Motivation

Intermediate code in normal form permits maximum flexibility in allocating temporary variables to physical registers.

This flexibility is not extended to user variables, and sometimes more registers than necessary will be used.

Register allocation can do a better job with user variables if we first translate code into *SSA form*.

Live ranges

User variables are often reassigned and reused many times over the course of a program, so that they become live in many different places.

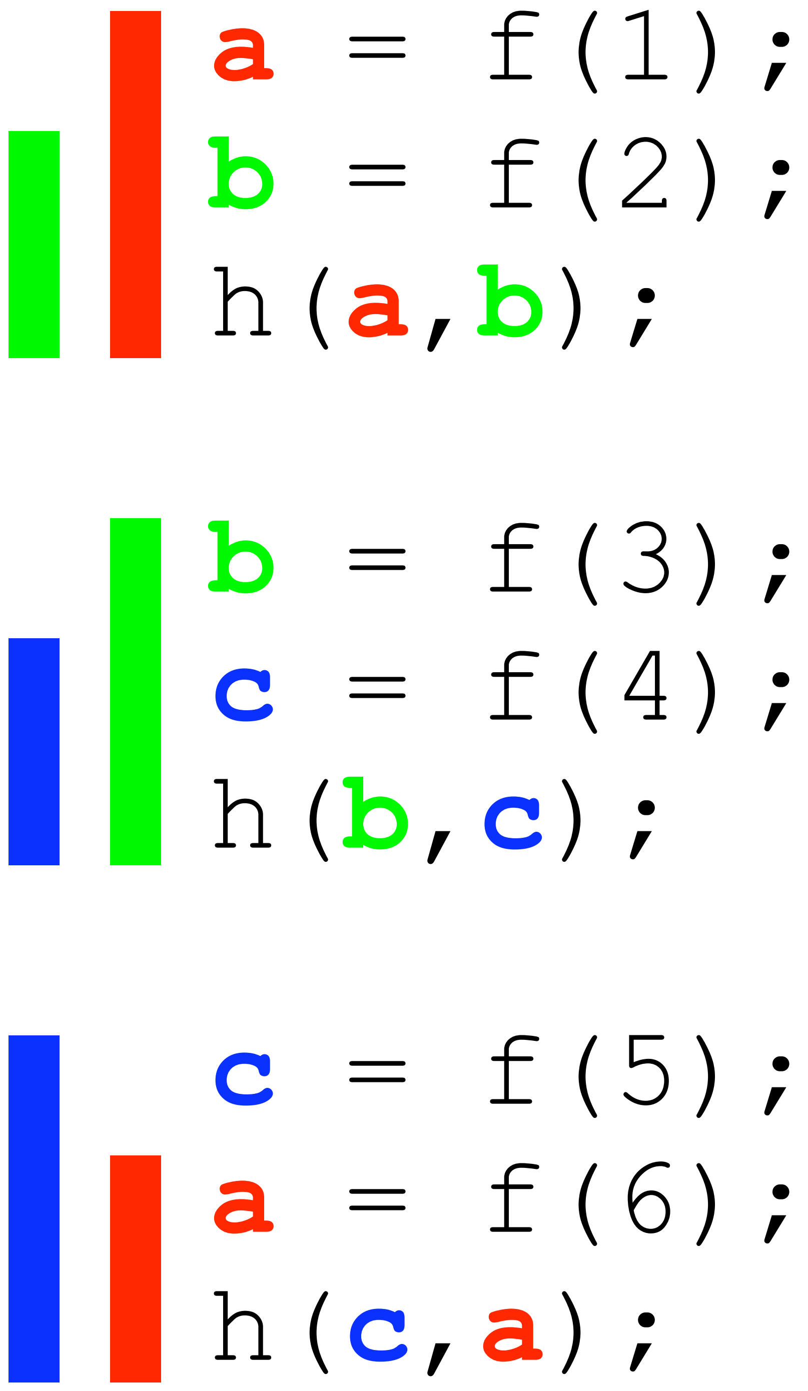
Our intermediate code generation scheme assumes that each user variable is kept in a single virtual register throughout the entire program.

