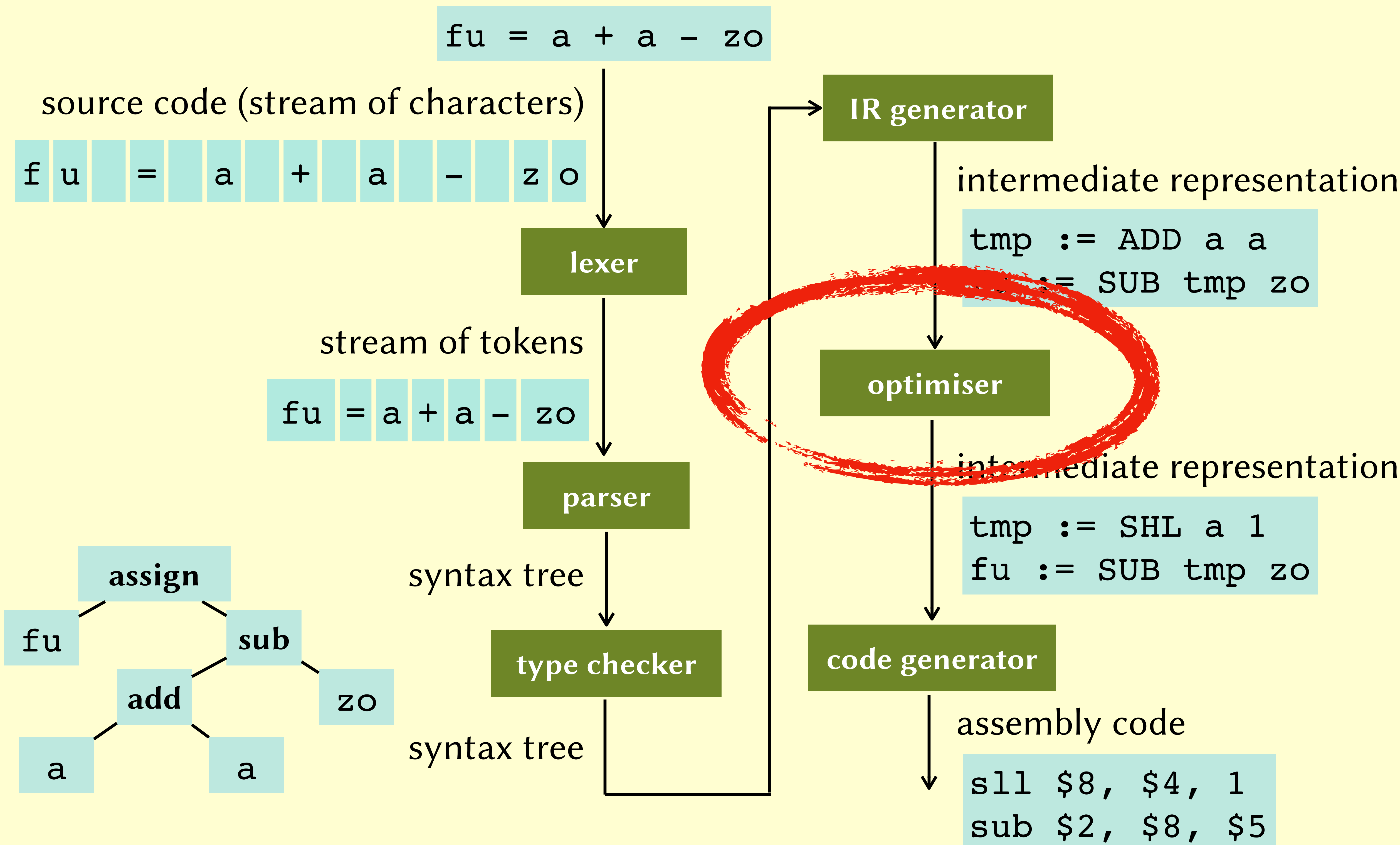
Lecture 12: Data flow analysis

John Wickerson

Compilers

Anatomy of a compiler



Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;

  f = d + a;

  g = a + c;

  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;

  e = d + b;

  f = d + a;

  g = a + c;

  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;

  f = d + a;

  g = a + c;

  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;

  g = a + c;

  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;

  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;

  f = d + a;

  g = a + c;

  h = e + g;

  return f + h;

}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;

  f = d + a;

  g = a + c;

  h = e + g;

  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;

  f = d + a;

  g = a + c;

  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;

  f = d + a;

  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;

  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;

  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;

  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

Live-variable analysis

```
int main (int a, int b, int c) {
  int d, e, f, g, h;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c, d }
  e = d + b;
  //{ a, c, d, e }
  f = d + a;
  //{ a, c, e, f }
  g = a + c;
  //{ e, f, g }
  h = e + g;
  //{ f, h }
  return f + h;
  //∅
}
```

$$\text{live}_{\text{before}}(s) = \text{live}_{\text{after}}(s) - \text{def}_V(s) \cup \text{use}_V(s)$$

$\text{def}_V($**`return`** `f + h`$) = \emptyset$

$\text{def}_V($`h = e + g`$) = \{$ `h` $\}$

$\text{use}_V($**`return`** `f + h`$) = \{$ `f`, `h` $\}$

$\text{use}_V($`h = e + g`$) = \{$ `e`, `g` $\}$

Live-variable analysis

- Useful for **register allocation**.

- Also useful for **dead code elimination**.

```
int main (int a, int b, int c) {
  int d, e;

  d = a + b;

  e = b * c;

  return a + e;

}
```

Live-variable analysis

- Useful for **register allocation**.

- Also useful for **dead code elimination**.

```
int main (int a, int b, int c) {
  int d, e;

  d = a + b;

  e = b * c;

  return a + e;
  //∅
}
```

Live-variable analysis

- Useful for **register allocation**.
- Also useful for **dead code elimination**.

```
int main (int a, int b, int c) {
  int d, e;

  d = a + b;

  e = b * c;
  //{a, e}
  return a + e;
  //∅
}
```

Live-variable analysis

- Useful for **register allocation**.
- Also useful for **dead code elimination**.

```
int main (int a, int b, int c) {
  int d, e;

  d = a + b;
  //{ a, b, c }
  e = b * c;
  //{ a, e }
  return a + e;
  //∅
}
```

Live-variable analysis

- Useful for **register allocation**.

