SSA vs ANF

Does FP make a better IR?

Does functional programming make a better intermediate representation?

Yes.

“SSA is Functional Programming”

– Appel 1998

Static Single-Assignment Form

- Invented by imperative compiler writers to make optimisations easier
- Arguments to functions must be atomic (i.e. every sub-expression is named)
- Each variable in a program is assigned only once
- Used by LLVM, GCC, HotSpot, SpiderMonkey, Crankshaft, Dalvik, PyPy, LuaJIT, HHVM, MLton

Static Single-Assignment Form

```
proc fac(x) {
  r ← 1;
  goto L1;
L1:
  r0 ← ϕ(start : r, L1 : r1);
  x0 ← ϕ(start : x, L1 : x1);
  if x0 then
    r1 ← mul(r0, x0);
    x1 ← sub(x0, 1);
    goto L1;
  else
    ret r0;
}

ret fac(10);
```

```
p  ::= proc x(xs) {b} p | e
b  ::= e | b; x:e | b1; x:{b2}
e  ::= x ← ϕ(gs); e
     | x ← v; e
     | x ← v(vs); e
     | goto x;
     | ret v;
     | ret v(vs);
     | if v then e1 else e2
g  ::= l:v
l  ::= x | start
v  ::= x | c
xs ::= x, xs | ε
vs ::= v, vs | ε
gs ::= g, gs | ε
x  ::= variable or label
c  ::= constant
```

Static Single-Assignment Form

```
proc fac(x) {
  r ← 1;
  goto L1;
L1:
  r0 ← ϕ(start : r, L1 : r1);
  x0 ← ϕ(start : x, L1 : x1);
  if x0 then
    r1 ← mul(r0, x0);
    x1 ← sub(x0, 1);
    goto L1;
  else
    ret r0;
}

ret fac(10);
```

```
p  ::= proc x(xs) {b} p | e
b  ::= e | b; x:e | b1; x:{b2}
e  ::= x ← ϕ(gs); e
     | x ← v; e
     | x ← v(vs); e
     | goto x;
     | ret v;
     | ret v(vs);
     | if v then e1 else e2
g  ::= l:v
l  ::= x | start
v  ::= x | c
xs ::= x, xs | ε
vs ::= v, vs | ε
gs ::= g, gs | ε
x  ::= variable or label
c  ::= constant
```

Administrative Normal Form

- Restricted form of lambda terms
- Like SSA, arguments to functions must be atomic
- Doesn’t differentiate between labels and procedures
- Used by GHC, DDC, Icicle, SML/NJ, MLton
- Also called A-Normal Form

Administrative Normal Form

```
letrec fac (x) =
  letrec L1 (r0, x0) =
    if x0 then
      let r1 = mul (r0, x0) in
      let x1 = sub (x0, 1)  in
      L1 (r1, x1)
    else
      r0
  in
    let r = 1 in
    L1 (r, x)
in
  fac (10)
```

```
e  ::= v
     | v(v)
     | let x = v in e
     | let x = v(vs) in e
     | letrec fs in e
     | if v then e1 else e2
f  ::= x(xs) = e
v  ::= x | c
xs ::= x, xs | ε
vs ::= v, vs | ε
fs ::= f; fs | ε
x  ::= variable
c  ::= constant
```

Administrative Normal Form

```
letrec fac (x) =
  letrec L1 (r0, x0) =
    if x0 then
      let r1 = mul (r0, x0) in
      let x1 = sub (x0, 1)  in
      L1 (r1, x1)
    else
      r0
  in
    let r = 1 in
    L1 (r, x)
in
  fac (10)
```

```
e  ::= v
     | v(v)
     | let x = v in e
     | let x = v(vs) in e
     | letrec fs in e
     | if v then e1 else e2
f  ::= x(xs) = e
v  ::= x | c
xs ::= x, xs | ε
vs ::= v, vs | ε
fs ::= f; fs | ε
x  ::= variable
c  ::= constant
```