This results in each virtual register having a large *live range*, which is likely to cause clashes.

Live ranges

```
extern int f(int);
extern void h(int,int);
void g()
{
  int a,b,c;
  a = f(1); b = f(2); h(a,b);
  b = f(3); c = f(4); h(b,c);
  c = f(5); a = f(6); h(c,a);
}
```

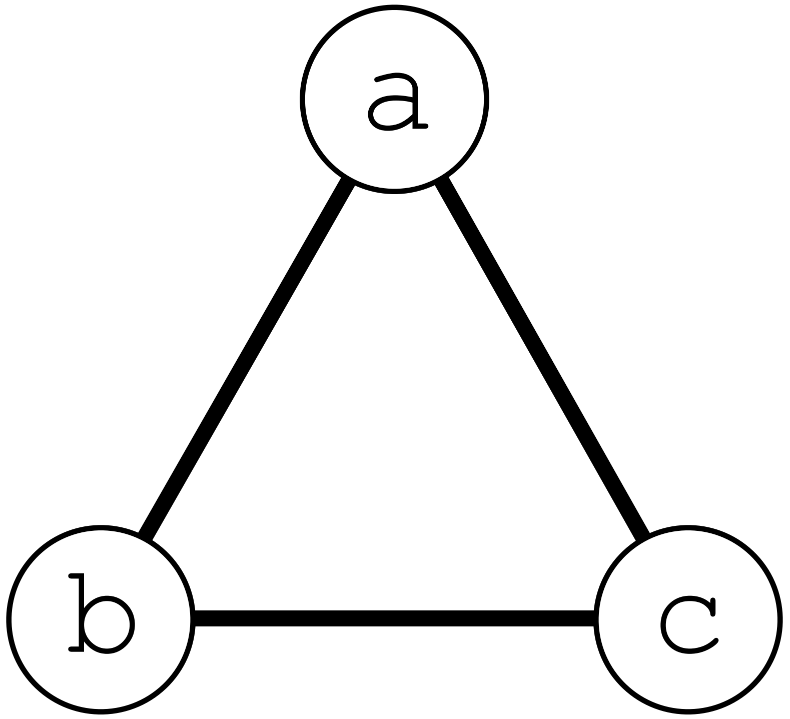
Live ranges



```
a = f(1);
b = f(2);
h(a,b);

b = f(3);
c = f(4);
h(b,c);

c = f(5);
a = f(6);
h(c,a);
```



3 registers needed

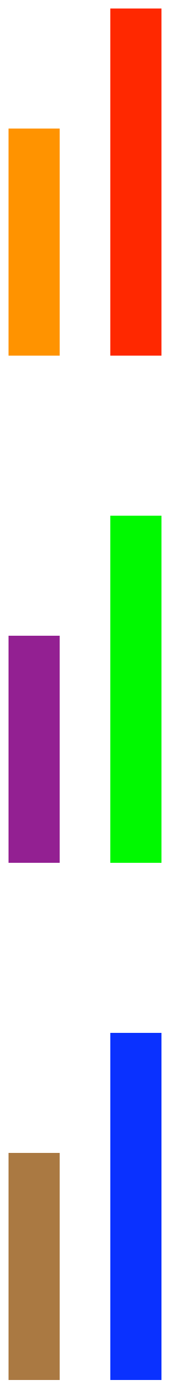
Live ranges

We may remedy this situation by performing a transformation called *live range splitting*, in which live ranges are made smaller by using a different virtual register to store a variable's value at different times, thus reducing the potential for clashes.