- Also useful for **dead code elimination**.

```
int main (int a, int b, int c) {
  int d, e;
  //{ a, b, c }
  d = a + b;
  //{ a, b, c }
  e = b * c;
  //{ a, e }
  return a + e;
  //∅
}
```

Analysis	Operates on	Transformation		
live variables	sets of variables	register allocation, dead code elimination		

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;

  d = a + b;

  e = c * b;

  f = a + b;

  a = f / 10;

  return a + b;
}
```

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;
  //∅
  d = a + b;

  e = c * b;

  f = a + b;

  a = f / 10;

  return a + b;
}
```

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;
  //∅
  d = a + b;
  //{ a+b }
  e = c * b;

  f = a + b;

  a = f / 10;

  return a + b;
}
```

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;
  //∅
  d = a + b;
  //{ a+b }
  e = c * b;
  //{ a+b, c*b }
  f = a + b;

  a = f / 10;

  return a + b;
}
```

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;
  //∅
  d = a + b;
  //{ a+b }
  e = c * b;
  //{ a+b, c*b }
  f = a + b;
  //{ a+b, c*b }
  a = f / 10;

  return a + b;
}
```

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;
  //∅
  d = a + b;
  //{ a+b }
  e = c * b;
  //{ a+b, c*b }
  f = a + b;
  //{ a+b, c*b }
  a = f / 10;
  //{ c*b, f/10 }
  return a + b;
}
```

$\text{avail}_{\text{after}}(s) = \text{avail}_{\text{before}}(s) \cup \text{gen}_{\text{AE}}(s) - \text{kill}_{\text{AE}}(s)$

$\text{gen}_{\text{AE}}(\texttt{return a + b}) = \{\, \texttt{a+b} \,\}$

$\text{gen}_{\text{AE}}(\texttt{a = f / 10}) = \{\, \texttt{f/10} \,\}$

$\text{kill}_{\text{AE}}(\texttt{return a + b}) = \emptyset$

$\text{kill}_{\text{AE}}(\texttt{a = f / 10}) = E_{\texttt{a}}$

Available expressions

```
int main (int a, int b, int c) {
  int d, e, f;
  //∅
  d = a + b;
  //{ a+b }
  e = c * b;
  //{ a+b, c*b }
  f = d;
  //{ a+b, c*b }
  a = f / 10;
  //{ c*b, f/10 }
  return a + b;
}
```

$avail_{after}(s) = avail_{before}(s) \cup gen_{AE}(s) - kill_{AE}(s)$

$gen_{AE}(\texttt{return a + b}) = \{ \texttt{a+b} \}$

$gen_{AE}(\texttt{a = f / 10}) = \{ \texttt{f/10} \}$

$kill_{AE}(\texttt{return a + b}) = \emptyset$

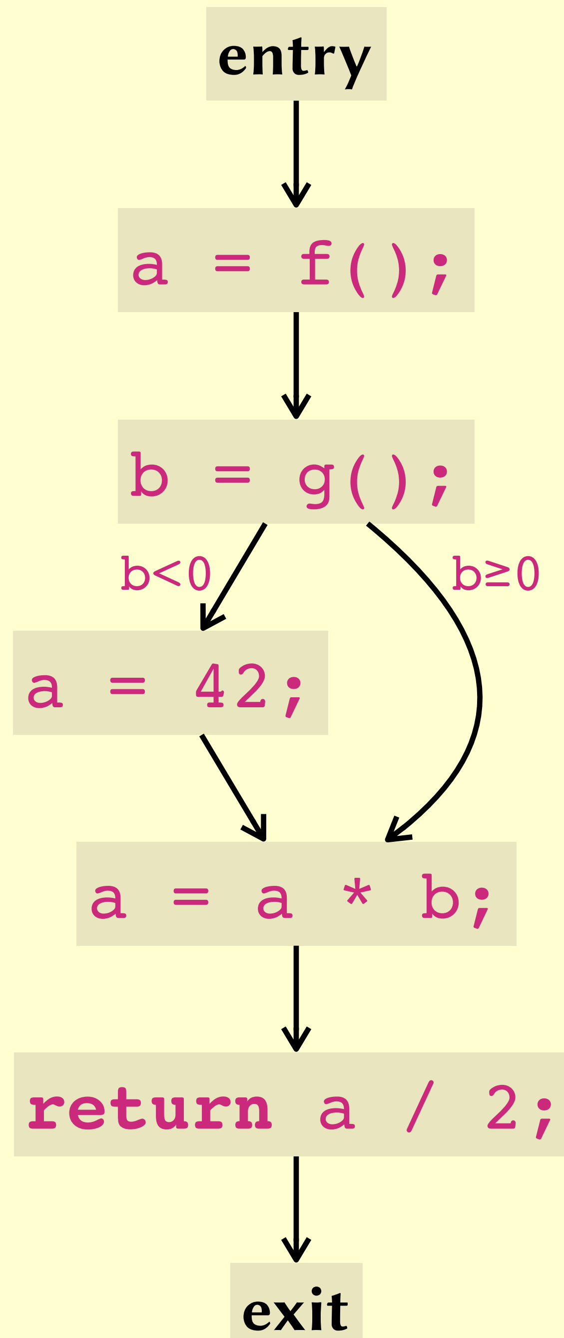
$kill_{AE}(\texttt{a = f / 10}) = E_{\texttt{a}}$

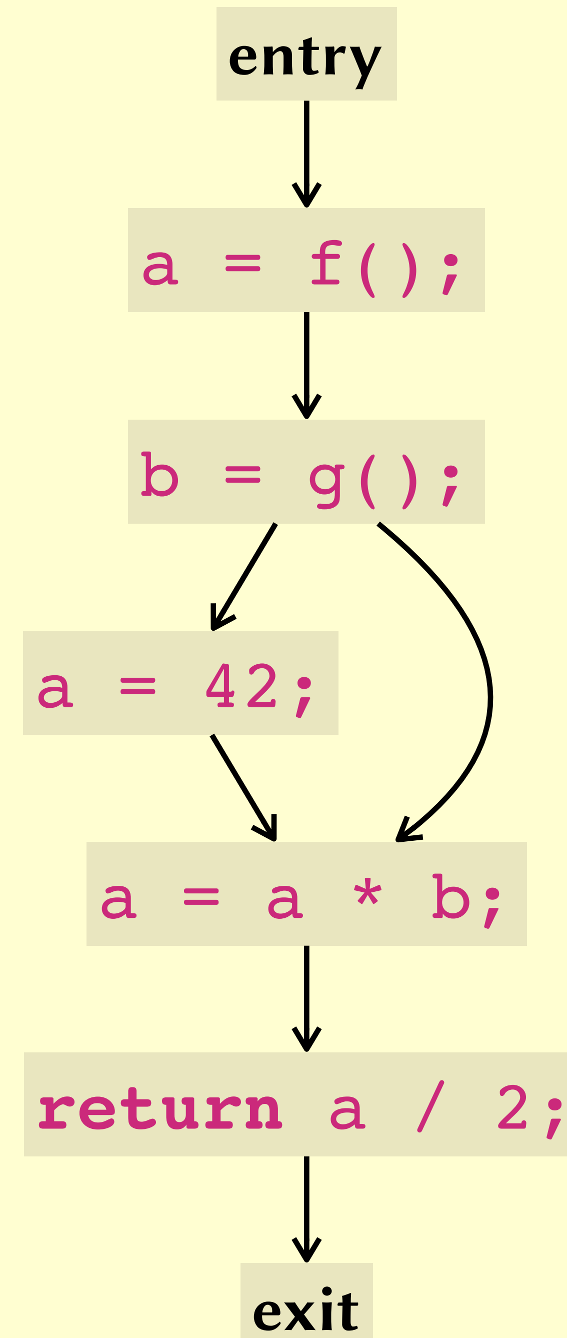
"common subexpression elimination"