Conditional Dead Code Elimination

Conditional DCE in SSA

```
proc calcProfit(x, y) {
  if 0 then
    x0 ← x;
    goto L1;
  else
    r0 ← mul(x, y);
    goto L2;
L1:
  …
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}
```

```
L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Conditional DCE in SSA

```
proc calcProfit(x, y) {
  if 0 then
    x0 ← x;
    goto L1;
  else
    r0 ← mul(x, y);
    goto L2;
L1:
  …
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}
```

```
L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Conditional DCE in SSA

```
proc calcProfit(x, y) {
  if 0 then
    x0 ← x;
    goto L1;
  else
    r0 ← mul(x, y);
    goto L2;
L1:
  ...
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}
```

```
L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Conditional DCE in SSA

```
proc calcProfit(x, y) {
  if 0 then
    x0 ← x;
    goto L1;
  else
    r0 ← mul(x, y);
    goto L2;
L1:
  ...
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}

L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Conditional DCE in SSA

```
proc calcProfit(x, y) {
    r0 ← mul(x, y);
    goto L2;
L1:
  ...
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}
```

```
L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Problem: **x0** doesn’t exist

Conditional DCE in SSA

```
proc calcProfit(x, y) {
    r0 ← mul(x, y);
    goto L2;
L1:
  …
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}
```

```
L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Problem: we never jump to L1

Unused Code Elimination in SSA

```
proc calcProfit(x, y) {
    r0 ← mul(x, y);
    goto L2;
L1:
  …
L2:
  p ← ϕ(L1 : x1, start : r0);
  ret p;
}
```

```
L1:
  x1 ← ϕ(start : x0, L1 : x2);
  if y0 then
    x2 ← mul(x1, 2);
    y1 ← sub(y0, 1);
    goto L1;
  else
    goto L2;
```

Unused Code Elimination in SSA

```
proc calcProfit(x, y) {




    r0 ← mul(x, y);
    goto L2;


L2:
  p ← ϕ(          start : r0);
  ret p;
}
```

Redundant ϕ Elimination in SSA

```
proc calcProfit(x, y) {




    r0 ← mul(x, y);
    goto L2;


L2:
  p ← ~~ϕ(          start : ~~r0~~)~~;
  ret p;
}
```

Redundant ϕ Elimination in SSA

```
proc calcProfit(x, y) {




    r0 ← mul(x, y);
    goto L2;


L2:
  p ←                         r0 ;
  ret p;
}
```

Block Merging in SSA

```
proc calcProfit(x, y) {



    r0 ← mul(x, y);
    goto L2;


L2:
  p ←                    r0 ;
  ret p;
}
```

Block Merging in SSA

```
proc calcProfit(x, y) {



    r0 ← mul(x, y);



  p ←                     r0 ;
  ret p;
}
```

Copy Propagation in SSA

```
proc calcProfit(x, y) {



    r0 ← mul(x, y);



  p ← r0 ;
  ret p r0;
}
```

Copy Propagation in SSA

```
proc calcProfit(x, y) {



    r0 ← mul(x, y);




  ret   r0;
}
```


That escalated quickly

Conditional DCE in ANF

```
letrec calcProfit (x, y) =
  if 0 then
    letrec loop (x0, y0) =
      if y0 then
        let x1 = mul (x0, 2)
        let y1 = sub (y0, 1)
        loop (x1, y1)
      else
        x0
    in
      loop (x, y)
  else
    mul (x, y)
in
  ...
```

Conditional DCE in ANF

```
letrec calcProfit (x, y) =
  if 0 then
    letrec loop (x0, y0) =
      if y0 then
        let x1 = mul (x0, 2)
        let y1 = sub (y0, 1)
        loop (x1, y1)
      else
        x0
    in
      loop (x, y)
  else
    mul (x, y)
in
  ...
```

Conditional DCE in ANF

```
letrec calcProfit (x, y) =




        mul (x, y)
in
    …
```

Too easy!


