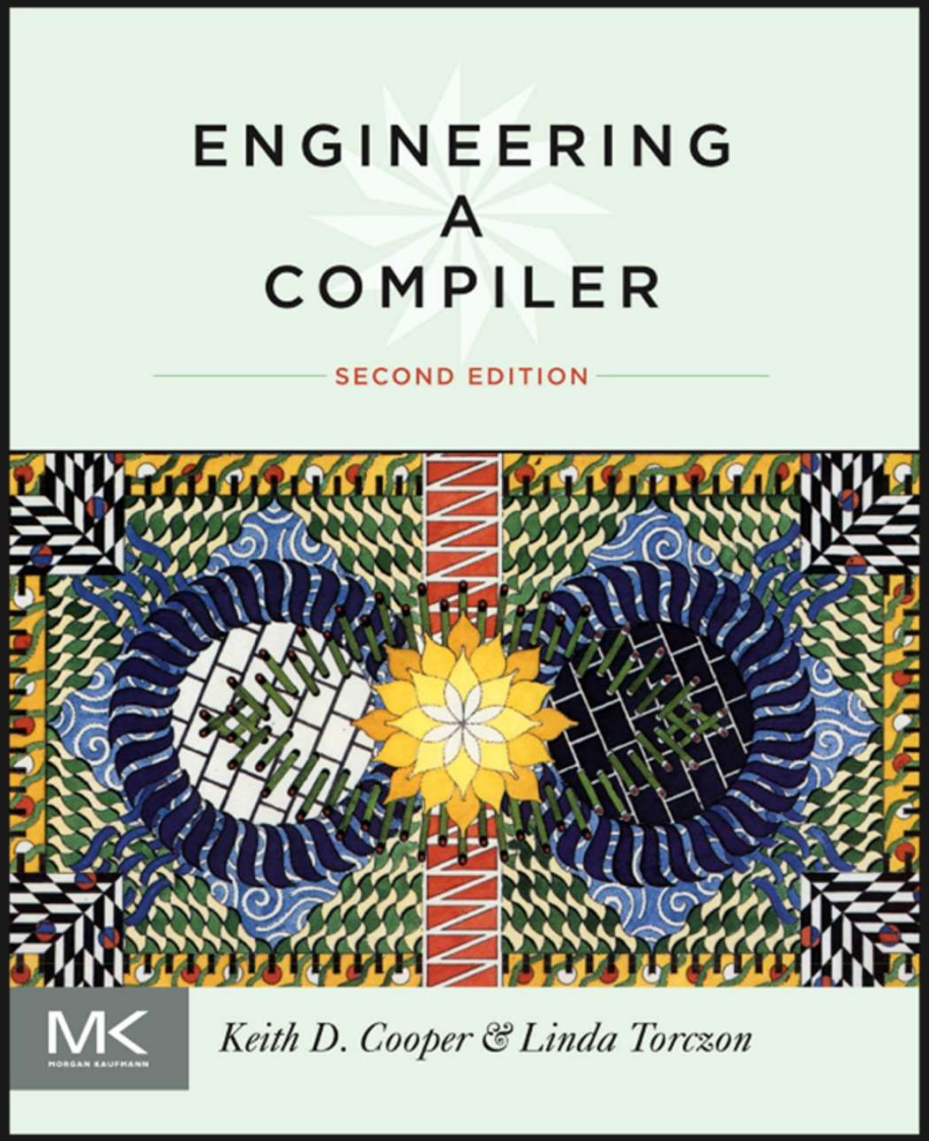
LVN DEMO

```
> cloc lvn.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.82  T=0.01 s (172.3 files/s, 36009.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1             39             22            148
-------------------------------------------------------------------------------
```

LVN DEMO

```
> cloc dce.py
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.82  T=0.00 s (233.1 files/s, 17948.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1             14             15             48
-------------------------------------------------------------------------------
```

WHERE TO LEARN MORE



Cornell CS 6120, self-guided online course, Lesson 3

LLVM's Global Value Numbering pass

Introducing a new, advanced Visual C++ code optimizer

RECAPPING VALUE NUMBERING

RECAPPING VALUE NUMBERING

- Redundancy everywhere

RECAPPING VALUE NUMBERING

- Redundancy everywhere
- VN is effective

RECAPPING VALUE NUMBERING

- Redundancy everywhere
- VN is effective
- Algebraic identities are important

RECAPPING VALUE NUMBERING

- Redundancy everywhere
- VN is effective
- Algebraic identities are important
- Don't have to be a rock star to implement it

CONCLUSIONS

CONCLUSIONS

- Compilers are full of marvels

CONCLUSIONS

- Compilers are full of marvels
- It is not black magic, it can be understood

CONCLUSIONS

- Compilers are full of marvels
- It is not black magic, it can be understood
- You can easily start to experiment

CONCLUSIONS

- Compilers are full of marvels
- It is not black magic, it can be understood
- You can easily start to experiment
- Simple albeit beautiful math can go a long way

CONCLUSIONS

- Compilers are full of marvels
- It is not black magic, it can be understood
- You can easily start to experiment
- Simple albeit beautiful math can go a long way
- You do not need to help the compiler (mostly)

Q&A