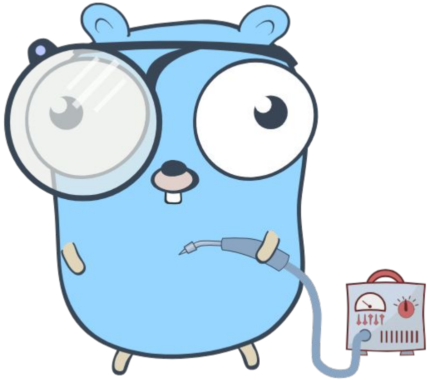
Exploring the Go Compiler: Adding a “four” loop



GopherCon 2024

Riley Thompson

Objective

Add 2 new keywords to the Go compiler: “four” loops and unless statements

Inspired by George Hotz’s stream adding “four” loops to Clang

Disclaimer: I don’t think these would actually be good additions to Go!

“four” Loop

```
four i := 8; i <= 20; i++ {
    fmt.Println(i)
}

for i := 8; i <= 20; i += 4 {
    fmt.Println(i)
}
```

```
Output:
8
12
16
20
```

Unless Statement

```
unless i%8 == 0 {
    fmt.Println(i, “is not divisible by 8”)
}

if !(i%8 == 0) {
    fmt.Println(i, “is not divisible by 8”)
}
```

Example

```go
for i := 8; i <= 20; i += 4 {
    if !(i%8 == 0) {
        fmt.Println(i, “is not divisible by 8”)
        continue
    }
    fmt.Println(i, “is divisible by 8”)
}
```

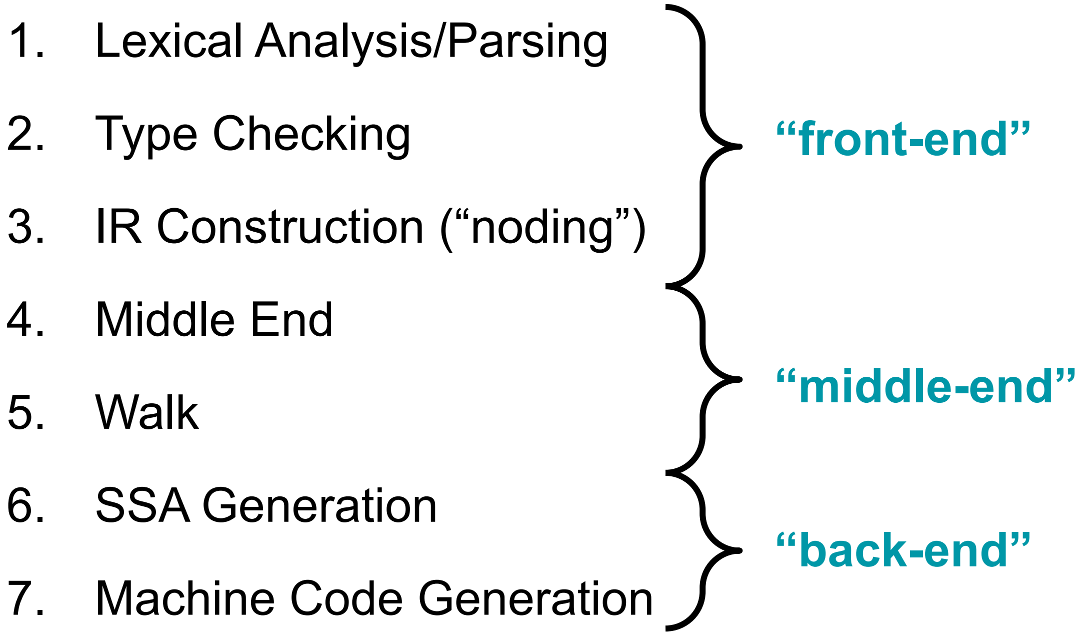
Example

```
Output:
8 is divisible by 8
12 is not divisible by 8
16 is divisible by 8
20 is not divisible by 8
```

Example

```
four i := 8; i <= 20; i++ {
    unless i%8 == 0 {
        fmt.Println(i, “is not divisible by 8”)
        continue
    }
    fmt.Println(i, “is divisible by 8”)
}
```

Compiler Overview



1. Lexical Analysis/Parsing
2. Type Checking
3. IR Construction (“noding”)
4. Middle End
5. Walk
6. SSA Generation
7. Machine Code Generation

“front-end”

“middle-end”

“back-end”

Lexical Analysis and Parsing

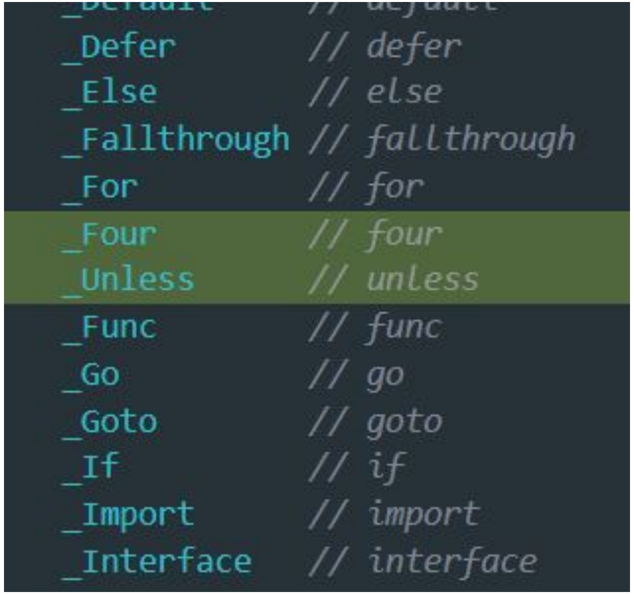
The source file is scanned character by character and tokenized.

A recursive descent parser processes these tokens and converts them to a concrete syntax tree.

- recursive descent parser: it works top-down, from package-level type and function definitions down to individual expressions
- concrete syntax tree: it is an exact representation of the source file

The syntax tree also has positional info for error reporting and debugging purposes.

Lexical Analysis and Parsing

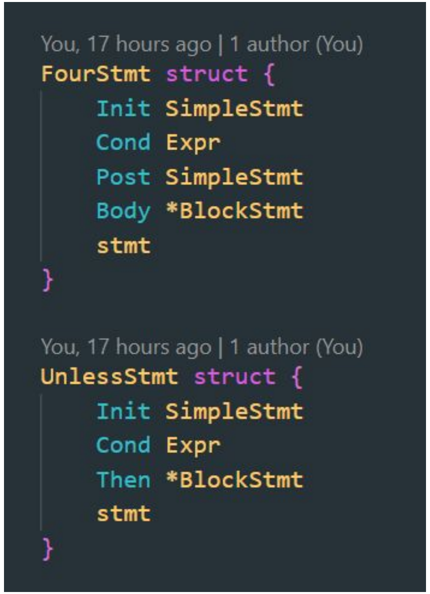


```
_Default      // default
_Defer        // defer
_Else         // else
_Fallthrough  // fallthrough
_For          // for
_Four         // four
_Unless       // unless
_Func         // func
_Go           // go
_Goto         // goto
_If           // if
_Import       // import
_Interface    // interface
```

```
$ go generate tokens.go
```