Live ranges

```
extern int f(int);
extern void h(int,int);
void g()
{
  int a1,a2, b1,b2, c1,c2;
  a1 = f(1); b2 = f(2); h(a1,b2);
  b1 = f(3); c2 = f(4); h(b1,c2);
  c1 = f(5); a2 = f(6); h(c1,a2);
}
```

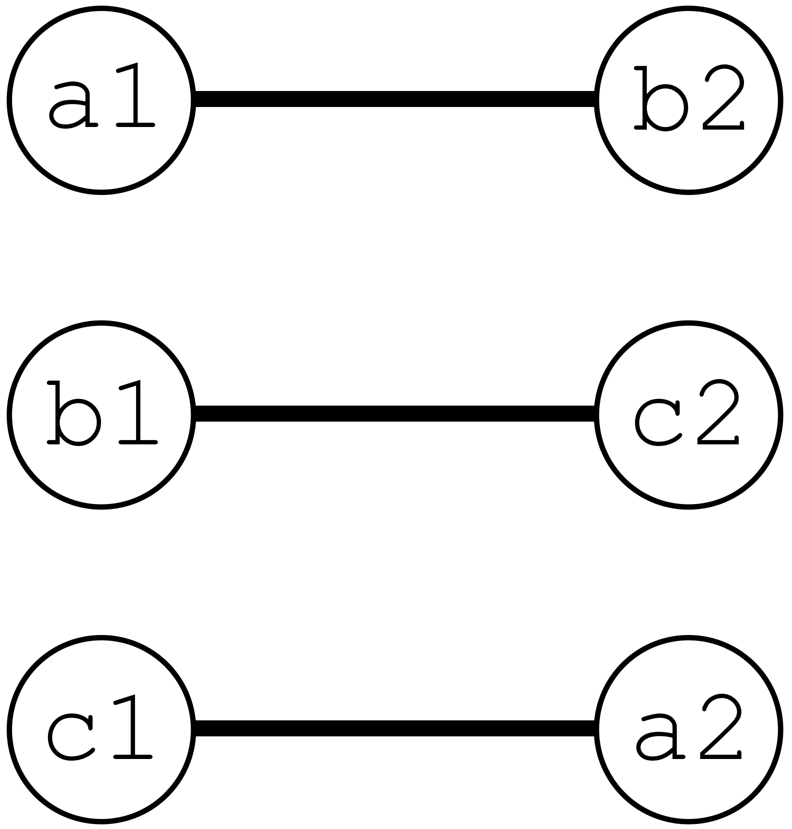
Live ranges



```
a1 = f(1);
b2 = f(2);
h(a1,b2);

b1 = f(3);
c2 = f(4);
h(b1,c2);

c1 = f(5);
a2 = f(6);
h(c1,a2);
```



2 registers needed

Static single-assignment

Live range splitting is a useful transformation: it gives the same benefits for user variables as normal form gives for temporary variables.

However, if each virtual register is only ever assigned to once (statically), we needn't perform live range splitting, since the live ranges are already as small as possible.

Code in *static single-assignment* (SSA) form has this important property.

Static single-assignment

It is straightforward to transform straight-line code into SSA form: each variable is renamed by being given a subscript, which is incremented every time that variable is assigned to.