Analysis	**Operates on**	**Transformation**	**Direction**	
live variables	sets of variables	register allocation, dead code elimination	Backward	
available expressions	sets of expressions	common subexpression elimination	Forward	

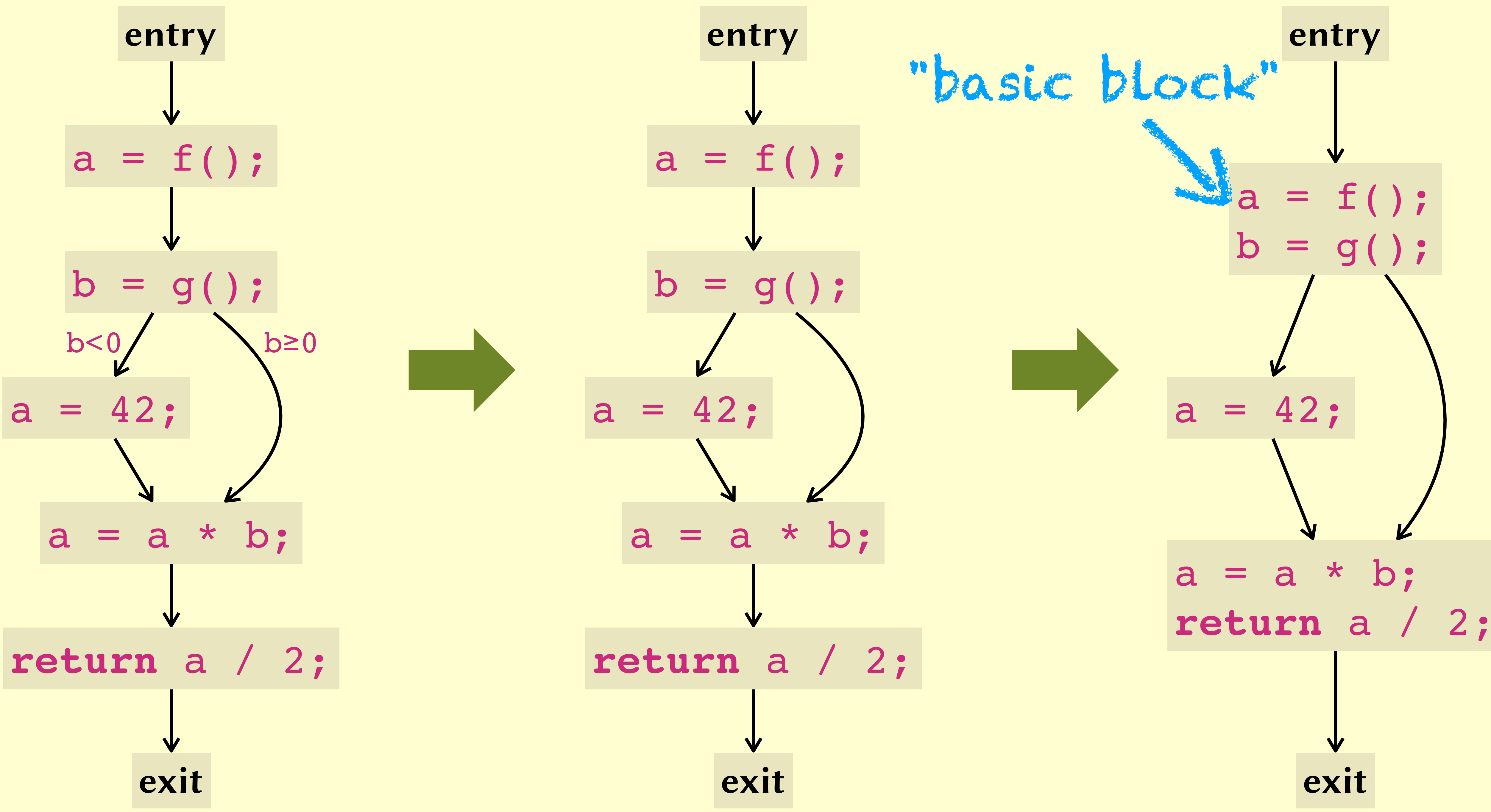
Control-flow graphs

```
a = f();
b = g();
if (b < 0) {
  a = 42;
}
a = a * b;
return a / 2;
```

