EECS 483: Compiler Construction

Lecture 6:
Tail Calls, Imperative Programs

February 2, 2025

Announcements

- Assignment 2 released today, due on Friday February 13.

- Builds on solution to Assignment 1: can use your own Assignment 1 solution or our provided reference solution as a starting point.

Learning Objectives

Clarifications in implementation of booleans

Define when functions are compileable to SSA blocks

Discuss pitfalls of compilation of branch with arguments to x86 Assembly code

Define minimal SSA form, benefits and how to construct it.

How to compile **imperative** code to SSA

Correction from Last time

There was an error in the code generation for intToBool last time. Let's revisit it.

Coercions and Representation

Two different "obvious" ways to handle boolean values at runtime:

- all 64-bit values are valid booleans, zero is false and everything else is true
- only 0 and 1 are valid booleans

The first matches our semantics more closely, but the second is easier to support

x86 Instructions: bitwise operators

and dest, src

or dest, src

bitwise and, or. **Not logical** and, or

```
mov rax, 0xF0
mov rcx, 0x0F
and rax, rcx
```

logical and of 0xF0 and 0x0F is true
bitwise and of 0xF0 and 0x0F is 0

Operations coincide when the only possible inputs are 0 or 1.

Implementing Coercions

Can implement coercions as the assembly or SSA level

1. Assembly level: coerce inputs to booleans before all logical operations

2. SSA level: add a coercion `intToBool` to SSA that is implemented by the assembly coercion

   advantage: can be removed by optimizations

   advantage: simplifies code generation

Lowering to SSA

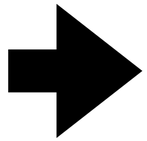
```
true  ➡  1

false ➡  0

x && y ➡  b = intToBool(x)
          c = intToBool(y)
          res = b & c
```

SSA to x86

```
x = intToBool(y)
```

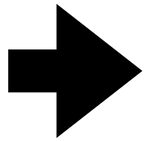


```
mov rax, [rsp - off(y)]
cmp rax, 0
mov rax, 0
setne al
mov [rsp - off(x)], rax
```

mov does not affect RFLAGS

SSA to x86

```
x = y & z
```



```
mov rax, [rsp - off(y)]
mov r10, [rsp - off(z)]
and rax, r10
mov [rsp - off(x)], rax
```

Extending the Snake Language

What source-level programming features would allow us to express cyclic control-flow graphs?

1. Functional: recursive functions, tail calls

2. Imperative: while/for loops, mutable variables

We'll look at these each in turn and study how to compile them to SSA.

Extending the Snake Language

What source-level programming features would allow us to express cyclic control-flow graphs?

1. Functional: recursive functions, tail calls

2. Imperative: while/for loops, mutable variables

We'll look at these each in turn and study how to compile them to SSA.

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. How should we adapt our intermediate representation to new features?
5. How can we generate assembly code from the IR?

Extending the Snake Language

```
pub enum Expr {
    ...
    FunDefs {
        decls: Vec<FunDecl>,
        body: Box<Expr>,
    },
    Call {
        fun_name: Fun,
        args: Vec<Expr>,
    },
}

pub struct FunDecl {
    pub name: String,
    pub parameters: Vec<String>,
    pub body: Expr,
}
```

Examples

recursion

Function definitions are recursive: the function is in scope within its own body as well as in the body of the continuation of its definition

```
def fac(x):
  def loop(x, acc):
    if x == 0:
      acc
    else:
      loop(x - 1, acc * x)
  in
  loop(x, 1)
in
fac(10)
```

Examples

mutual recursion

Function definitions separated by an and are **mutually recursive.** Mutually recursive functions are all in scope of each other.

```
def even(x):
  def evn(n):
    if n == 0:
      true
    else:
      odd(n - 1)
  and
  def odd(n):
    if n == 0:
      false
    else:
      even(n - 1)
  in
  if x >= 0:
    evn(x)
  else:
    evn(-1 * x)
in
even(24)
```

Examples

variable capture

Function definitions can access variables in scope at their definition site.

```
def pow(m, n):
  def loop(n, acc):
    if n == 0:
      acc
    else:
      loop(n - 1, acc * m)
  in
  loop(n, 1)
```

Functions as Blocks

When can a function call be compiled to a branch with arguments?

