EECS 483: Compiler Construction

Lecture 7:
Loops, Mutable Variables

February 5, 2025

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 ot 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

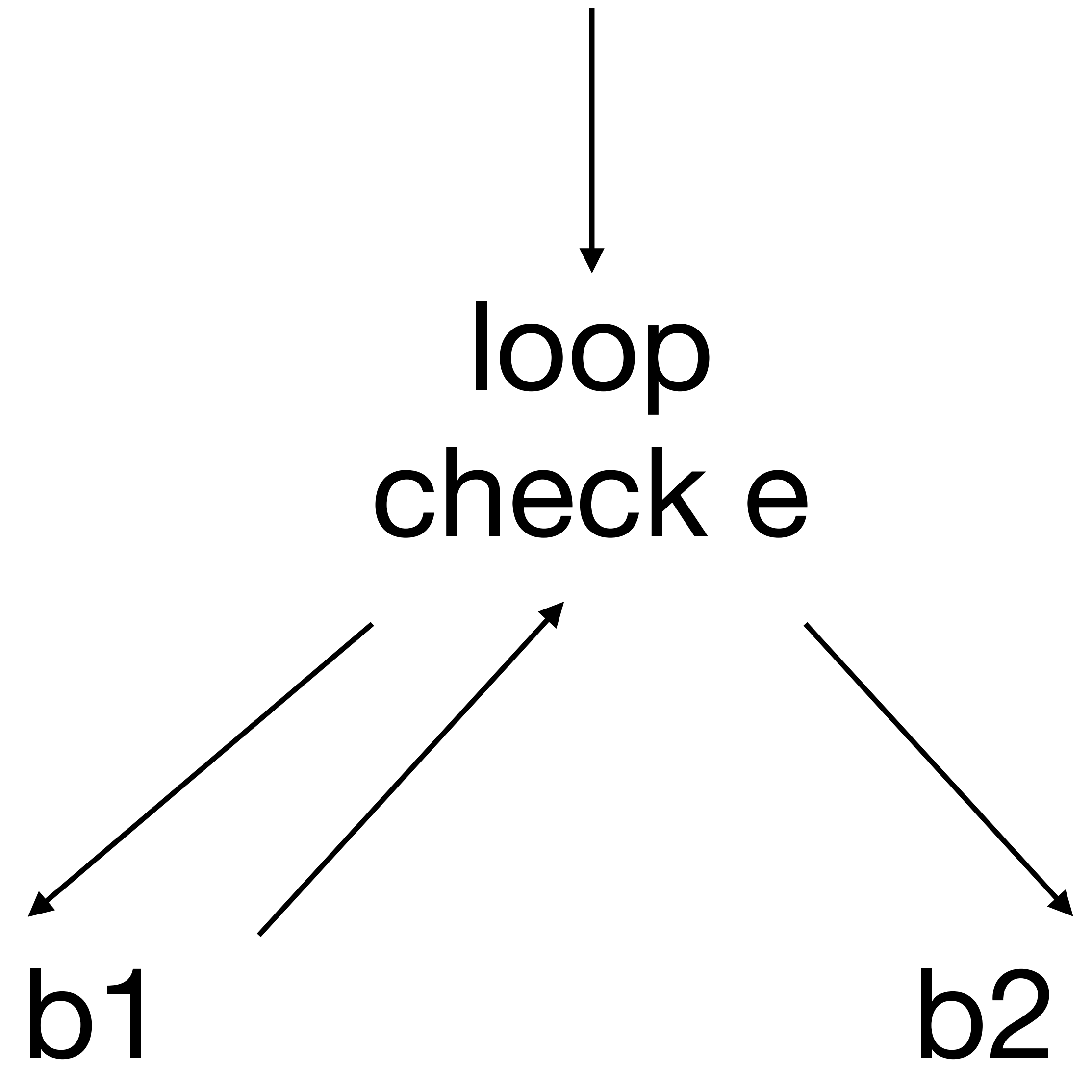
Step 4: while loops

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 loop(...) 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**