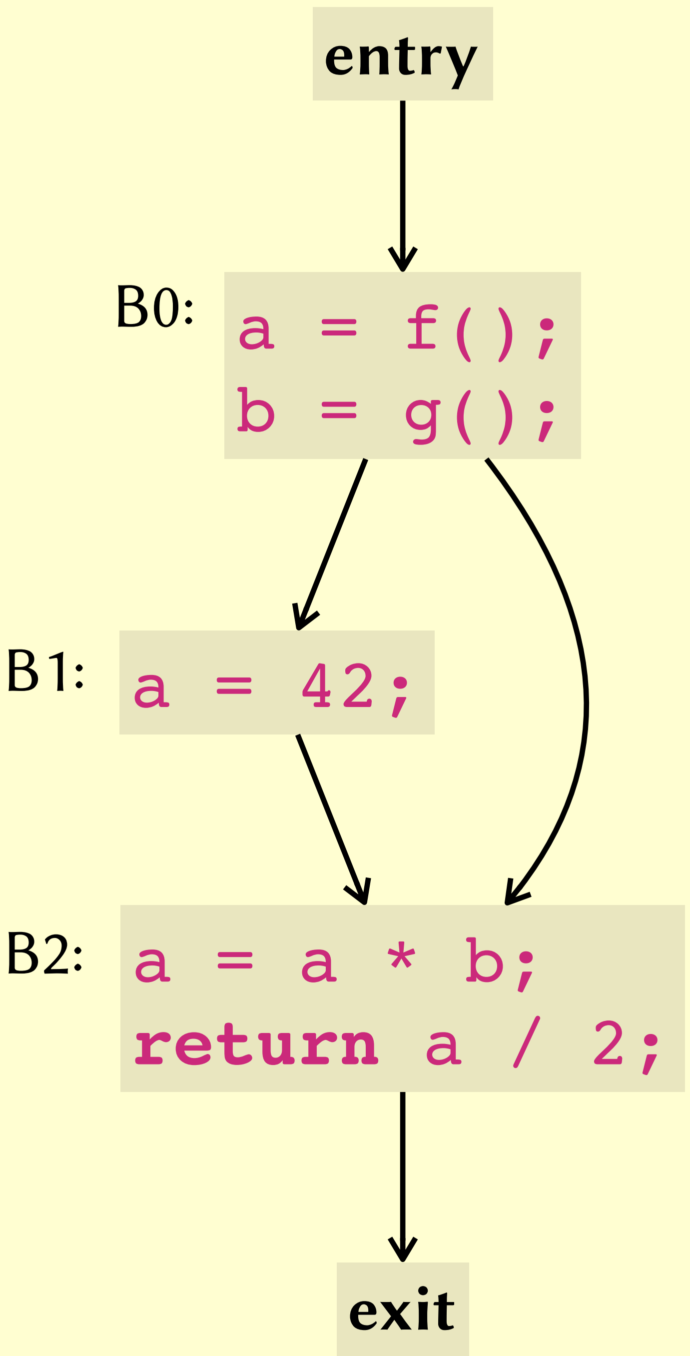




Control-flow graphs



Terminology



succ(B) = all blocks that B can jump to

pred(B) = all blocks that can jump to B

analyseBlock(B, X) = analyse block B, starting from X

LV on a CFG

in:
entry
out:

in:
B0: `a = f();`
`b = g();`
out: `a` `b`

in: `b`
B1: `a = 42;`
out: `a` `b`

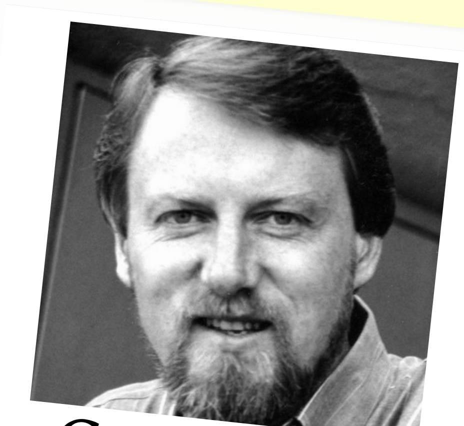
in: `a` `b`
B2: `a = a * b;`
`return a / 2;`
out:

in:
exit
out:

$$out[B] = in[B_1] \cup \ldots \cup in[B_n]$$
$$\text{where } succ(B) = \{B_1,\ldots,B_n\}$$

$$in[B] = analyseBlock_{LV}(B, out[B])$$

Start with $in[B] = out[B] = \emptyset$, for all B. Then keep applying these definitions until nothing changes.



Gary Kildall
1942–1994

Analysis precision

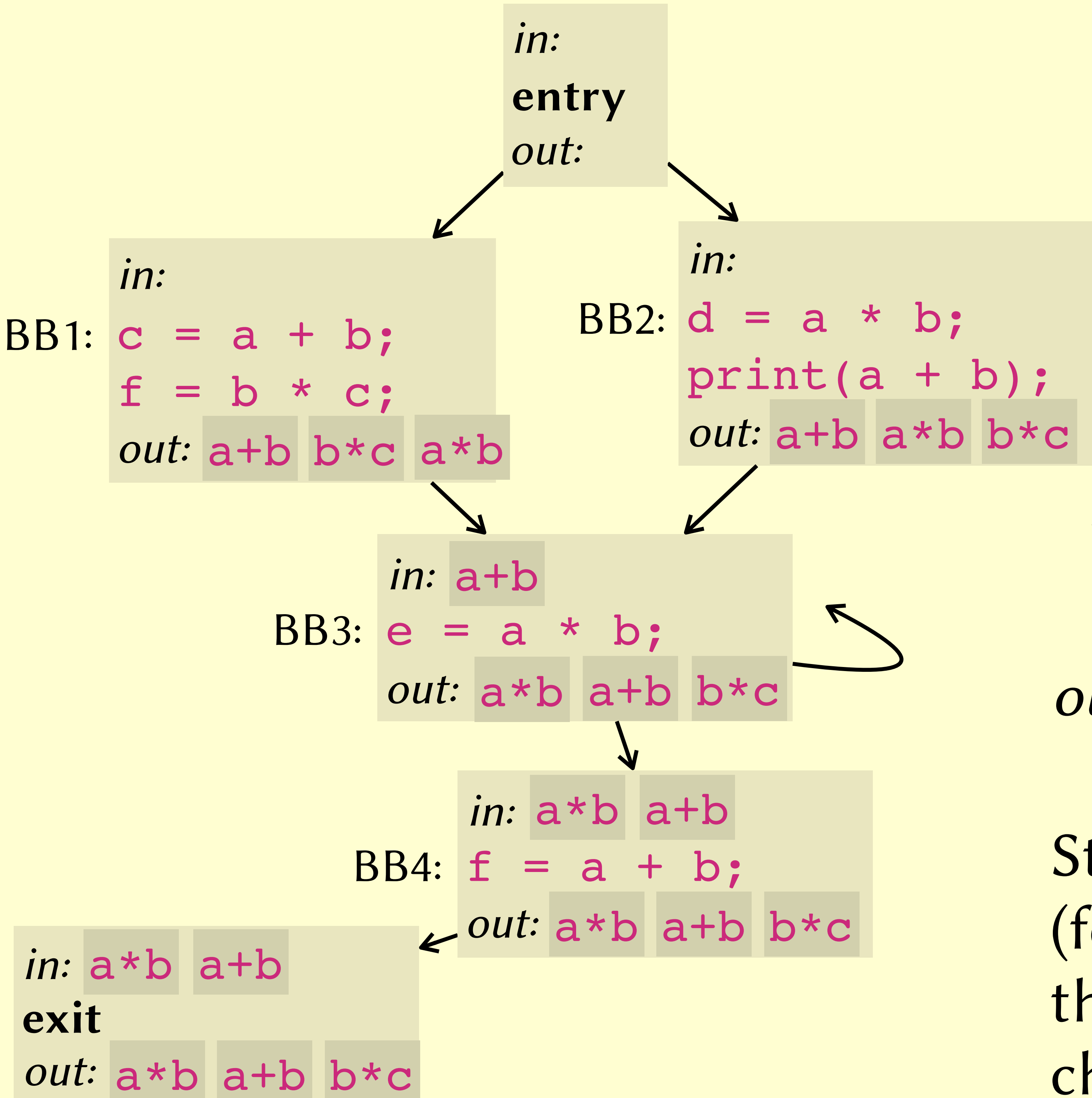
```
int main(int a,
         int b) {
  int foo[2];
  foo[a] = 42;
  return foo[b];
}
```

```
int main() {
  int a = 42;
  if (1) return 0;
  else return a;
}
```

```
int main() {
  int a = 42;
  while(1);
  return a;
}
```

- What happens if the analysis wrongly says a variable **is not live**?
- What happens if the analysis wrongly says that a variable **is live**?
- Live-variable analysis **overapproximates** the set of live variables.