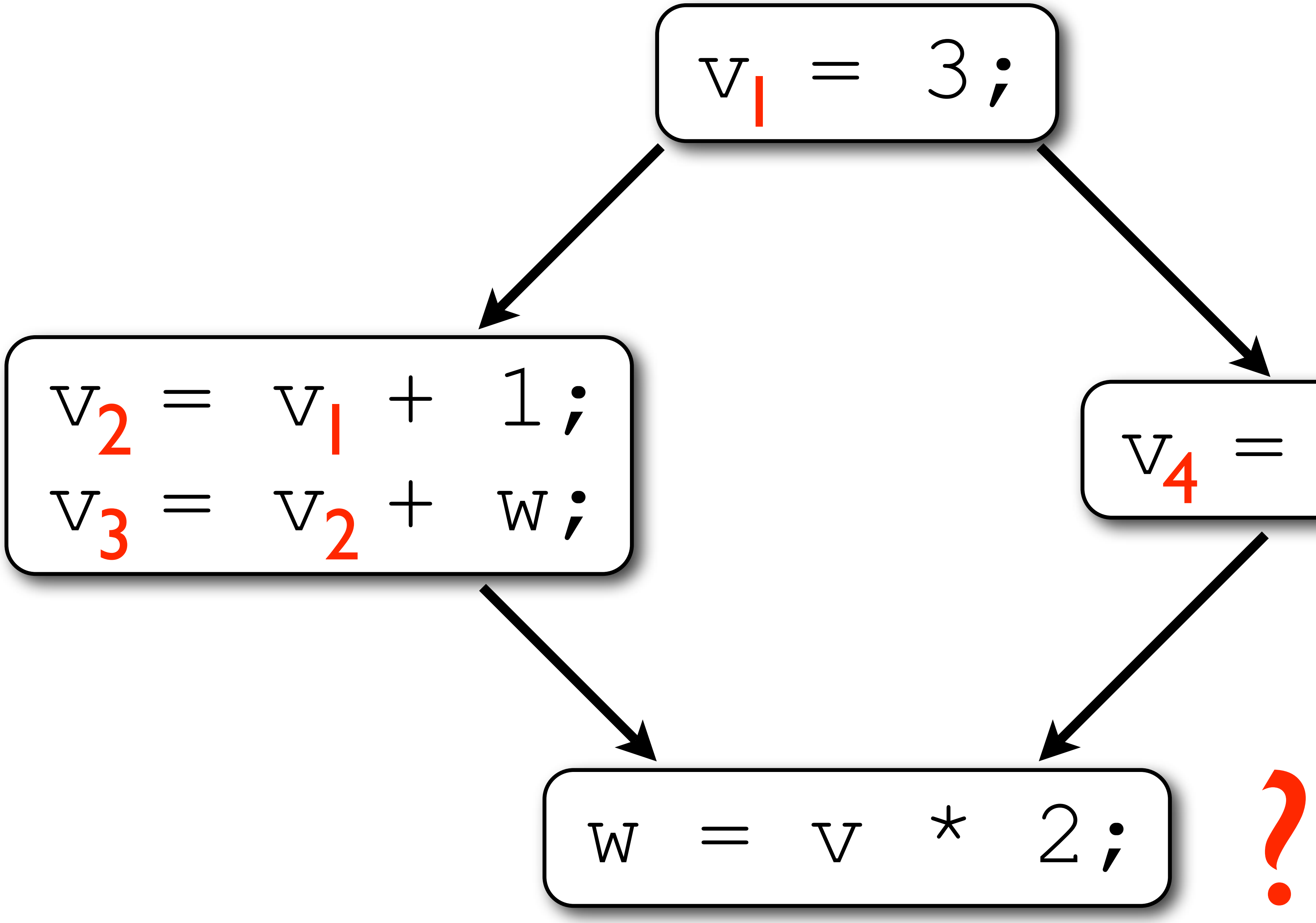
$$
\begin{aligned}
v_1 &= 3; \\
v_2 &= v_1 + 1; \\
v_3 &= v_2 + w_1; \\
w_2 &= v_3 + 2;
\end{aligned}
$$