Lexical Analysis and Parsing

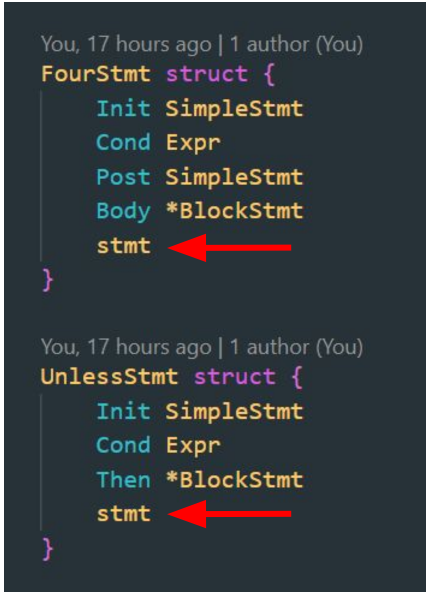
```
four <init>; <cond>; <post> {
    <body>
}

unless <init>; <cond> {
    <then>
}
```



```
You, 17 hours ago | 1 author (You)
FourStmt struct {
    Init SimpleStmt
    Cond Expr
    Post SimpleStmt
    Body *BlockStmt
    stmt
}

You, 17 hours ago | 1 author (You)
UnlessStmt struct {
    Init SimpleStmt
    Cond Expr
    Then *BlockStmt
    stmt
}
```

Lexical Analysis and Parsing

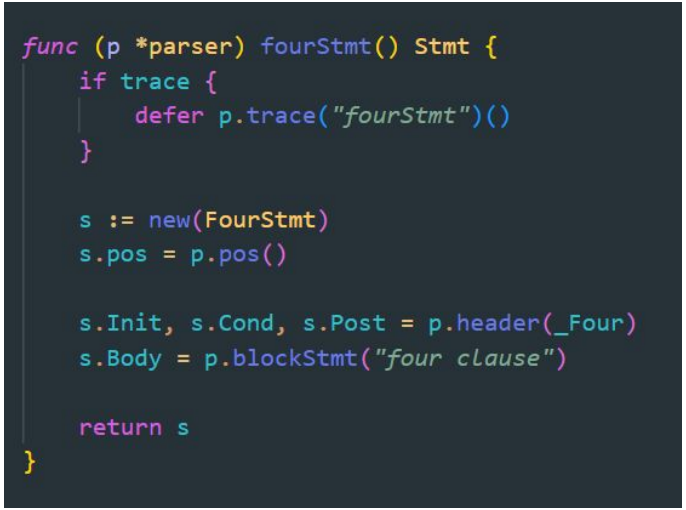
```
four <init>; <cond>; <post> {
    <body>
}

unless <init>; <cond> {
    <then>
}
```



```
You, 17 hours ago | 1 author (You)
FourStmt struct {
    Init SimpleStmt
    Cond Expr
    Post SimpleStmt
    Body *BlockStmt
    stmt
}

You, 17 hours ago | 1 author (You)
UnlessStmt struct {
    Init SimpleStmt
    Cond Expr
    Then *BlockStmt
    stmt
}
```

Lexical Analysis and Parsing

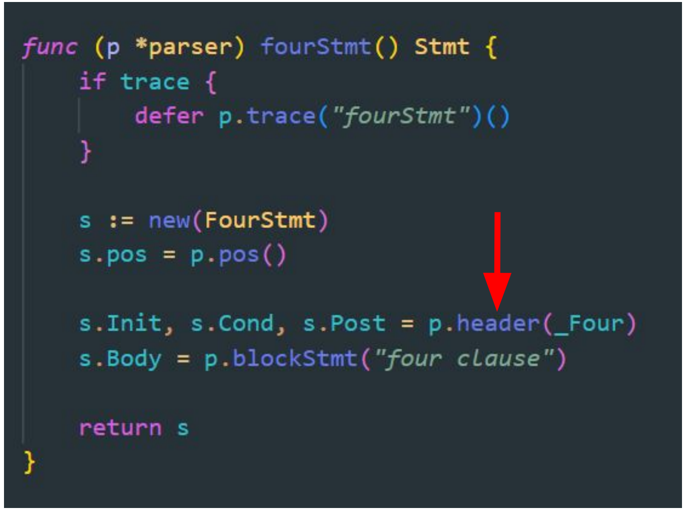
```
four <init>; <cond>; <post> {
    <body>
}

unless <init>; <cond> {
    <then>
}
```



```
func (p *parser) fourStmt() Stmt {
	if trace {
		defer p.trace("fourStmt")()
	}

	s := new(FourStmt)
	s.pos = p.pos()

	s.Init, s.Cond, s.Post = p.header(_Four)
	s.Body = p.blockStmt("four clause")

	return s
}
```

Lexical Analysis and Parsing

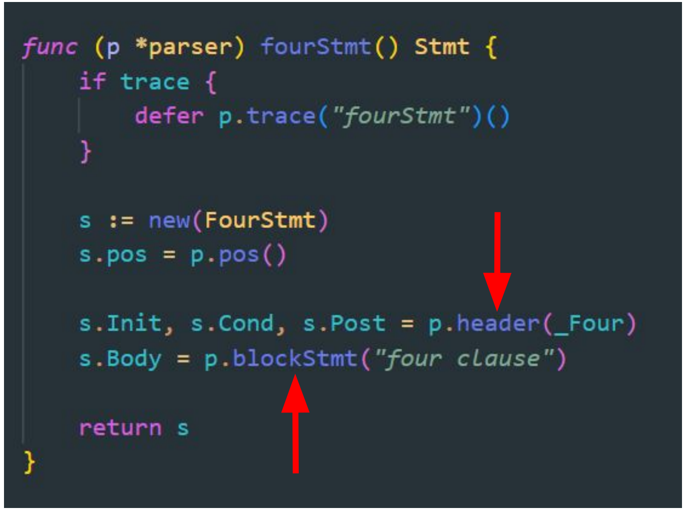
```
four <init>; <cond>; <post> {
    <body>
}

unless <init>; <cond> {
    <then>
}
```



```go
func (p *parser) fourStmt() Stmt {
	if trace {
		defer p.trace("fourStmt")()
	}

	s := new(FourStmt)
	s.pos = p.pos()

	s.Init, s.Cond, s.Post = p.header(_Four)
	s.Body = p.blockStmt("four clause")

	return s
}
```

Lexical Analysis and Parsing

```
four <init>; <cond>; <post> {
    <body>
}

unless <init>; <cond> {
    <then>
}
```



```go
func (p *parser) fourStmt() Stmt {
	if trace {
		defer p.trace("fourStmt")()
	}

	s := new(FourStmt)
	s.pos = p.pos()

	s.Init, s.Cond, s.Post = p.header(_Four)
	s.Body = p.blockStmt("four clause")

	return s
}
```

Type Checking

Type checking is done in several phases over the syntax tree, e.g.

- Name resolution: mapping identifiers to language objects
- Constant folding: computing compile-time constants
- Type inference: computing the type of every expression and checking for compliance with language specification

Type Checking

```
case *syntax.UnlessStmt:
    check.openScope(s, "unless")
    defer check.closeScope()

    check.simpleStmt(s.Init)
    var x operand
    check.expr(nil, &x, s.Cond)
    if x.mode != invalid && !allBoolean(x.typ) {
        check.error(s.Cond, InvalidCond, "non-boolean condition in unless statement")
    }
    check.stmt(inner, s.Then)
```

Type Checking

```
case *syntax.UnlessStmt:
    check.openScope(s, "unless")
    defer check.closeScope()

    check.simpleStmt(s.Init)
    var x operand
    check.expr(nil, &x, s.Cond)
    if x.mode != invalid && !allBoolean(x.typ) {
        check.error(s.Cond, InvalidCond, "non-boolean condition in unless statement")
    }
    check.stmt(inner, s.Then)
```