AE on a CFG

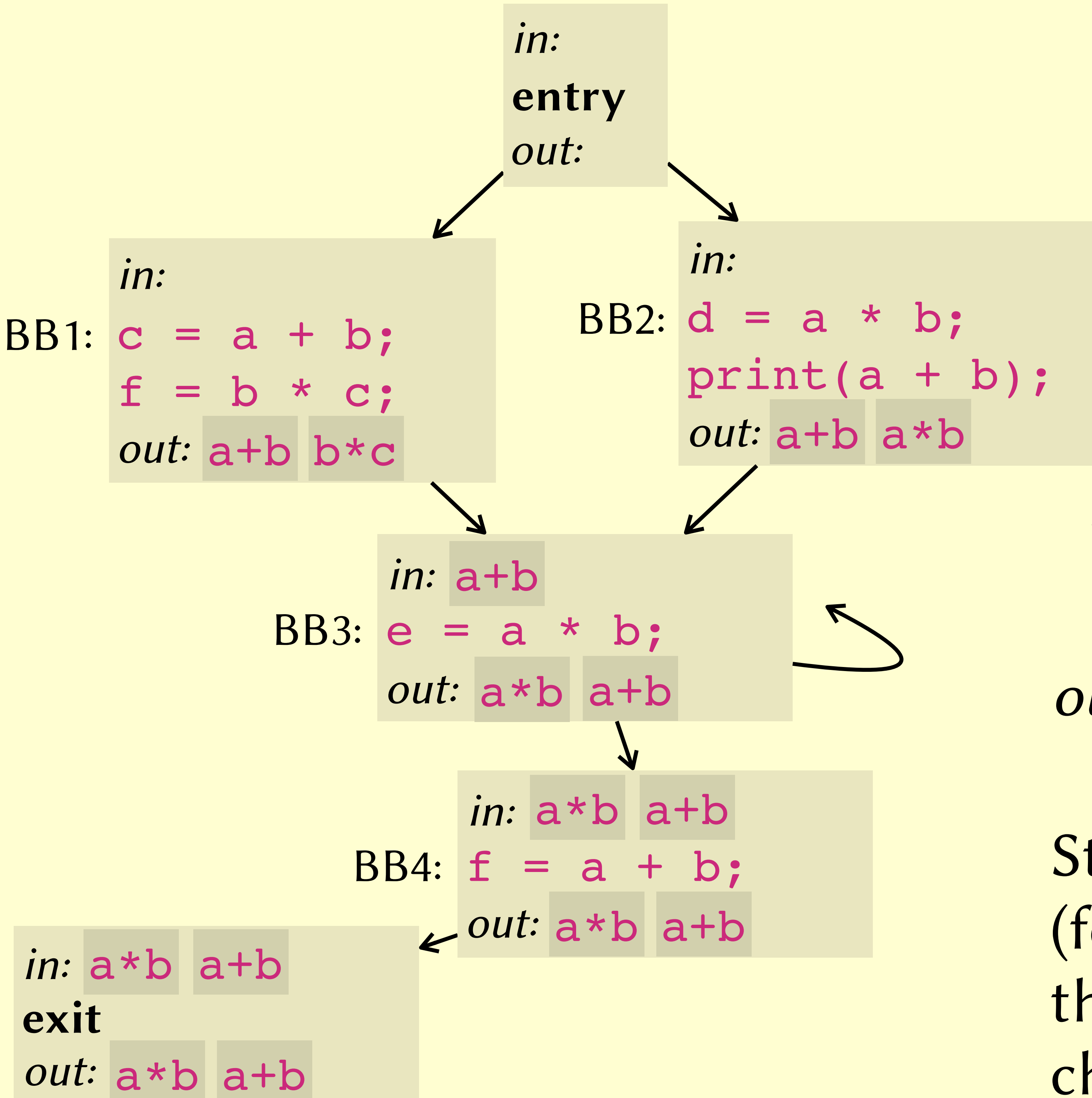


$$\mathit{in}[\mathrm{B}] = \mathit{out}[\mathrm{B}_1] \cap \ldots \cap \mathit{out}[\mathrm{B}_n]$$
$$\text{where } \mathrm{pred}(\mathrm{B}) = \{\mathrm{B}_1,\ldots,\mathrm{B}_n\}$$

$$\mathit{out}[\mathrm{B}] = \mathrm{analyseBlock}_{\mathrm{AE}}(\mathit{in}[\mathrm{B}])$$

Start with *out*[B] = *all expressions* (for all B). Then keep applying these definitions until nothing changes.