Static single-assignment

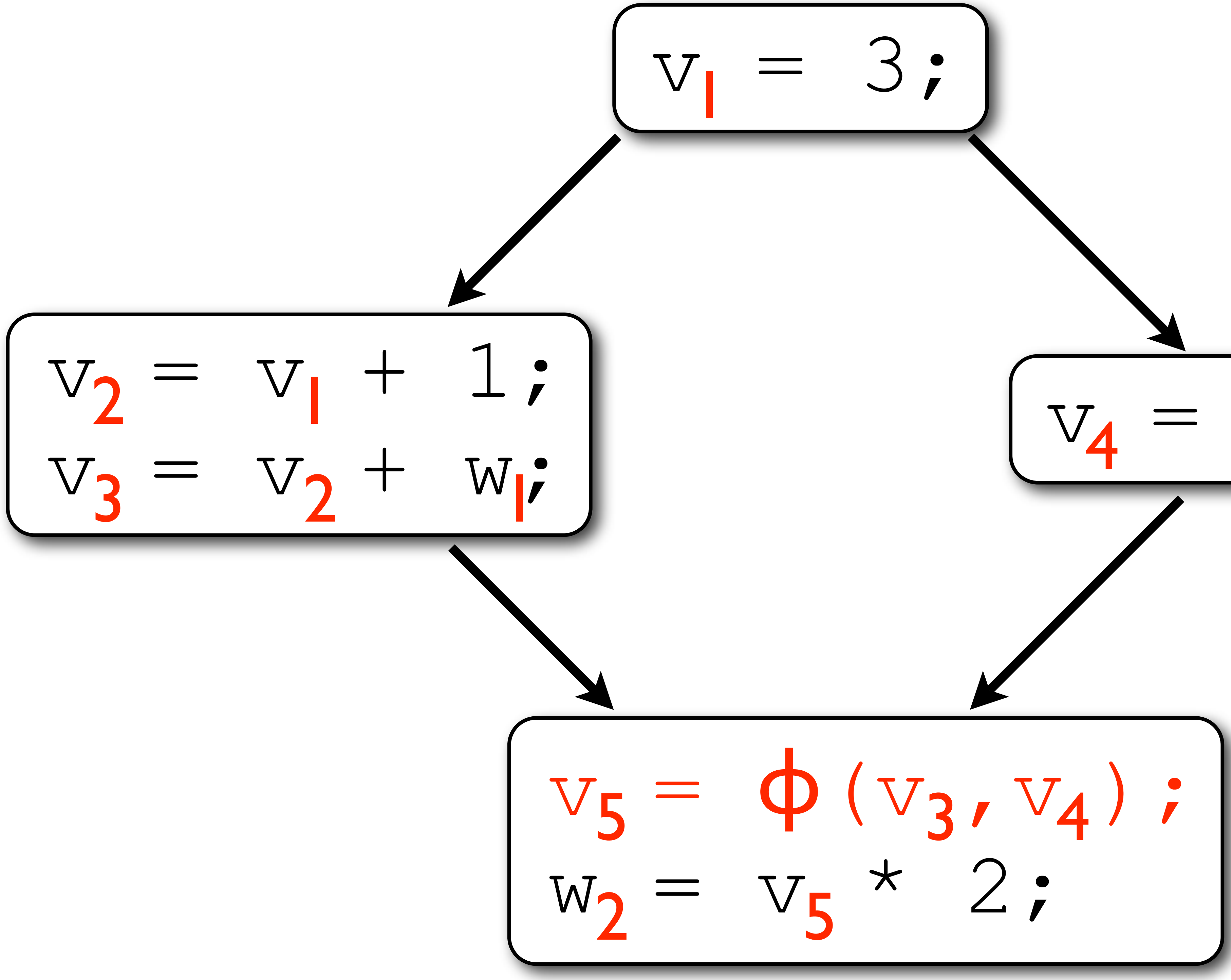
When the program’s control flow is more complex, extra effort is required to retain the original data-flow behaviour.

Where control-flow edges meet, two (or more) differently-named variables must now be merged together.

Static single-assignment



Static single-assignment



Static single-assignment

The ϕ-functions in SSA keep track of which variables are merged at control-flow join points.

They are not executable since they do not record *which* variable to choose (cf. gated SSA form).

Static single-assignment

“Slight lie”: SSA is useful for much more than register allocation!

In fact, the main advantage of SSA form is that, by representing data dependencies as precisely as possible, it makes many optimising transformations simpler and more effective, e.g. constant propagation, loop-invariant code motion, partial-redundancy elimination, and strength reduction.