When it is in **tail position**, i.e., the result of the called function is immediately returned by the caller.

If this is the case, the call can be compiled directly to a branch.

Otherwise it is a **true call** and implementing it requires storing data on the call stack. Revisit this next week

Tail Position

```
def fac(x):
  def loop(x, acc):
    if x == 0:
      acc
    else:
      loop(x - 1, acc * x)
  in
  loop(x, 1)
in
fac(10)
```

```
def factorial(x):
  if x == 0:
    1
  else:
    x * factorial(x - 1)
in
factorial(6)
```

Tail Position

When is an expression in **tail position?**

- It depends on the **context**, not the expression itself

Tail Position

```
pub struct Prog<Var, Fun> {
    pub param: (Var, SrcLoc),
    pub main: Expr<Var, Fun>,
}
```

The **main** expression is in tail position, as its result is the result of the main function

Tail Position

```
Prim {
    prim: Prim,
    args: Vec<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

```
Call {
    fun: Fun,
    args: Vec<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The **args** of a prim or a call are **never** in tail position, as we always have to do something else after evaluating them (the prim/call)

Tail Position

```
Let {
    bindings: Vec<Binding<Var, Fun>>,
    body: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The expressions in the **bindings** are **never** in tail position, as we always have to do something else after evaluating them (the let body)

The **body** of the let is in tail position if the let itself is in tail position

Tail Position

```
If {
    cond: Box<Expr<Var, Fun>>,
    thn: Box<Expr<Var, Fun>>,
    els: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The expressions in the **cond** position is **never** in tail position, as we always have to do something else after evaluating them (the if)

The **thn** and **els** branches are in tail position if the if itself is in tail position

Tail Position

```
FunDefs {
    decls: Vec<FunDecl<Var, Fun>>,
    body: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The **body** of a fundef is in tail position if the FunDefs expression itself is in tail position

Tail Position

```
pub struct FunDecl<Var, Fun> {
    pub name: Fun,
    pub params: Vec<(Var, SrcLoc)>,
    pub body: Expr<Var, Fun>,
    pub loc: SrcLoc,
}
```

The **body** of a FunDecl is always in tail position

Function definitions to Blocks

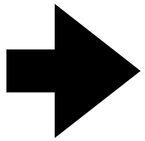
Compile each function definition directly to a corresponding block.

Compile mutually-recursive function definitions to mutually recursive blocks

Compile **tail** function calls to branch with arguments, with left-to-right evaluation order of arguments:

Tail calls to Branches

```
f(e1,e2,e3)
```



```
... ;; e1 code
x1 = ...
... ;; e2 code
x2 = ...
... ;; e3 code
x3 = ...
br f(x1,x2,x3)
```

No continuation to use
because call is assumed to be in tail position

Compiling Branch with Arguments

Semantically, a branch with arguments is a **simultaneous** move, all of the variables get updated at once.

This is not supported in our target architecture, in reality we have to sequentialize those moves into a sequence.

Compiling Branch with Arguments

Semantically, a branch with arguments is a **simultaneous** move, all of the variables get updated at once.

This is not supported in our target architecture, in reality we have to sequentialize those moves into a sequence.

Can cause correctness issues if we are not careful

Compiling Branch with Arguments

```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
br f(5, y)
```

where is each variable stored?