AE on a CFG



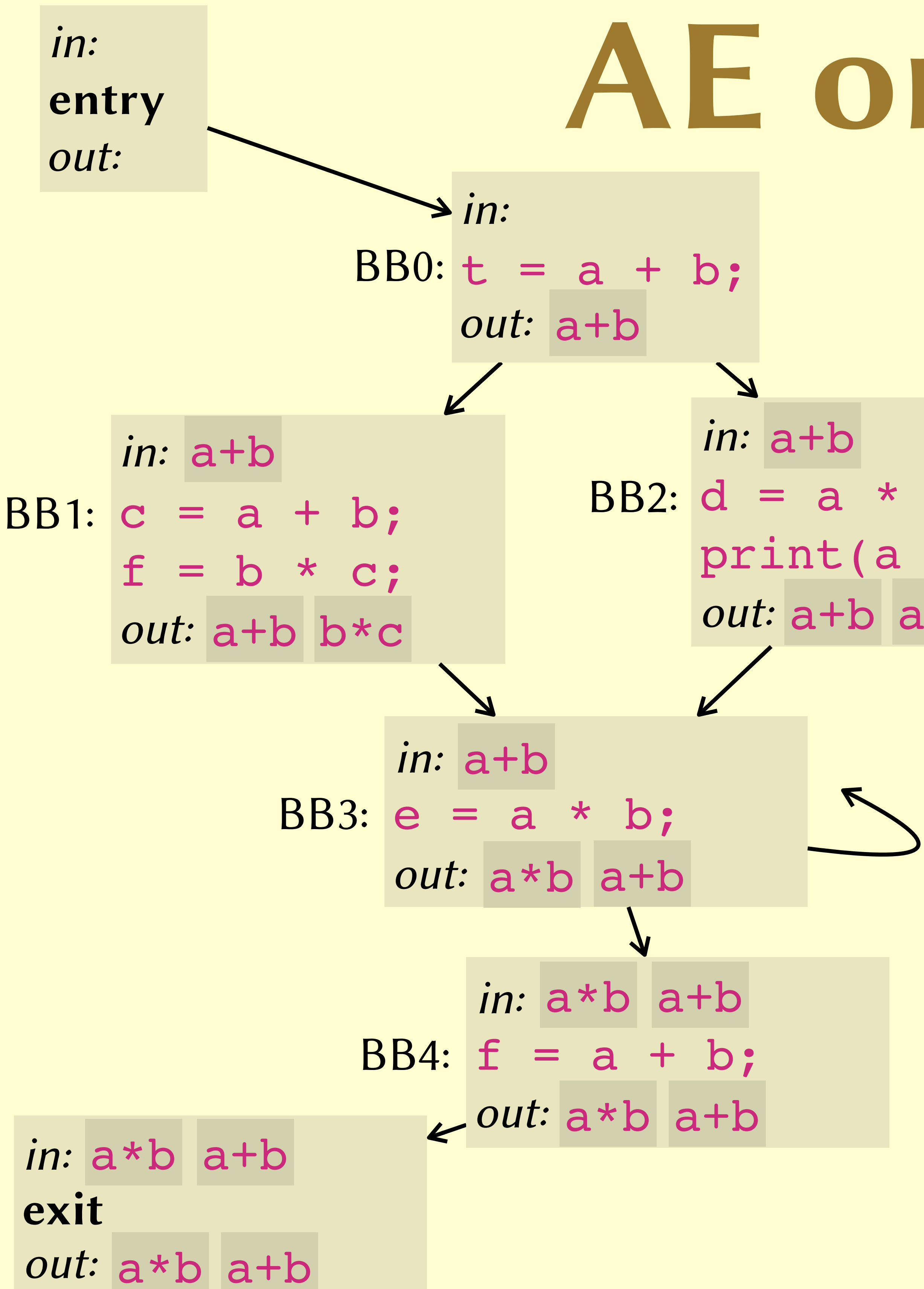
$$in[B] = out[B_1] \cap \ldots \cap out[B_n]$$

$$\text{where } \mathrm{pred}(B) = \{B_1,\ldots,B_n\}$$

$$out[B] = \mathrm{analyseBlock}_{AE}(in[B])$$

Start with *out*[B] = *all expressions* (for all B). Then keep applying these definitions until nothing changes.

AE on a CFG

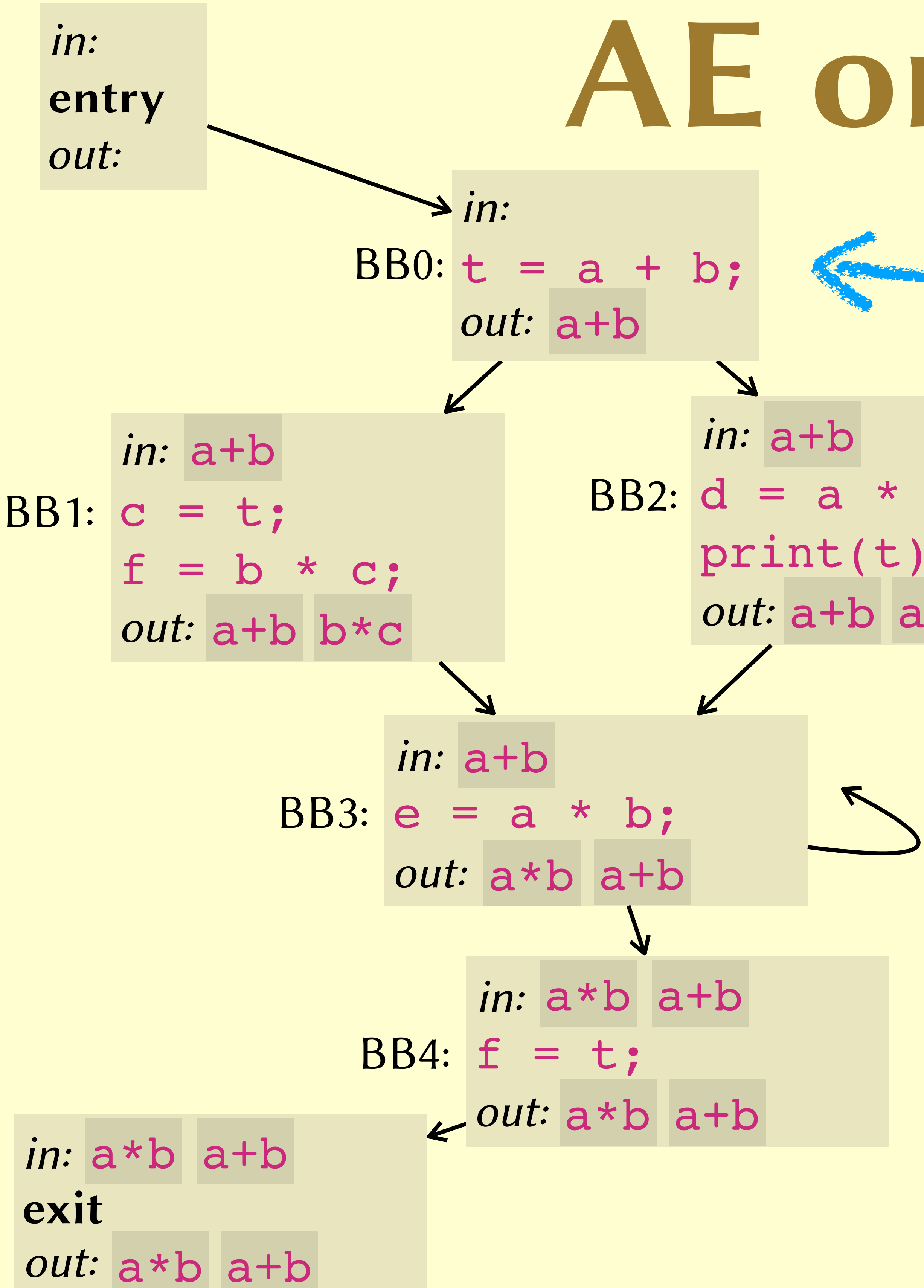


in[B] = *out*[B$_1$] ∩ … ∩ *out*[B$_n$]
where pred(B) = {B$_1$,…,B$_n$}

out[B] = analyseBlock$_{AE}$(*in*[B])

Start with *out*[B] = *all expressions* (for all B). Then keep applying these definitions until nothing changes.