Type Checking

```
case *syntax.UnlessStmt:
    check.openScope(s, "unless")
    defer check.closeScope()

    check.simpleStmt(s.Init)
    var x operand
    check.expr(nil, &x, s.Cond)
    if x.mode != invalid && !allBoolean(x.typ) {
        check.error(s.Cond, InvalidCond, "non-boolean condition in unless statement")
    }
    check.stmt(inner, s.Then)
```

Type Checking

```
case *syntax.UnlessStmt:
    check.openScope(s, "unless")
    defer check.closeScope()

    check.simpleStmt(s.Init)
    var x operand
    check.expr(nil, &x, s.Cond)
    if x.mode != invalid && !allBoolean(x.typ) {
        check.error(s.Cond, InvalidCond, "non-boolean condition in unless statement")
    }
    check.stmt(inner, s.Then)
```

Type Checking

```
case *syntax.UnlessStmt:
    check.openScope(s, "unless")
    defer check.closeScope()

    check.simpleStmt(s.Init)
    var x operand
    check.expr(nil, &x, s.Cond)
    if x.mode != invalid && !allBoolean(x.typ) {
        check.error(s.Cond, InvalidCond, "non-boolean condition in unless statement")
    }
    check.stmt(inner, s.Then)
```

IR Construction (“noding”)

IR: Intermediate Representation, a representation better suited for optimization and translation.

Convert from a type checked concrete syntax tree to an abstract syntax tree.

Go calls this process “noding”.

IR Construction (“noding”)

You, 3 minutes ago | 1 author (You)

```go
type FourStmt struct {
    miniStmt
    Label *types.Sym
    Cond  Node
    Post  Node
    Body  Nodes
}

func NewFourStmt(pos src.XPos, init Node, cond, post Node, body []Node) *FourStmt {
    n := &FourStmt{Cond: cond, Post: post}
    n.pos = pos
    n.op = OFOUR
    if init != nil {
        n.init = []Node{init}
    }
    n.Body = body
    return n
}
```

```
$ go generate node.go
$ go run mknode.go
```

IR Construction (“noding”)

You, 3 minutes ago | 1 author (You)

```go
type FourStmt struct {
	miniStmt
	Label *types.Sym
	Cond   Node
	Post   Node
	Body   Nodes
}

func NewFourStmt(pos src.XPos, init Node, cond, post Node, body []Node) *FourStmt {
	n := &FourStmt{Cond: cond, Post: post}
	n.pos = pos
	n.op = OFOUR
	if init != nil {
		n.init = []Node{init}
	}
	n.Body = body
	return n
}
```

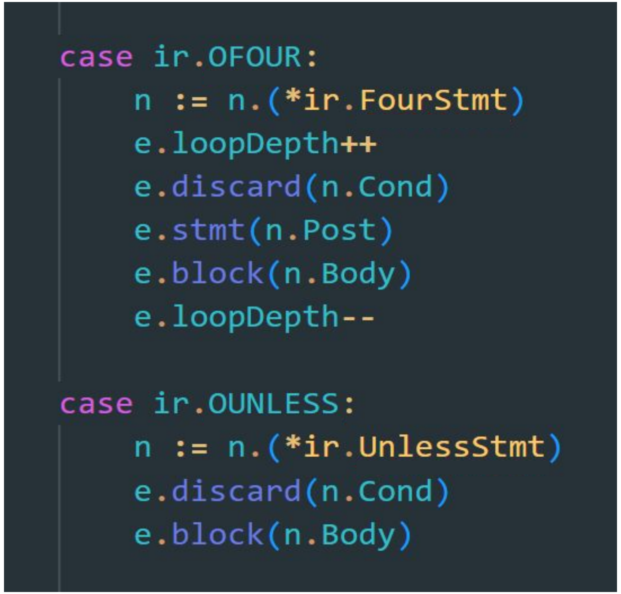
```
$ go generate node.go
$ go run mknode.go
```

Middle End

Several optimization passes are performed on the AST, e.g.

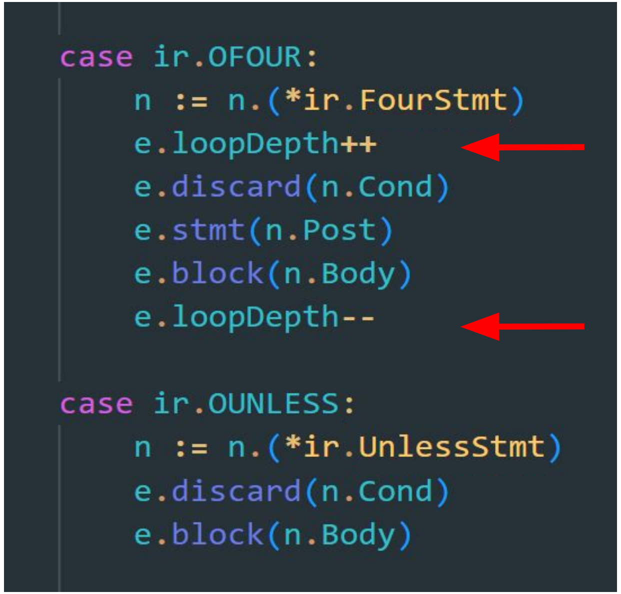
- Dead code elimination
- Devirtualization
- Function inlining
- Escape analysis

Escape Analysis



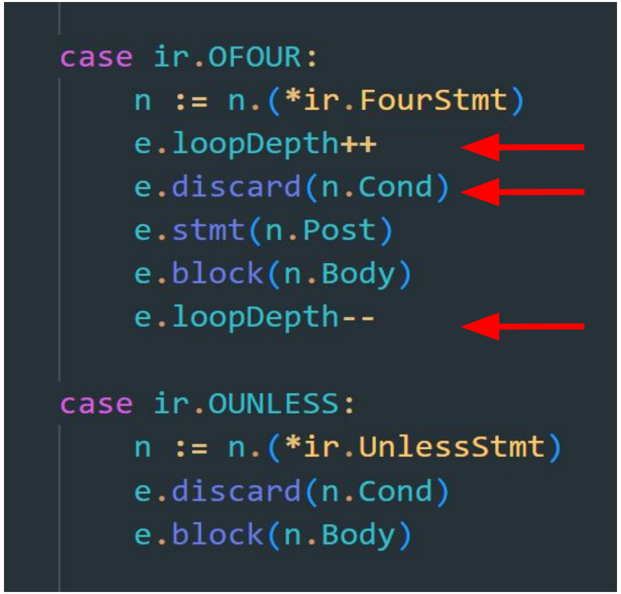
```
case ir.OFOUR:
    n := n.(*ir.FourStmt)
    e.loopDepth++
    e.discard(n.Cond)
    e.stmt(n.Post)
    e.block(n.Body)
    e.loopDepth--

case ir.OUNLESS:
    n := n.(*ir.UnlessStmt)
    e.discard(n.Cond)
    e.block(n.Body)
```

Escape Analysis



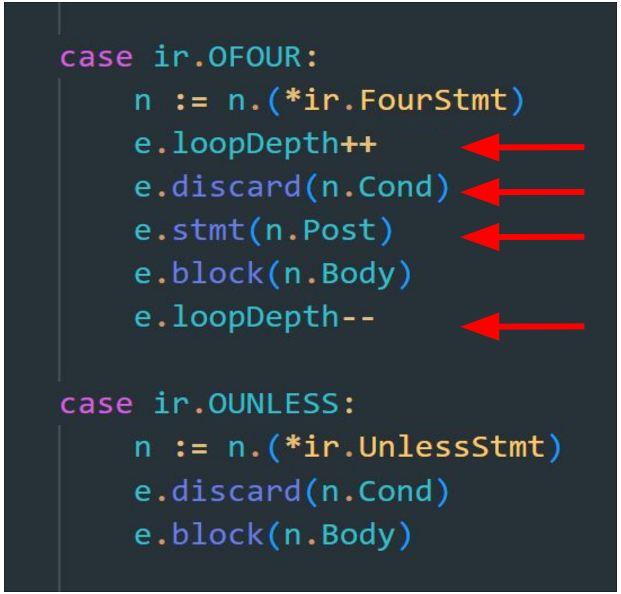
```
case ir.OFOUR:
    n := n.(*ir.FourStmt)
    e.loopDepth++
    e.discard(n.Cond)
    e.stmt(n.Post)
    e.block(n.Body)
    e.loopDepth--

case ir.OUNLESS:
    n := n.(*ir.UnlessStmt)
    e.discard(n.Cond)
    e.block(n.Body)
```