```
x: rsp - 8
y: rsp - 16
a: rsp - 16
b: rsp - 24
z: rsp - 32
w: rsp - 40
```

Compiling Branch with Arguments

```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
br f(5, y)
```

```
mov [rsp - 16], 5 ;; a = 5
mov rax, [rsp - 16]
mov [rsp - 24], rax ;; b = y
jmp f
```

Compiling Branch with Arguments

easy, sub-optimal solution

To ensure we don't overwrite memory we are about to use, we can introduce extra temporaries for the arguments.

Since we allocate variables based on their nested definitions, and the block we branch to is in scope, this guarantees that the new temporaries occur higher on the stack than their targets, so they won't be overwritten

Revisit this to get a more efficient allocation scheme when we perform register allocation

Compiling Branch with Arguments

easy, sub-optimal solution

```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
a2 = 5
b2 = y
br f(a2, b2)
```

```
mov rax, [rsp - 24]
mov [rsp - 16], rax ;; a = a2
mov rax, [rsp - 32]
mov [rsp - 24], rax ;; b = b2
jmp f
```

Functional to SSA

Summary:

If a function is only ever tail-called locally, it can be compiled directly to an SSA block with arguments. Tail calls can then be compiled to branch with arguments

A tail call is a call to a function in tail position: the result of the function call is immediately returned.

Functional to SSA

It's easy to map functional code to an SSA code since SSA is essentially functional.

But, is that the **best** translation of the functional code? Probably not!

Minimal SSA

An SSA program is **minimal** if it uses as few block arguments (phi nodes) as possible.

Useful for optimization: branching to a block with arguments is compiled to a **mov,** potentially causing memory access. Want to reduce these as much as possible.

Minimal SSA

The following SSA is **not** minimal

$$
\begin{aligned}
&\textbf{function } f_1() = \textbf{let } v = 1,\ z = 8,\ y = 4 \\
&\qquad\qquad\qquad\qquad \textbf{in } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_2(v, z, y) = \textbf{let } x = 5 + y,\ y = x \times z,\ x = x - 1 \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in if } x = 0 \textbf{ then } f_3(y, v) \textbf{ else } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_3(y, v) = \textbf{let } w = y + v \textbf{ in } w \textbf{ end} \\
&\textbf{in } f_1() \textbf{ end}
\end{aligned}
$$

SSA Minimization

Minimizing SSA form consists of two phases:

1. Block Sinking: pushing block definitions lower in the SSA AST, so that more variables are in scope of its definition

2. Parameter dropping: removing unnecessary block parameters

Block Sinking

Push function definitions inside of others if they are **dominated**. I.e., given f and g, if g is only ever called inside f or g, then f **dominates** g, and so g's definition could be sunk inside of the definition of f.

$$
\begin{aligned}
&\textbf{function } f_1() = \textbf{let } v = 1,\ z = 8,\ y = 4 \\
&\qquad\qquad\qquad\qquad \textbf{in } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_2(v, z, y) = \textbf{let } x = 5 + y,\ y = x \times z,\ x = x - 1 \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in if } x = 0 \textbf{ then } f_3(y, v) \textbf{ else } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_3(y, v) = \textbf{let } w = y + v \textbf{ in } w \textbf{ end} \\
&\textbf{in } f_1() \textbf{ end}
\end{aligned}
$$

which of f1, f2, f3 dominates which?

Block Sinking

f1 dominates f2 dominates f3. Sink blocks accordingly:

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(v, z, y) =
            let x = 5 + y, y = x × z, x = x − 1
            in if x = 0
                then function f3(y, v) = let w = y + v in w end
                        in f3(y, v) end
                else f2(v, z, y)
            end
        in f2(v, z, y) end
    end
in f1() end
```

Parameter Dropping

If a parameter **x** is always instantiated with **y** or itself, then we can remove **x** and replace all occurrences with **y** as long as it is in the scope of **y**.

Parameter Dropping

Which parameters can be dropped?