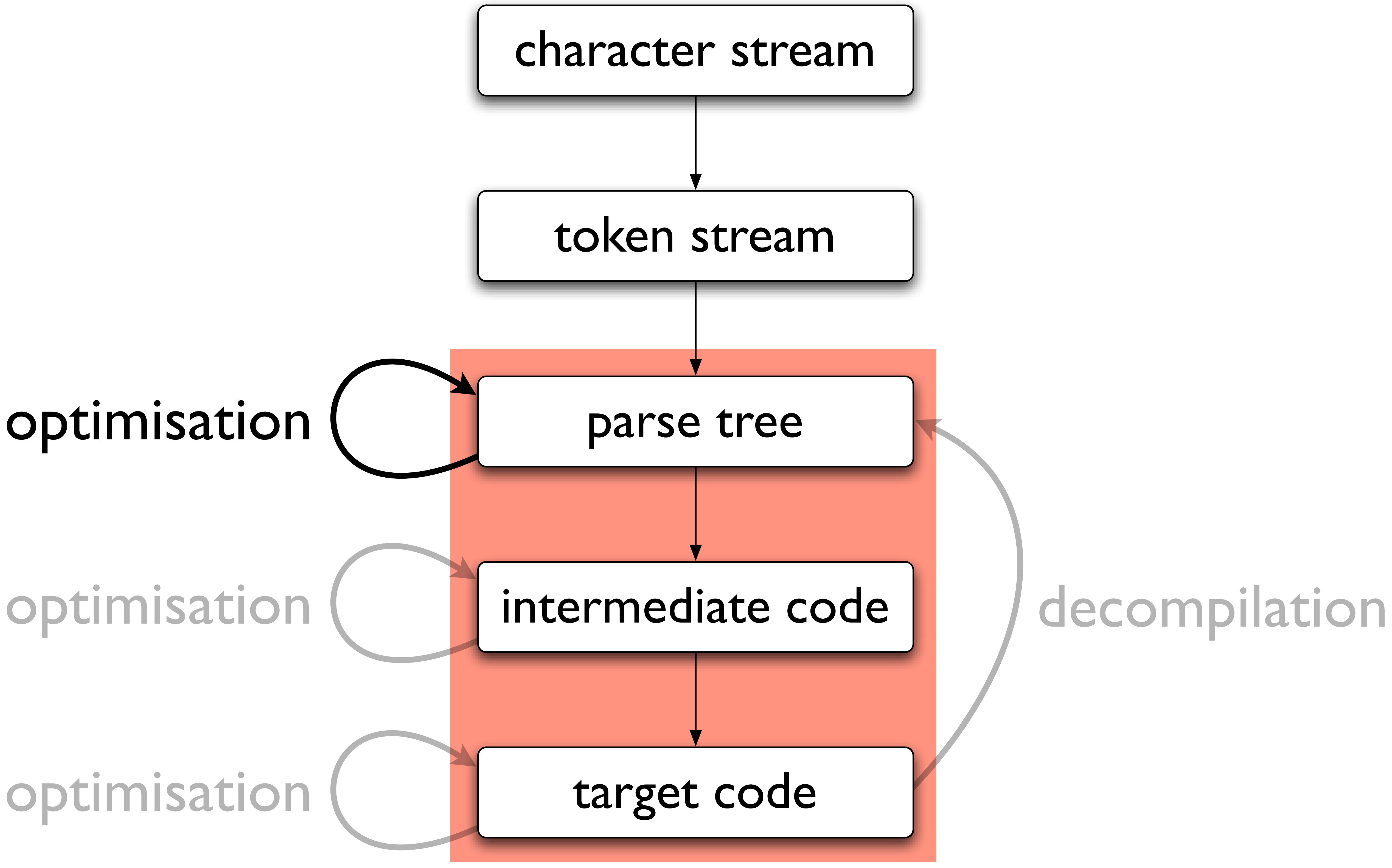
Phase ordering

We now have many optimisations which we can perform on intermediate code.

It is generally a difficult problem to decide in which *order* to perform these optimisations; different orders may be more appropriate for different programs.

Certain optimisations are antagonistic: for example, CSE may superficially improve a program at the expense of making the register allocation phase more difficult (resulting in spills to memory).

Higher-level optimisations



Higher-level optimisations

- More modern optimisations than those in Part A
  - Part A was mostly imperative
  - Part B is mostly functional
- Now operating on syntax of source language vs. an intermediate representation
- Functional languages make the presentation clearer, but many optimisations will also be applicable to imperative programs

Algebraic identities

The idea behind peephole optimisation of intermediate code can also be applied to abstract syntax trees.

There are many trivial examples where one piece of syntax is *always* (algebraically) equivalent to another piece of syntax which may be smaller or otherwise "better"; simple rewriting of syntax trees with these rules may yield a smaller or faster program.