Inlining

Inlining in SSA

```
proc fac0ver(n) {
  goto L1;
L1:
  n0 ← ϕ(start : n, L1 : n1);
  a  ← fac(n0);
  b  ← gt(a, n0);
  if b then
    ret n0;
  else
    n1 ← add(n0, 1);
    goto L1;
}
```

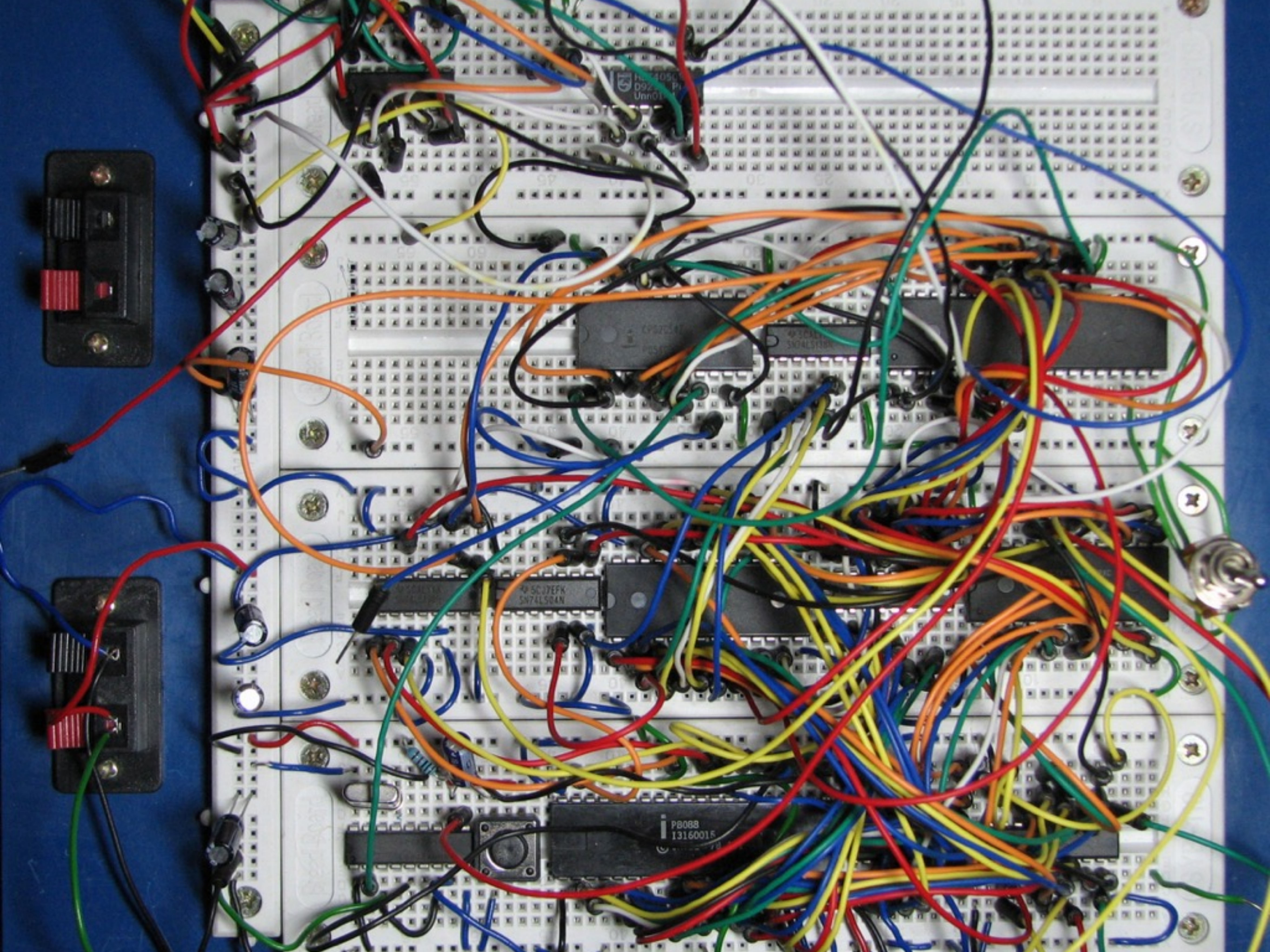
```
proc fac(x) {
  r ← 1;
  goto L4;
L4:
  r0 ← ϕ(start : r, L4 : r1);
  x0 ← ϕ(start : x, L4 : x1);
  if x0 then
    r1 ← mul(r0, x0);
    x1 ← sub(x0, 1);
    goto L4;
  else
    ret r0;
}

ret fac0ver(100);
```