Escape Analysis



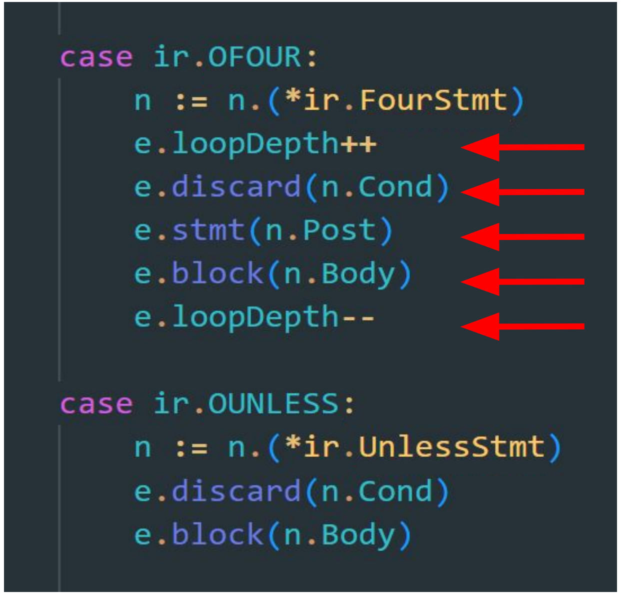
```
case ir.OFOUR:
    n := n.(*ir.FourStmt)
    e.loopDepth++
    e.discard(n.Cond)
    e.stmt(n.Post)
    e.block(n.Body)
    e.loopDepth--

case ir.OUNLESS:
    n := n.(*ir.UnlessStmt)
    e.discard(n.Cond)
    e.block(n.Body)
```

Escape Analysis



```
case ir.OFOUR:
    n := n.(*ir.FourStmt)
    e.loopDepth++
    e.discard(n.Cond)
    e.stmt(n.Post)
    e.block(n.Body)
    e.loopDepth--

case ir.OUNLESS:
    n := n.(*ir.UnlessStmt)
    e.discard(n.Cond)
    e.block(n.Body)
```

Escape Analysis



```
case ir.OFOUR:
    n := n.(*ir.FourStmt)
    e.loopDepth++
    e.discard(n.Cond)
    e.stmt(n.Post)
    e.block(n.Body)
    e.loopDepth--

case ir.OUNLESS:
    n := n.(*ir.UnlessStmt)
    e.discard(n.Cond)
    e.block(n.Body)
```

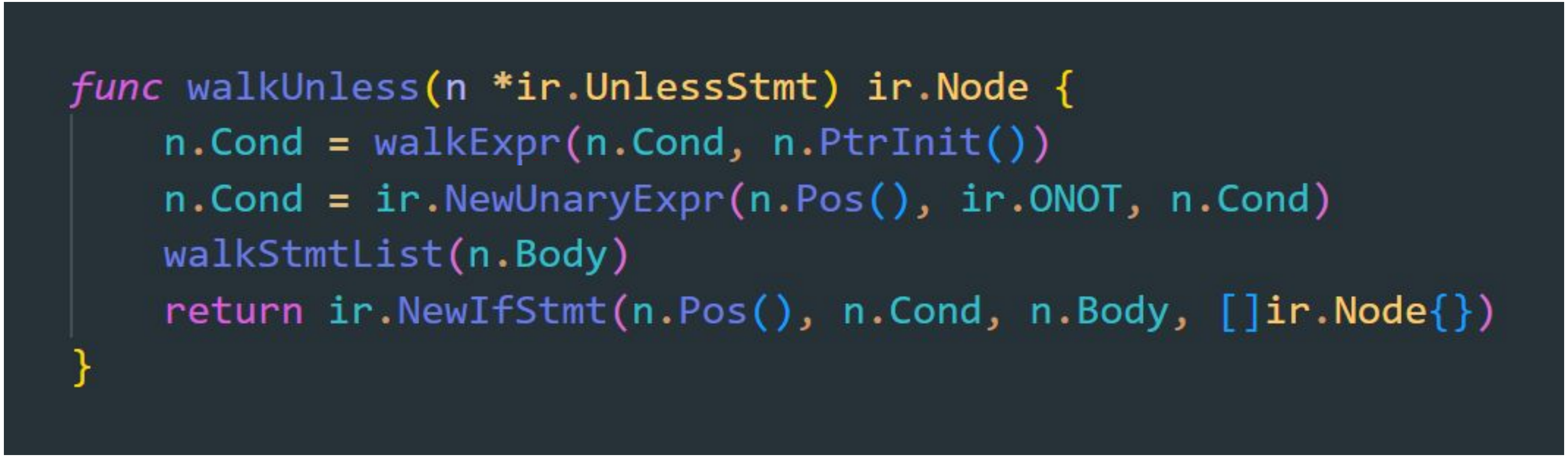
Walk

The walk is the final pass over the AST in the Go compiler.

It has 2 steps:

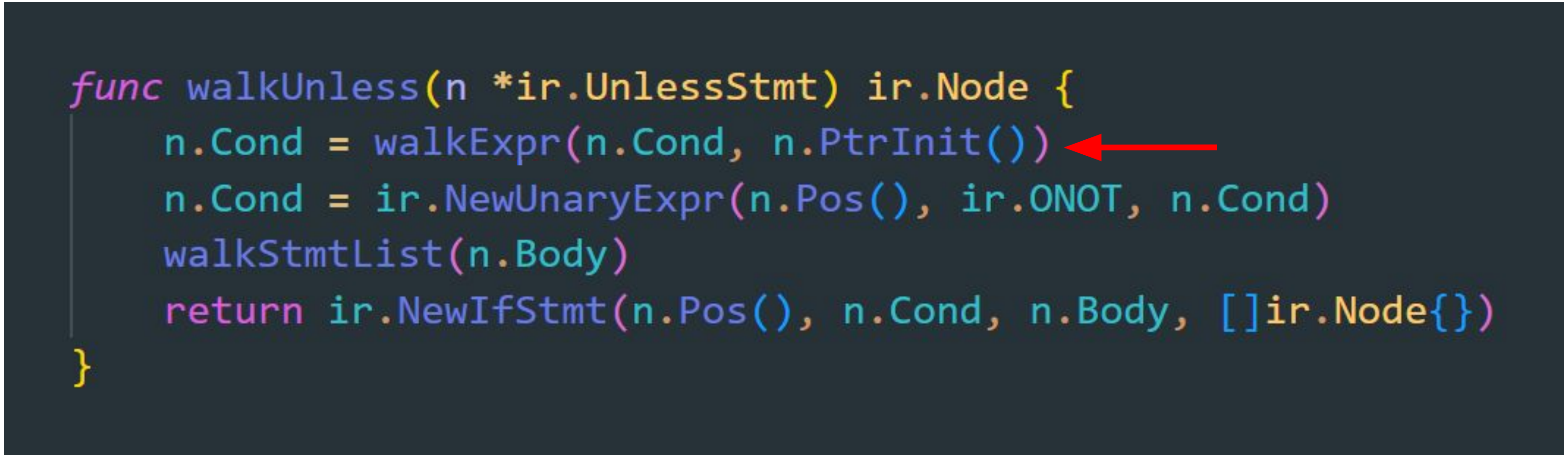
- “Order” step: convert complex statements into simpler ones, introducing temporary variables and respecting order of evaluation.
- “Desugar” step: convert high level constructs into more primitive ones, e.g. range clauses in for loops rewritten with an explicit loop variable.