AE on a CFG



but was it worth it?

$$in[B] = out[B_1] \cap \ldots \cap out[B_n]$$
where $\text{pred}(B) = \{B_1,\ldots,B_n\}$

$$out[B] = \text{analyseBlock}_{AE}(in[B])$$

Start with *out*[B] = *all expressions* (for all B). Then keep applying these definitions until nothing changes.

Analysis precision

- What happens if the analysis wrongly says an expression **is not available**?
- What happens if the analysis wrongly says that an expression **is available**?
- Available-expression analysis **underapproximates** the set of available expressions.

Analysis	Operates on	Transformation	Direction	Meet
live variables	sets of variables	register allocation, dead code elimination	Backward	$\cup$
available expressions	sets of expressions	common subexpression elimination	Forward	$\cap$

Reaching definitions

```
int main () {
  int a = f();
  int b = g();
  if (b > 0) a = 42;
  a = a * b;
  return a / 2;
}
```

Reaching definitions

```
int main () {
  //∅
  int a = f();

  int b = g();

  if (b > 0) a = 42;

  a = a * b;

  return a / 2;
}
```

Reaching definitions

```
23 int main () {
     // ∅
24   int a = f();
     // { 24: a }
25   int b = g();

26   if (b > 0) a = 42;

27   a = a * b;

28   return a / 2;
   }
```

Reaching definitions

```
23 int main () {
     // ∅
24   int a = f();
     // { 24: a }
25   int b = g();
     // { 24: a, 25: b }
26   if (b > 0) a = 42;

27   a = a * b;

28   return a / 2;
   }
```

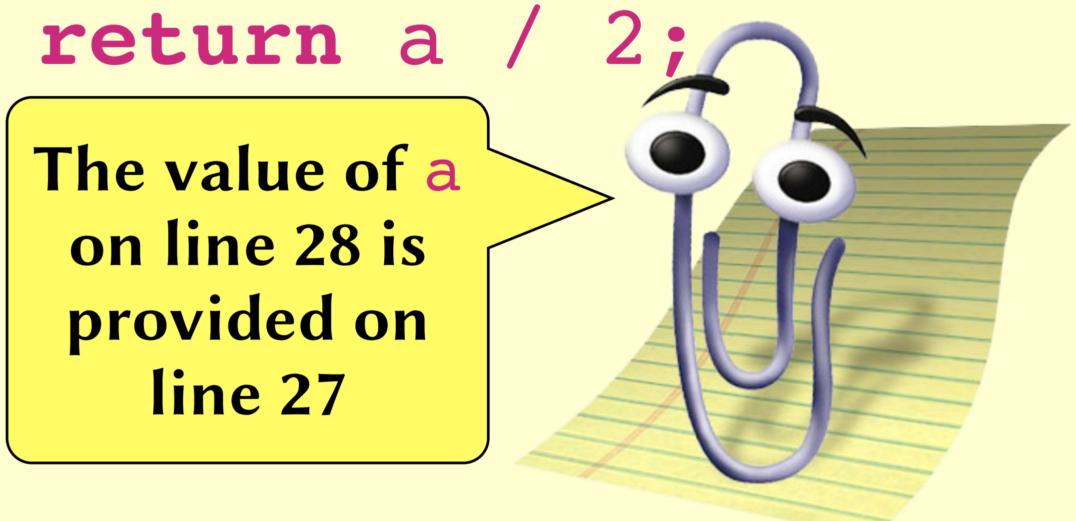
Reaching definitions

```
23 int main () {
     // ∅
24   int a = f();
     // { 24: a }
25   int b = g();
     // { 24: a, 25: b }
26   if (b > 0) a = 42;
     // { 24: a, 25: b, 26: a }
27   a = a * b;

28   return a / 2;
   }
```

Reaching definitions

```
23 int main () {
     // ∅
24   int a = f();
     // { 24: a }
25   int b = g();
     // { 24: a, 25: b }
26   if (b > 0) a = 42;
     // { 24: a, 25: b, 26: a }
27   a = a * b;
     // { 25: b, 27: a }
28   return a / 2;
   }
```




```
int main () {
  int a = 7;
  int b = f();
  return a - b;
}
```

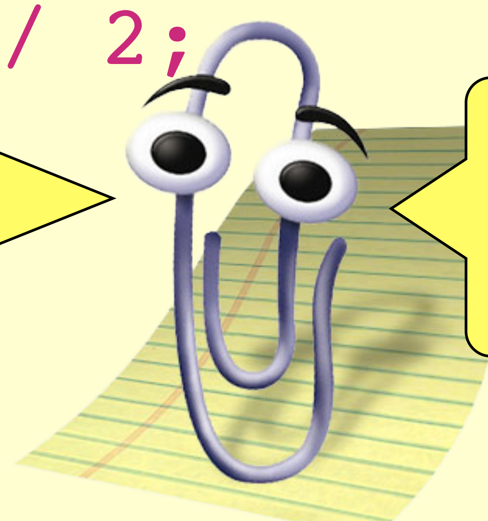
Reaching definitions

```
23 int main () {
     // ∅
24   int a = f();
     // { 24: a }
25   int b = g();
     // { 24: a, 25: b }
26   if (b > 0) a = 42;
     // { 24: a, 25: b, 26: a }
27   a = a * b;
     // { 25: b, 27: a }
28   return a / 2;
   }
```

```
int main () {
  int a = 7;
  int b = f();
  return 7 - b;
}
```

"constant propagation"

The value of a on line 28 is provided on line 27



Oops, you forgot to initialise d!

```
int main () {
  int a = f();
  return a - d;
}
```

Reaching definitions

```
23 int main () {
     // ∅
24   int a = f();
     // { 24: a }
25   int b = g();
     // { 24: a, 25: b }
26   if (b > 0) a = 42;
     // { 24: a, 25: b, 26: a }
27   a = a * b;
     // { 25: b, 27: a }
28   return a / 2;
   }
```

$$\text{reach}_{\text{after}}(s) = \text{reach}_{\text{before}}(s) - \text{kill}_D(s) \cup \text{gen}_D(s)$$