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(v, z, y) =
            let x = 5 + y, y = x × z, x = x − 1
            in if x = 0
                then function f3(y, v) = let w = y + v in w end
                        in f3(y, v) end
                else f2(v, z, y)
            end
        in f2(v, z, y) end
    end
in f1() end
```

Parameter Dropping

Which parameters can be dropped?

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(v, z, y) =
            let x = 5 + y, y = x × z, x = x − 1
             in if x = 0
                then function f3() = let w = y + v in w end
                     in f3() end
                else f2(v, z, y)
            end
        in f2(v, z, y) end
    end
in f1() end
```

Parameter Dropping

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(y) =
            let x = 5 + y, y = x × z, x = x − 1
            in if x = 0
                then function f3() = let w = y + v in w end
                        in f3() end
                else f2(y)
            end
        in f2(y) end
    end
in f1() end
```

Minimal: only block arg is y and this does take on multiple values

Extending the Snake Language

What source-level programming features would allow us to express cyclic control-flow graphs?

1. Functional: recursive functions, tail calls

2. Imperative: loops, mutable variables

We'll look at these each in turn and study how to compile them to SSA.

Imperative Snake Language

Imperative Snake Language "Imp"

- Mutable variables

- statement-expression distinction

- while loops

- return/break/continue

```
var m = 100;
var n = 25;
while !(m == n) {
  if m < n {
    n := n - m
  } else {
    m := m - n
  }
};
return m
```

Imperative Snake Language

concrete syntax

⟨*block*⟩:

| ⟨statement⟩
| ⟨statement⟩ `;` ⟨statement⟩

⟨*statement*⟩:

| `var` ***IDENTIFIER*** `=` ⟨expr⟩
| ***IDENTIFIER*** `:=` ⟨expr⟩
| `if` ⟨expr⟩ `{` ⟨block⟩ `}`
| `if` ⟨expr⟩ `{` ⟨block⟩ `}` `else` `{` ⟨block⟩ `}`
| `while` ⟨expr⟩ `{` ⟨block⟩ `}`
| `continue`
| `break`
| `return` ⟨expr⟩

⟨*expr*⟩:

| ***IDENTIFIER***
| ***NUMBER***
| `true`
| `false`
| `!` ⟨expr⟩
| ⟨prim1⟩ `(` ⟨expr⟩ `)`
| ⟨expr⟩ ⟨prim2⟩ ⟨expr⟩
| `(` ⟨expr⟩ `)`

Imperative Snake Language

abstract syntax

```
pub enum Block {
    End(Box<Statement>),
    Sequence(Box<Statement>, Box<Block>),
}
```

```
pub enum Statement {
    VarDecl(String, Expression),
    VarUpdate(String, Expression),
    If(Expression, Block, Block),
    IfElse(Expression, Block, Block),
    While(Expression, Block),
    Continue,
    Break,
    Return(Expression),
}
```

```
pub enum Expression {
    Var(String),
    Num(i64),
    Bool(bool),
    Prim(Prim, Vec<Expression>),
}
```

Imperative Snake Language

well-formedness

Still have a notion of scope, shadowing:

1. Check variables are declared before use

2. Shadowing is allowed, semantically shadowed var is a **different** mutable variable

Translate away shadowing to unique variable names to avoid headaches, as usual

Imperative Snake Language

well-formedness

```
var x = y + z;
return x
```

undeclared var y, z

similar to existing scope checker

Imperative Snake Language

well-formedness

If implementing a procedure that returns a value, need to ensure that every code path ends in a return

```
...
if b {
    ...
    return x;
} else {
  x := 5
}
```

Imperative Snake Language

well-formedness

Naked break/continue:

Verify that break/continue operations only occur inside of an enclosing while loop

```
while x != 0 {
  x := x - 1
  if y > 10 {
    continue
  }
  ...
}
continue
```

Imperative Snake Language

semantics

Each variable acts like a 64-bit "register"

When evaluating, need to keep track of the current state of all the variables

Imperative Snake Language

semantics