Walk



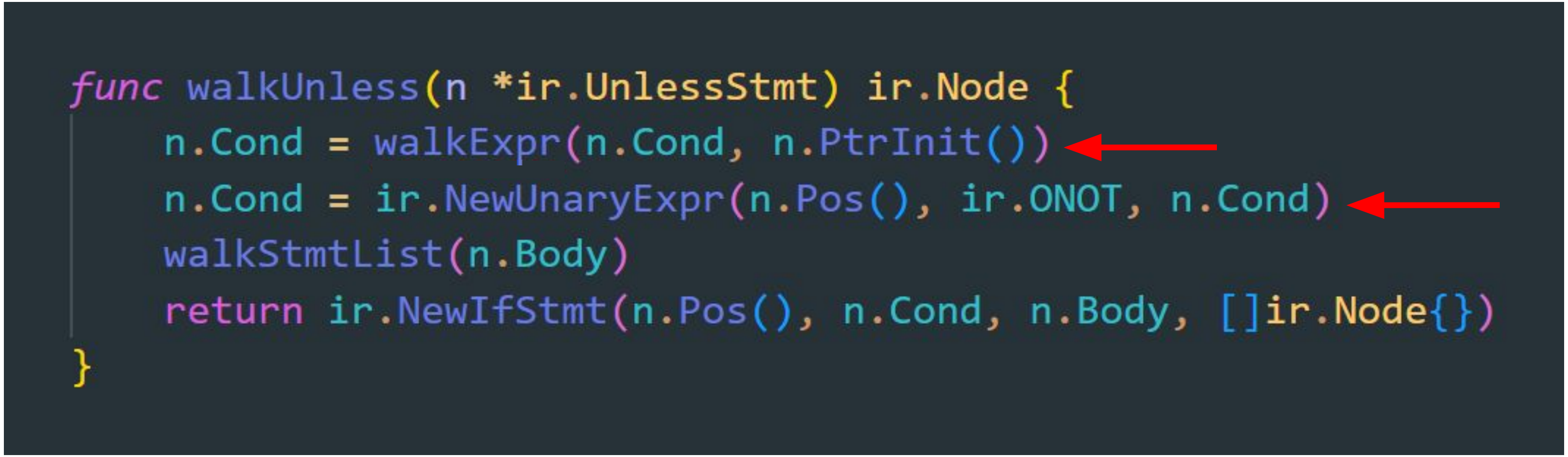
```go
func walkUnless(n *ir.UnlessStmt) ir.Node {
    n.Cond = walkExpr(n.Cond, n.PtrInit())
    n.Cond = ir.NewUnaryExpr(n.Pos(), ir.ONOT, n.Cond)
    walkStmtList(n.Body)
    return ir.NewIfStmt(n.Pos(), n.Cond, n.Body, []ir.Node{})
}
```

Walk



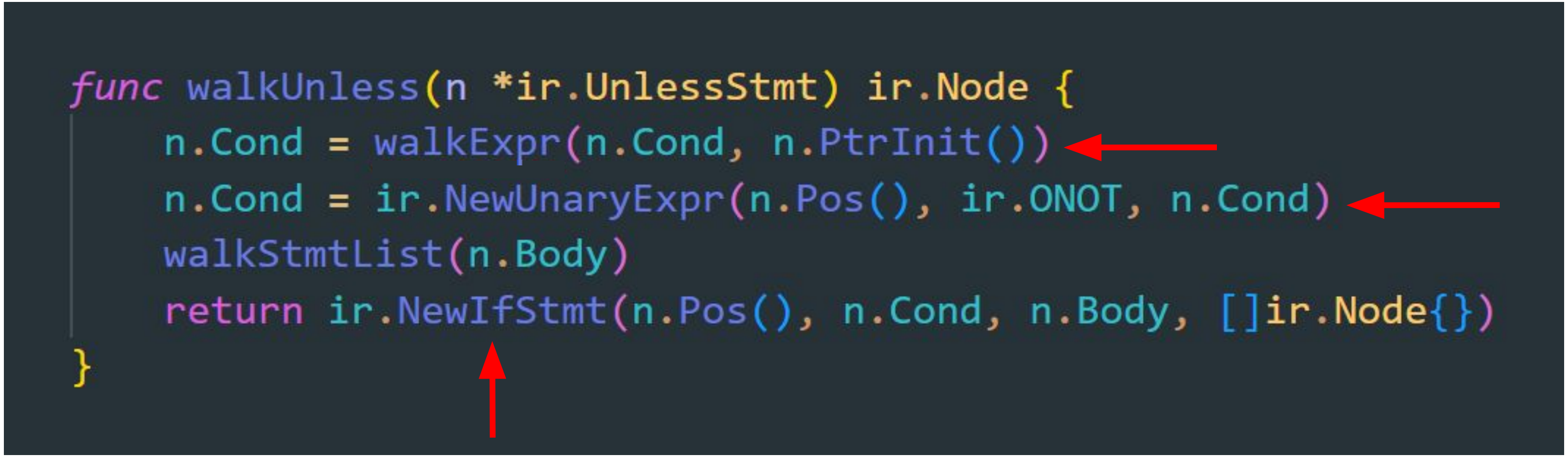
```
func walkUnless(n *ir.UnlessStmt) ir.Node {
    n.Cond = walkExpr(n.Cond, n.PtrInit())
    n.Cond = ir.NewUnaryExpr(n.Pos(), ir.ONOT, n.Cond)
    walkStmtList(n.Body)
    return ir.NewIfStmt(n.Pos(), n.Cond, n.Body, []ir.Node{})
}
```

Walk



```go
func walkUnless(n *ir.UnlessStmt) ir.Node {
	n.Cond = walkExpr(n.Cond, n.PtrInit())
	n.Cond = ir.NewUnaryExpr(n.Pos(), ir.ONOT, n.Cond)
	walkStmtList(n.Body)
	return ir.NewIfStmt(n.Pos(), n.Cond, n.Body, []ir.Node{})
}
```

Walk



```
func walkUnless(n *ir.UnlessStmt) ir.Node {
	n.Cond = walkExpr(n.Cond, n.PtrInit())
	n.Cond = ir.NewUnaryExpr(n.Pos(), ir.ONOT, n.Cond)
	walkStmtList(n.Body)
	return ir.NewIfStmt(n.Pos(), n.Cond, n.Body, []ir.Node{})
}
```

SSA Generation

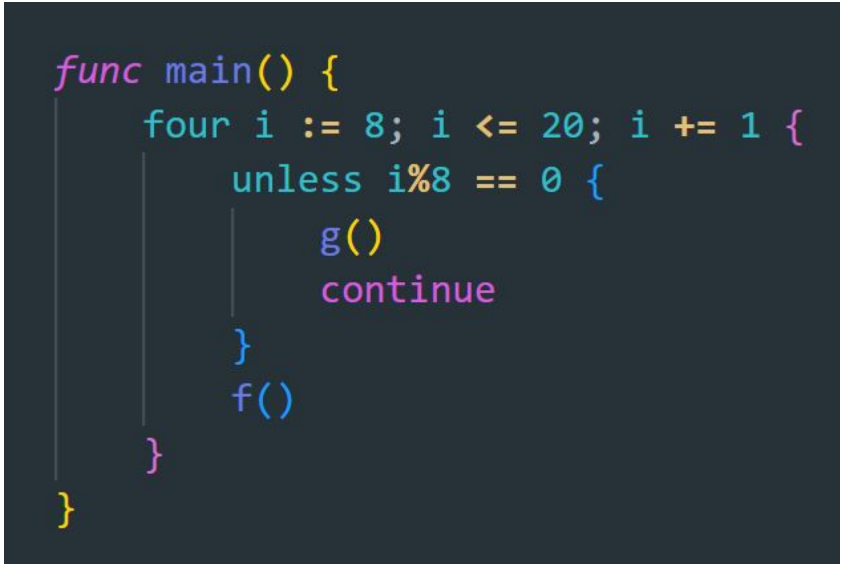
The abstract syntax tree is converted to an IR in Static Single Assignment form.