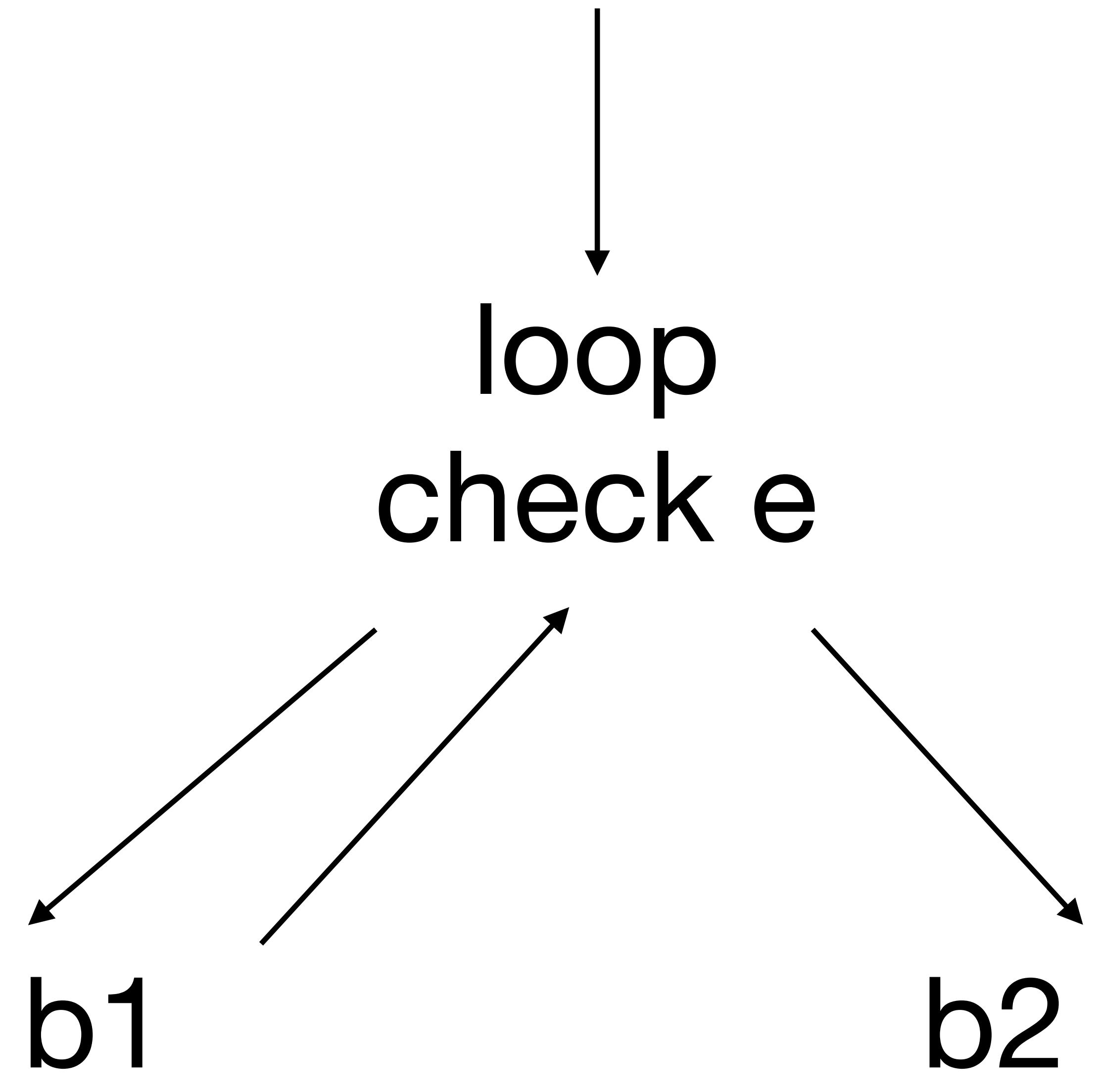
when we enter a while loop, we make blocks for the entry point and exit point