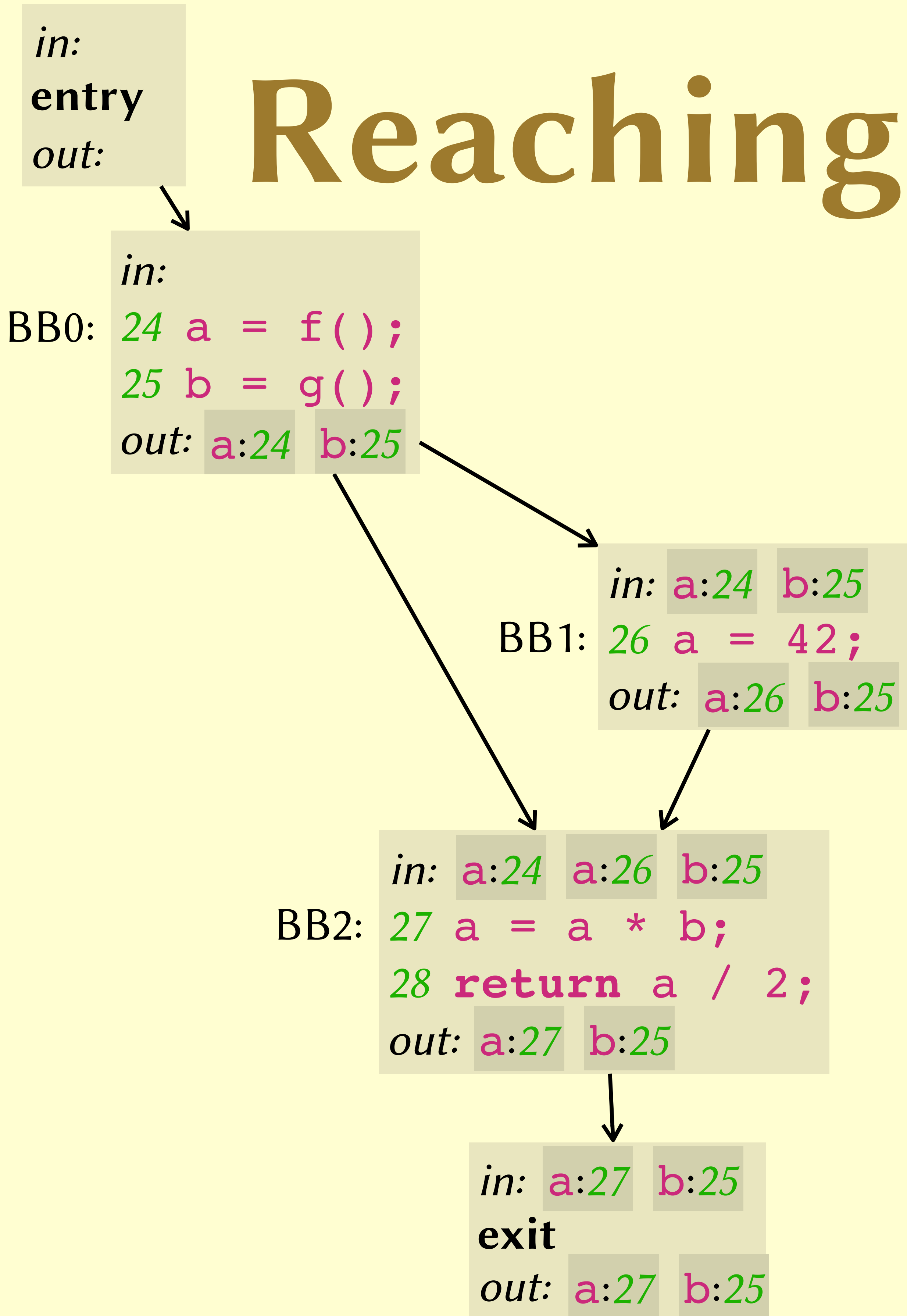
$$\text{kill}_D(L\text{: }\mathtt{return\ a\ /\ 2}) = \emptyset$$

$$\text{kill}_D(L\text{: }\mathtt{a\ =\ 42}) = D_{\mathtt{a}}$$

$$\text{gen}_D(L\text{: }\mathtt{return\ a\ /\ 2}) = \emptyset$$

$$\text{gen}_D(L\text{: }\mathtt{a\ =\ 42}) = \{\ L\text{: }\mathtt{a}\ \}$$

Reaching definitions



$$in[\mathrm{B}] = out[\mathrm{B}_1] \cup \ldots \cup out[\mathrm{B}_n]$$
$$\text{where } \mathrm{pred}(\mathrm{B}) = \{\mathrm{B}_1,\ldots,\mathrm{B}_n\}$$

$$out[\mathrm{B}] = \mathrm{analyseBlock}_{\mathrm{RD}}(in[\mathrm{B}])$$

Start with $out[\mathrm{B}] = in[\mathrm{B}] = \varnothing$, for all B. Then keep applying these definitions until nothing changes.

Dataflow analyses

Frances Allen
1932-2020

Analysis	Operates on	Transformation	Direction	Meet
live variables	sets of variables	register allocation, dead code elimination	Backward	$\cup$
available expressions	sets of expressions	common subexpression elimination	Forward	$\cap$
reaching definitions	sets of definitions	constant propagation, spotting undefined variables	Forward	$\cup$

Dataflow analyses

Analysis	Operates on	Transformation	Direction	Meet
live variables	sets of variables	register allocation, dead code elimination	Backward	$\cup$
available expressions	sets of expressions	common subexpression elimination	Forward	$\cap$
reaching definitions	sets of definitions	constant propagation, spotting undefined variables	Forward	$\cup$
?	?	?	Backward	$\cap$

Very busy expressions

```
int main (int a, int b, int c) {

  if (b > 0) {
    print(b + c);
  } else if (b < 0) {
    f(b + c);
  } else {
    a = b + c;
  }
  a = a * a;
  return a;
}
```

Very busy expressions

```
int main (int a, int b, int c) {
  int t = b + c;
  if (b > 0) {
    print(t);
  } else if (b < 0) {
    f(t);
  } else {
    a = t;
  }
  a = a * a;
  return a;
}
```

"hoisting"

Summary

- Data flow analysis approximates a program by a **flow graph** made up of **basic blocks**.
- Analyses can go **forwards** or **backwards**.
- Analyses can consider **all** ($\cap$) paths or **any** ($\cup$) paths.
- Analyses may not be completely **precise**, so need to **overapproximate** or **underapproximate** as appropriate.
- Analyses enable **transformations** (but need to consider whether each transformation is actually worth doing).