Static Single Assignment: Program is split up into blocks where each variable is assigned only once.

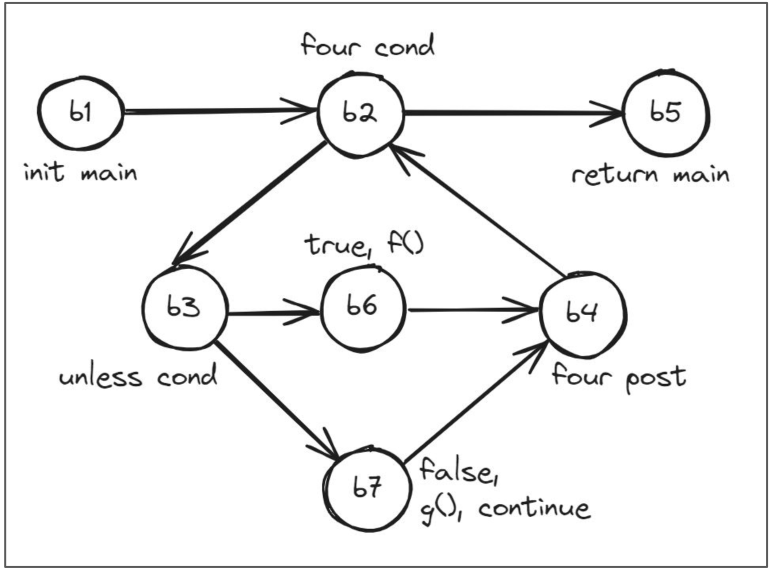
A series of machine independent optimization passes are run on the SSA IR, e.g.

- Removing unused branches
- Removing unneeded nil checks

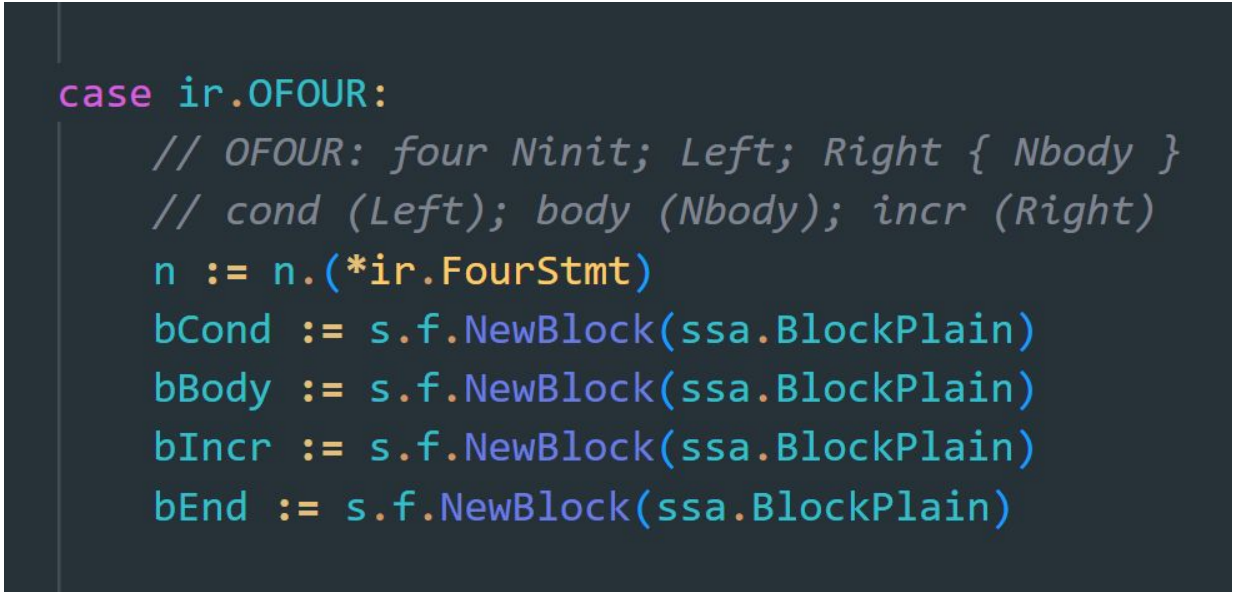
SSA Control Flow Graph



```
func main() {
    four i := 8; i <= 20; i += 1 {
        unless i%8 == 0 {
            g()
            continue
        }
        f()
    }
}
```