Inlining in SSA

```
proc fac0ver(n) {
  goto L1;
L1:
  n0 ← ϕ(start : n, ~~L1~~L2 : n1);
  x  ← n0
  goto L3;
L2:
  a  ← r0;
  b  ← gt(a, n0);
  if b then
    ret n0;
  else
    n1 ← add(n0, 1);
    goto L1;
L3:
  r ← 1;
  goto L4;
L4:
  r0 ← ϕ(~~start~~L3 : r, L4 : r1);
  x0 ← ϕ(~~start~~L3 : x, L4 : x1);
  if x0 then
    r1 ← mul(r0, x0);
    x1 ← sub(x0, 1);
    goto L4;
  else
    ~~ret r0~~goto L2;
}

ret fac0ver(100);
```


P8088
I3160015

Inlining in ANF

```
letrec
  facOver (n) =
    letrec loop (n0) =
      let a = fac (n0)
      let b = gt (a, n0)
      if b then
        n0
      else
        let n1 = add (n0, 1)
        loop (n1)
    in
      loop (n)


 in
   facOver (100)
```

```
fac (x) =
  letrec L1 (r0, x0) =
    if x0 then
      let r1 = mul (r0, x0)
      let x1 = sub (x0, 1)
      L1 (r1, x1)
    else
      r0
  in
    let r = 1
    L1 (r, x)
```

Inlining in ANF

```
letrec
  facOver (n) =
    letrec loop (n0) =
      letrec L1 …
      in
        let r = 1
        let a = ~~fac (n0)~~ L1 (r, n0)
        let b = gt (a, n0)
        if b then
          n0
        else
          let n1 = add (n0, 1)
          loop (n1)
    in
      loop (n)
in
  facOver (100)
```

```
letrec L1 (r0, x0) =
  if x0 then
    let r1 = mul (r0, x0)
    let x1 = sub (x0, 1)
    L1 (r1, x1)
  else
    r0
```

12
2014
HUBLOT
Brazil
FIFA
2014 FIFA World Cup Brazil
FIFA.com
POWERADE

But what about all the great optimisation passes in LLVM?

- Algorithms designed to operate on SSA programs can readily be translated to operate on ANF programs
- [1] gives a formally proven (in Coq) translation from SSA to ANF
- [1] also shows how to implement Sparse Conditional Constant Propagation (SCCP) [2] on ANF
- Check out my github [3] to see a Haskell implementation of the above

1. Chakravarty, Keller, Zadarnowski. *A functional perspective on SSA optimisation algorithms* (2003)
2. Wegman, Zadeck. *Constant Propagation with Conditional Branches* (1991)
3. https://github.com/jystic/ssa-anf

“In optimizing compilers, data structure choices directly influence the power and efficiency of practical program optimization. A poor choice of data structure can inhibit optimization or slow compilation to the point that advanced optimization features become undesirable.”

– Cytron, Ferrante, Rosen, Wegman & Zadeck 1991

So use ANF!

Further Reading

- Flanagan, Sabry, Duba, Felleisen. *Retrospective: The essence of compiling with continuations* (2010)
- Chakravarty, Keller, Zadarnowski. *A functional perspective on SSA optimisation algorithms* (2003)
- Appel. *SSA is functional programming* (1998)
- Kelsey. A correspondence between *Continuation Passing Style and Static Single Assignment Form* (1995)
- Flanagan, Sabry, Duba, Felleisen. *The essence of compiling with continuations* (1993)
- Cytron, Ferrante, Rosen, Wegman, Zadeck. *Efficiently computing static single assignment form and the control dependence graph* (1991)
- https://github.com/jystic/ssa-anf