```
var x = 10;
...
if x != y {
   var x = 14;
   ...
   x := x + 1
}
return x;
```

shadowed variables should not be overwritten. Making variable names unique makes this easier to get right

Imperative Snake Language

semantics

while loop:

check the condition expression

true: execute the block and repeat

false: execute the next statement

break:

in a while loop, goto the next statement after the loop

continue:

in a loop, goto the beginning of the loop

Imperative to SSA

Step 1: Expressions, variable declarations

Step 2: variable updates

Step 3: Join Points

Step 4: Loops

Step 5: Break, Continue, Return

Imperative to SSA

Step 1: Expressions, variable declarations

Expressions are defined just as in Adder: generate temporaries and use continuations to turn tree of operations into straightline code

Variable declarations are implemented just as with Let: a var declaration in Imp becomes a variable assignment in SSA

```
var x = 10;
var p = (x * x) + 5 * x + 7;
...
```

```
x = 10
tmp0 = x * x
tmp1 = 5 * x
tmp2 = tmp0 + tmp2
p = tmp2 + 7
...
```

Imperative to SSA

Step 2: Variable Updates

```
var x = 10;
x := (x * 2) + 1;
x := x + x
...
```

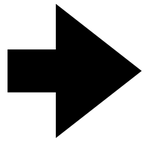
how to compile to SSA?

idea: the updated x acts like it's shadowing the original. Treat it as an assignment to a new variable

Imperative to SSA

Step 2: Variable Updates

```
var x = 10;
x := (x * 2) + 1;
x := x + x
...
```



```
x0 = 10
tmp0 = x0 * 2
x1 = tmp0 + 1
x2 = x1 + x1
...
```

Keep track in an environment of the current "version" of each variable in scope

Imperative to SSA

Step 2: Variable Updates

Simple idea: replace mutable updates with assignments to a new variable

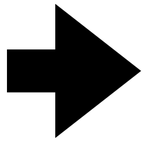
in straightline code, mutable variables are just shadowing!

Imperative to SSA

Step 2: If

Imperative to SSA

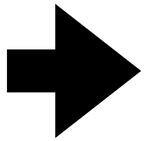
```
...
var x = 10;
if y {
  x = x + 1
} else {
  x = x * 2
  x = x - 1
}
return x
```



```
x0 = 10
thn():
  x1 = x0 + 1
  br ??
els():
  x2 = x0 * 2
  x3 = x2 - 1
  br ??
cbr y thn() els()
```

Imperative to SSA

```
...
var x = 10;
if y {
  x = x + 1
} else {
  x = x * 2
  x = x - 1
}
return x
```



Join points!

```
x0 = 10
jn(x4):
  ret x4
thn():
  x1 = x0 + 1
  br jn(x1)
els():
  x2 = x0 * 2
  x3 = x2 - 1
  br jn(x3)
cbr y thn() els()
```

Imperative to SSA

Step 2: If

Generate join points for if statements.

In an imperative program, join points are parameterized not just by a single variable, but by as many as can be updated in the two branches.

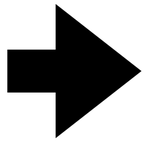
Need to calculate which variables to include in the join point:

Simplest algorithm is called **crude** ϕ-node insertion: add **every** variable that is in scope to the join point.

Rely on a later SSA-minimization pass to remove unnecessary parameters

Unnecessary Parameters

```
...
var x = 10;
var z = 7;
if y {
  x = x + 1
} else {
  y = x * 2
  x = x - 1
}
var w = z * x
return w + y
```