continue: branch to entry of loop

break: branch to exit of loop

Imperative to SSA

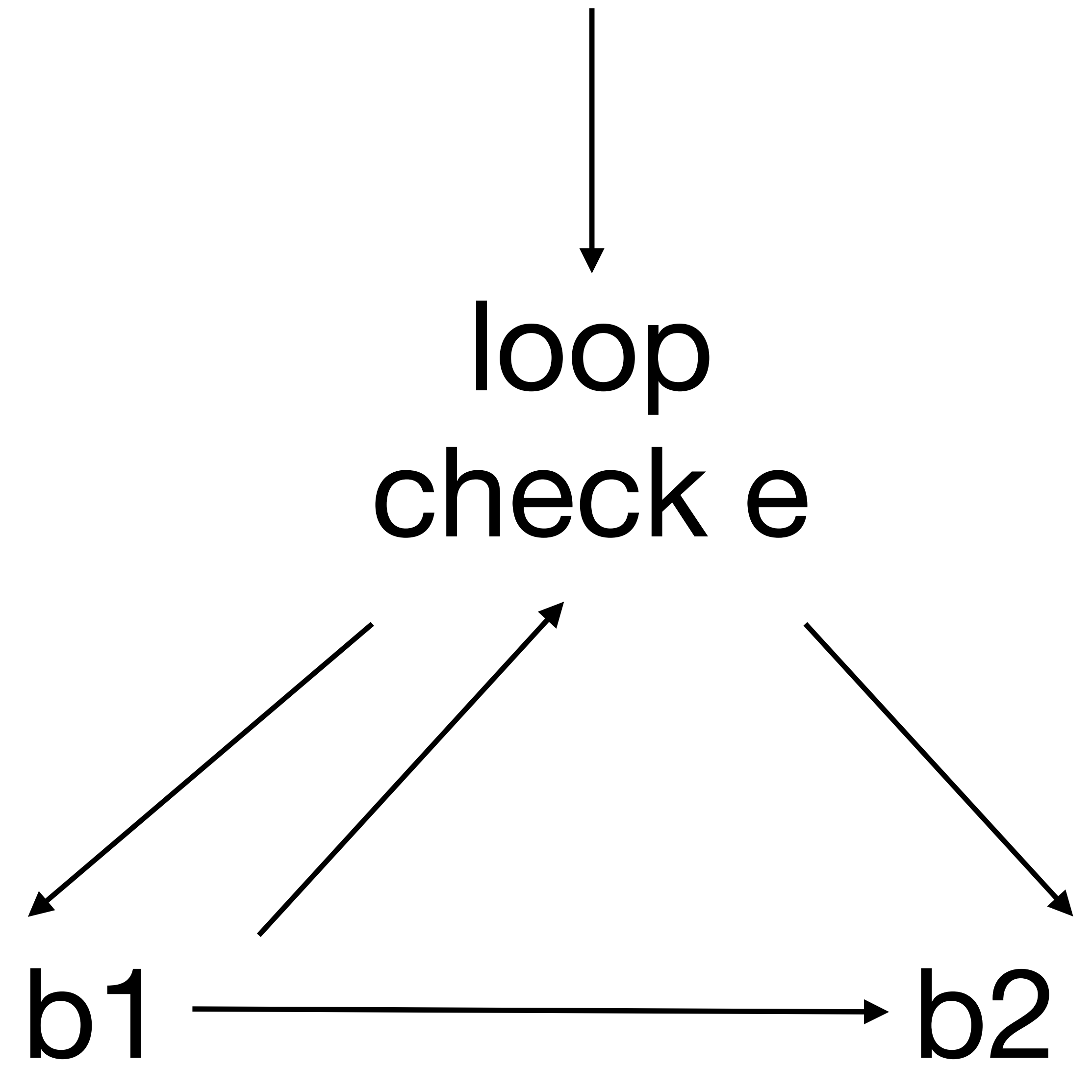
```
while e {
  b1
}
b2
```



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 loop(...) done(...)
br loop(...)
```

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

$$
\begin{aligned}
x &= 1; \\
y &= x + 1; \\
x &= 2; \\
z &= x + 1;
\end{aligned}
$$

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