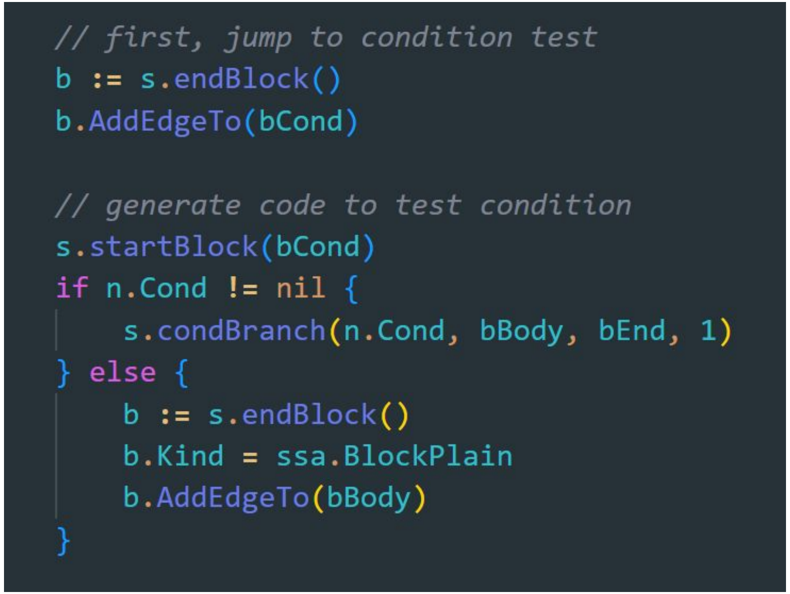


SSA Generation



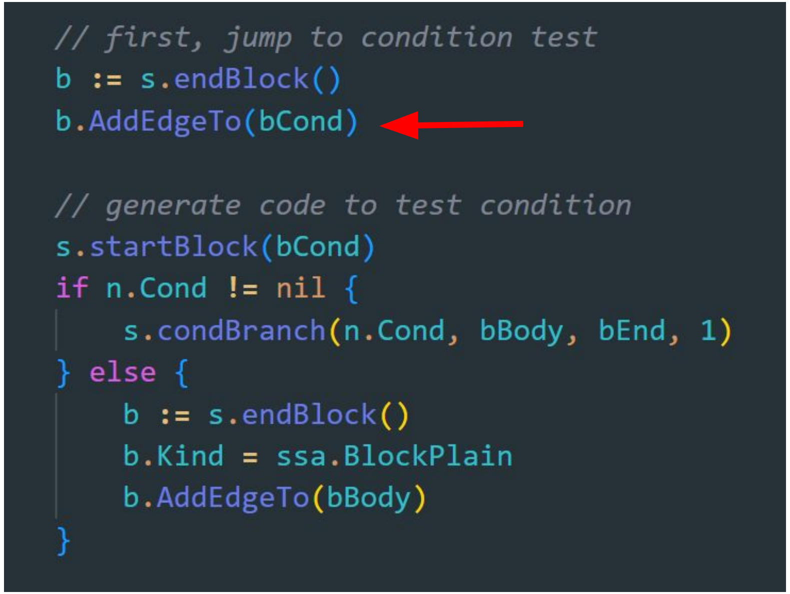
```
case ir.OFOUR:
    // OFOUR: four Ninit; Left; Right { Nbody }
    // cond (Left); body (Nbody); incr (Right)
    n := n.(*ir.FourStmt)
    bCond := s.f.NewBlock(ssa.BlockPlain)
    bBody := s.f.NewBlock(ssa.BlockPlain)
    bIncr := s.f.NewBlock(ssa.BlockPlain)
    bEnd := s.f.NewBlock(ssa.BlockPlain)
```

SSA Generation

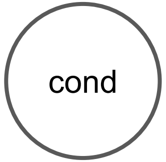


```
// first, jump to condition test
b := s.endBlock()
b.AddEdgeTo(bCond)

// generate code to test condition
s.startBlock(bCond)
if n.Cond != nil {
    s.condBranch(n.Cond, bBody, bEnd, 1)
} else {
    b := s.endBlock()
    b.Kind = ssa.BlockPlain
    b.AddEdgeTo(bBody)
}
```

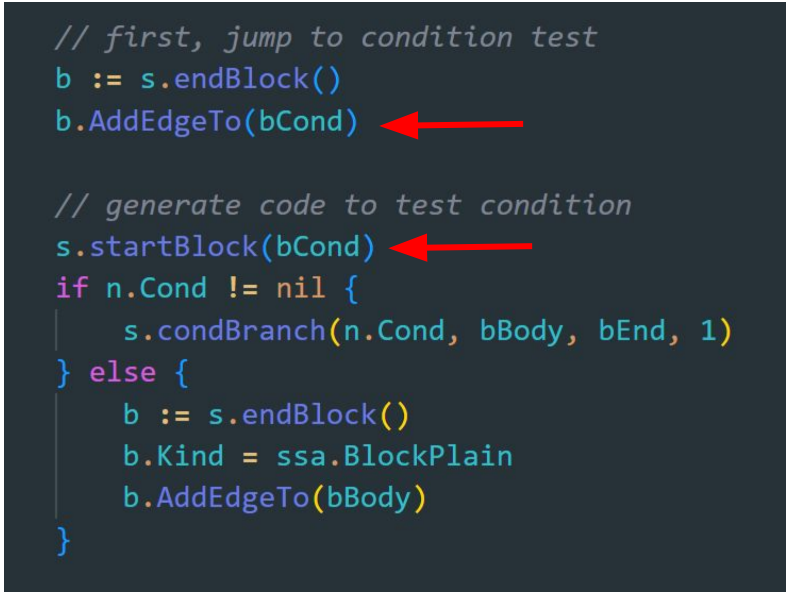
SSA Generation



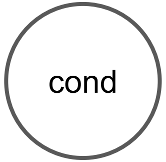
```
// first, jump to condition test
b := s.endBlock()
b.AddEdgeTo(bCond)

// generate code to test condition
s.startBlock(bCond)
if n.Cond != nil {
    s.condBranch(n.Cond, bBody, bEnd, 1)
} else {
    b := s.endBlock()
    b.Kind = ssa.BlockPlain
    b.AddEdgeTo(bBody)
}
```



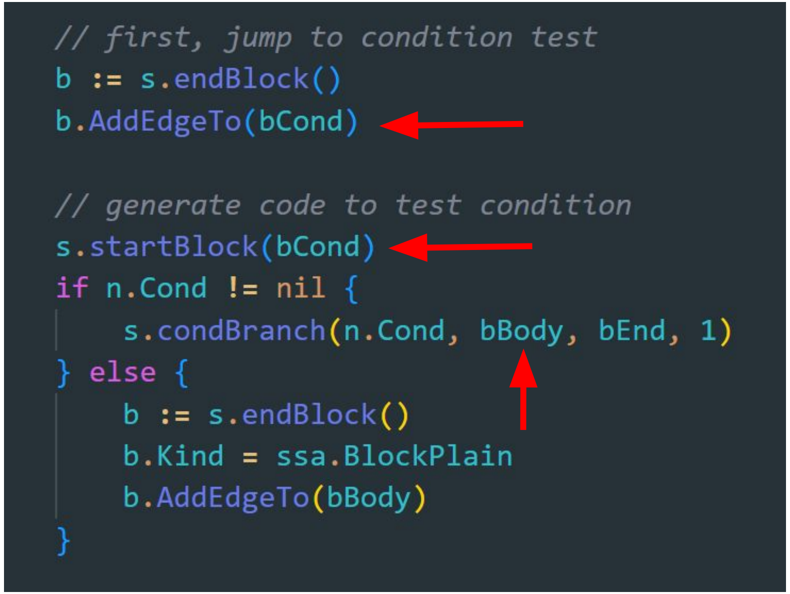
SSA Generation



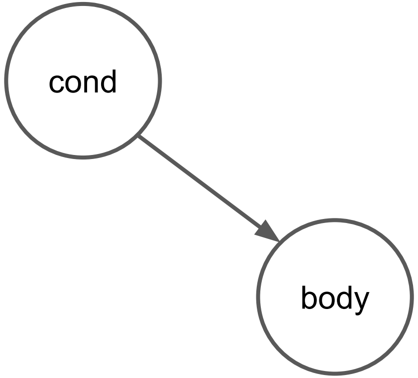
```
// first, jump to condition test
b := s.endBlock()
b.AddEdgeTo(bCond)

// generate code to test condition
s.startBlock(bCond)
if n.Cond != nil {
    s.condBranch(n.Cond, bBody, bEnd, 1)
} else {
    b := s.endBlock()
    b.Kind = ssa.BlockPlain
    b.AddEdgeTo(bBody)
}
```