```
...
x0 = 10
z0 = 7
jn(x4, y1, z1):
  w = z1 * x4
  tmp = w + y1
  ret tmp
thn():
  x1 = x0 + 1
  br jn(x1, y0, z0)
els():
  y2 = x0 * 2
  x2 = x1 - 1
  br jn(x2, y2, z0)
cbr y0 thn() els()
```

Imperative to SSA

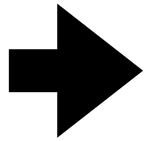
Step 4: while loops

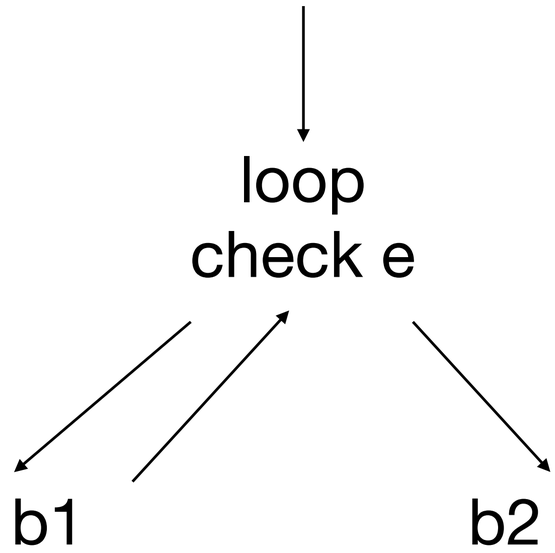
encode semantics using SSA blocks

which blocks in a loop are join points?

Imperative to SSA

```
while e {
   b1
}
b2
```

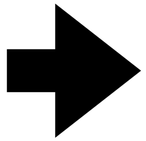




Notice: loop has 2 predecessors, so it is a join point, add block parameters

Imperative to SSA

```
while e {
  b1
}
b2
```



```
loop(...): ;; loop is a join point, include all in-scope vars
  done():
    ... ;; compiled code for b2
  body():
    ... ;; compiled code for b1
    br loop(...)
  ...
  c = ... ;; compiled code for e
  cbr c body() done()
br loop(...)
```

Imperative to SSA

Step 5: return, break, continue

Return is easy: just compile the expression and produce the ret terminator

Break, continue: depend on the **context**

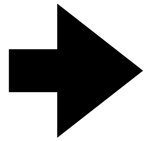
when we enter a while loop, we make blocks for the entry point and exit point

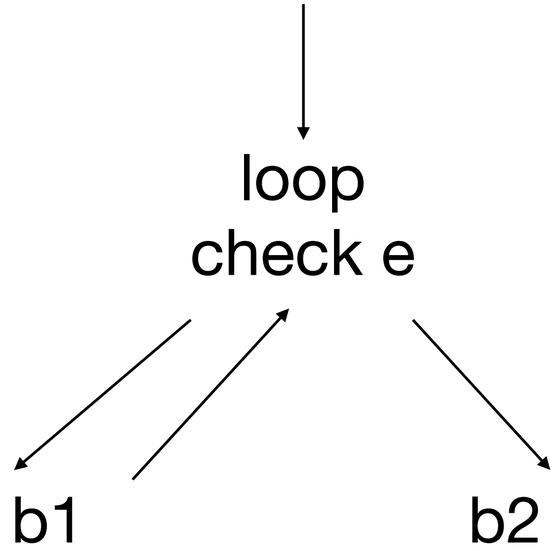
continue: branch to entry of loop

break: branch to exit of loop

Imperative to SSA

```
while e {
   b1
}
b2
```

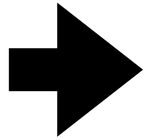


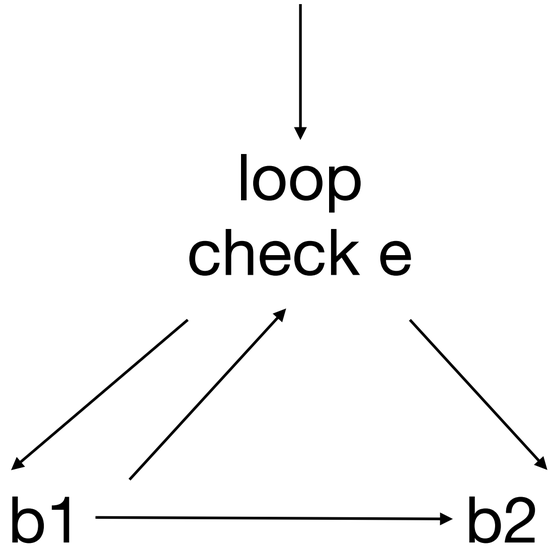


Notice: loop has 2 predecessors, so it is a join point, add block parameters

Imperative to SSA