Algebraic identities

$$\begin{tikzcd}
\ldots\,\texttt{e + 0}\,\ldots \arrow[d] \\
\ldots\,\texttt{e}\,\ldots
\end{tikzcd}$$

$$\begin{tikzcd}
\ldots\,\texttt{(e + n) + m}\,\ldots \arrow[d] \\
\ldots\,\texttt{e + (n + m)}\,\ldots
\end{tikzcd}$$

Algebraic identities

These optimisations are boring, however, since they are always applicable to any syntax tree.

We're interested in more powerful transformations which may only be applied when some analysis has confirmed that they are safe.

Algebraic identities

In a lazy functional language,

$$\begin{tikzcd}
\texttt{let x = } \mathbf{e} \texttt{ in if } \mathbf{e'} \texttt{ then } \ldots \texttt{x} \ldots \texttt{ else } \mathbf{e''} \arrow[d] \\
\texttt{if } \mathbf{e'} \texttt{ then let x = } \mathbf{e} \texttt{ in } \ldots \texttt{x} \ldots \texttt{ else } \mathbf{e''}
\end{tikzcd}$$

provided e′ and e′′ do not contain `x`.

This is still *quite* boring.

Strength reduction

More interesting analyses (i.e. ones that aren't purely syntactic) enable more interesting transformations.

Strength reduction is an optimisation which replaces expensive operations (e.g. multiplication and division) with less expensive ones (e.g. addition and subtraction).

It is most interesting and useful when done inside loops.

Strength reduction

For example, it may be advantageous to replace multiplication (`2*e`) with addition (`let x = e in x + x`) as before.

Multiplication may happen a lot inside loops (e.g. using the loop variable as an index into an array), so if we can spot a *recurring* multiplication and replace it with an addition we should get a faster program.

Strength reduction

```
int i;
for (i = 0; i < 100; i++)
{
  v[i] = 0;
}
```

Strength reduction

```
int i; char *p;
for (i = 0; i < 100; i++)
{
  p = (char *)v + 4*i;
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
}
```

Strength reduction

```
int i; char *p;
for (i = 0; i < 100; i++)
{
  p = (char *)v + 4*i;
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
}
```

Strength reduction

```
int i; char *p;
p = (char *)v;
for (i = 0; i < 100; i++)
{
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
  p += 4;
}
```

Strength reduction

```
int i; char *p;
p = (char *)v;
for (i = 0; p < (char *)v + 400; i++)
{
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
  p += 4;
}
```

Strength reduction

```
int i; int *p;
p = v;
for (i = 0; p < v + 100; i++)
{
  *p = 0;
  p++;
}
```

Strength reduction

```
int i; int *p;
p = v;
for (i = 0; p < v + 100; i++)
{
  *p = 0;
  p++;
}
```

Strength reduction

```
int *p;
for (p = v; p < v + 100; p++)
{
  *p = 0;
}
```

Multiplication has been replaced with addition.

Strength reduction

Note that, while this code is now almost optimal, it has obfuscated the intent of the original program.

Don't be tempted to *write* code like this!

For example, when targeting a 64-bit architecture, the compiler may be able to transform the original loop into fifty 64-bit stores, but will have trouble with our more efficient version.

Strength reduction

We are not restricted to replacing multiplication with addition, as long as we have

- induction variable: `i = i` $\oplus$ `c`
- another variable: `j = ` $c_2 \oplus$ `(` $c_1 \otimes$ `i)`

for *some operations* $\oplus$ and $\otimes$ such that

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$

Strength reduction

It might be easier to perform strength reduction on the intermediate code, but only if annotations have been placed on the flowchart to indicate loop structure.

At the syntax tree level, all loop structure is apparent.

Summary

- Live range splitting reduces register pressure
- In SSA form, each variable is assigned to only once
- SSA uses ϕ-functions to handle control-flow merges
- SSA aids register allocation and many optimisations
- Optimal ordering of compiler phases is difficult
- Algebraic identities enable code improvements
- Strength reduction uses them to improve loops