```
while x != 0 {
  x := x - 1
  if y > 10 {
    break
  }
  ...
}
b2
```



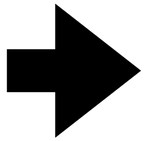


If we can break, then b1 can branch directly to b2

if break is used, b2 is **also** a join point

Imperative to SSA

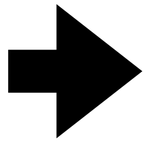
```
while e {
  b1
}
b2
```



```
loop(...): ;; loop is a join point, include all in-scope vars
  done(...): ;; done is a join point as well because of break
    ... ;; compiled code for b2
  body():
    ... ;; compiled code for b1
    br loop(...)
  ...
  c = ... ;; compiled code for e
  cbr c body() done(...)
br loop(...)
```

Imperative to SSA

```
var m = 100
var n = 25
while ! (m == n) {
  if m < n {
    n := n - m
  } else {
    m := m - n
  }
}
return m
```



```
m0 = 100
n0 = 25
loop(m2, n2):
  done(m1,n2):
    return m1
  body(m3, n3):
    lt():
      n4 = n3 - m3
      br loop(m3, n4)
    gt():
      m4 = m3 - n3
      br loop(m4, n3)
    b = m3 < n3
    cbr b lt() gt()
  c = m2 == n2
  d = not c
  cbr d body(m2, n2) done(m2, n2)
loop(m0, n0)
```

Minimal SSA

An SSA program is **minimal** if it uses as few block arguments (phi nodes) as possible.

Useful for optimization: branching to a block with arguments is compiled to a **mov,** potentially causing memory access. Want to reduce these as much as possible.

Minimal SSA Form

Translating Imperative code to SSA using crude phi node insertion produces **very** non-minimal SSA: many extra block parameters

But because imperative code is well-structured, block sinking is not necessary, blocks are already nested inside their immediate dominators

Only need to implement parameter dropping.

> Theorem: crude **phi** node insertion + parameter dropping produces minimal SSA

Why all the trouble?

Modern compiler infrastructure for imperative languages:

input program: mutates variables directly, variables similar semantics to registers

middle end: translates into SSA, functional intermediate representation where variables are never mutated

backend: translate out of SSA, map variables to registers (or memory), mutate their values

SSA Benefits

Programs are **easier** to reason about

Common sub-expression elimination:

y and z have the same definition, so just replace z with y.

Valid with SSA

Not valid in imperative code

$$x = 1;$$
$$y = x + 1;$$

$$z = x + 1;$$

SSA Benefits

Programs are **easier** to reason about

Common sub-expression elimination:

y and z have the same definition, so just replace z with y.

Valid with SSA

Not valid in imperative code

$$x = 1;$$
$$y = x + 1;$$
$$x = 2;$$
$$z = x + 1;$$

SSA Benefits

Program analyses can be implemented more **efficiently**.

Can set up data structures that map variable uses directly to their definitions. Skips over a great deal of irrelevant information.

In an imperative program variables can be updated anywhere, putting the program in SSA form makes the dataflow information easier to access

$$x_1 = 1;$$
$$y = x_1 + 1;$$
$$x_2 = 2;$$
$$z = x_2 + 1;$$

SSA Benefits

When program analysis is **easier**:

1. More efficient generated code: Easier for compiler writers to implement more and better analyses/ optimizations

2. More efficient compiler: accessibility of information in SSA form allows efficient data structures for program analysis, since more information is manifest in the program format

SSA History, Benefits

Further Reading: SSA Book Chapter 1