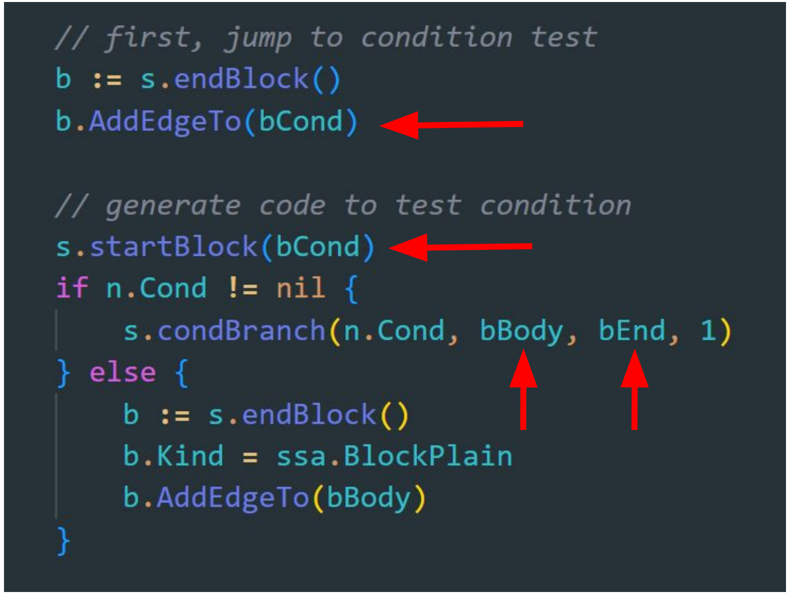
SSA Generation



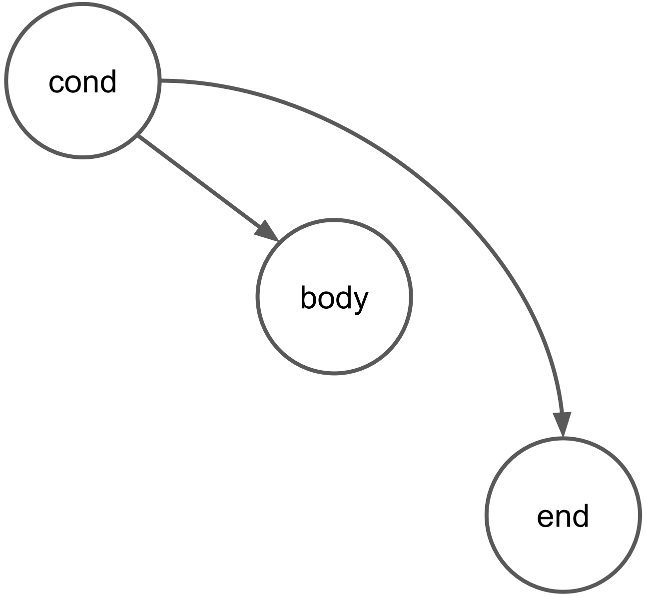
```
// first, jump to condition test
b := s.endBlock()
b.AddEdgeTo(bCond)

// generate code to test condition
s.startBlock(bCond)
if n.Cond != nil {
    s.condBranch(n.Cond, bBody, bEnd, 1)
} else {
    b := s.endBlock()
    b.Kind = ssa.BlockPlain
    b.AddEdgeTo(bBody)
}
```



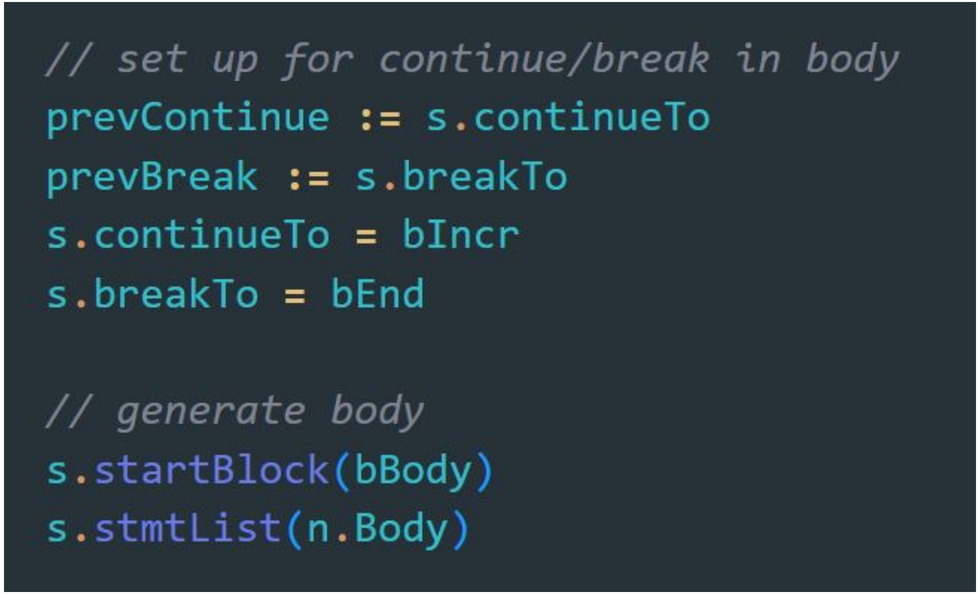
SSA Generation



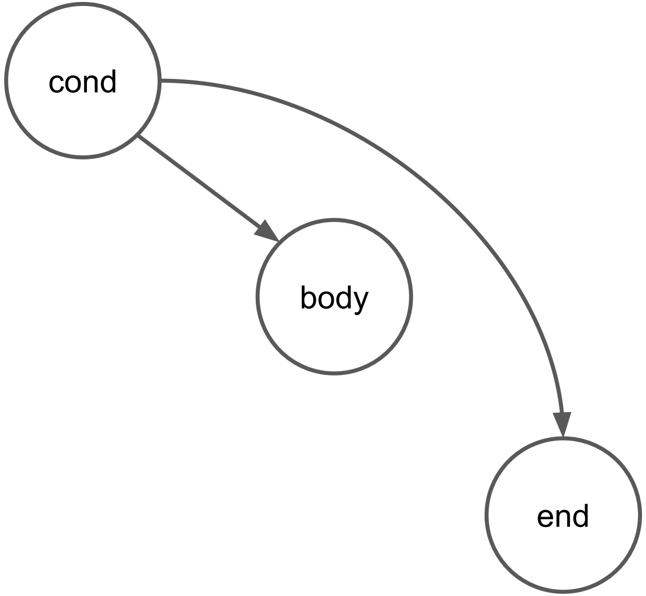
```
// first, jump to condition test
b := s.endBlock()
b.AddEdgeTo(bCond)

// generate code to test condition
s.startBlock(bCond)
if n.Cond != nil {
    s.condBranch(n.Cond, bBody, bEnd, 1)
} else {
    b := s.endBlock()
    b.Kind = ssa.BlockPlain
    b.AddEdgeTo(bBody)
}
```



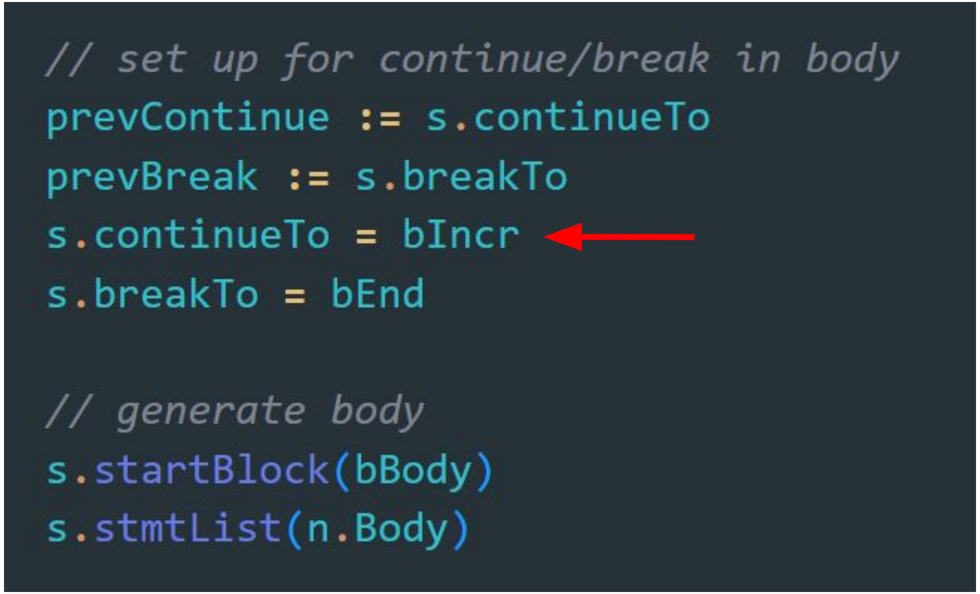
SSA Generation



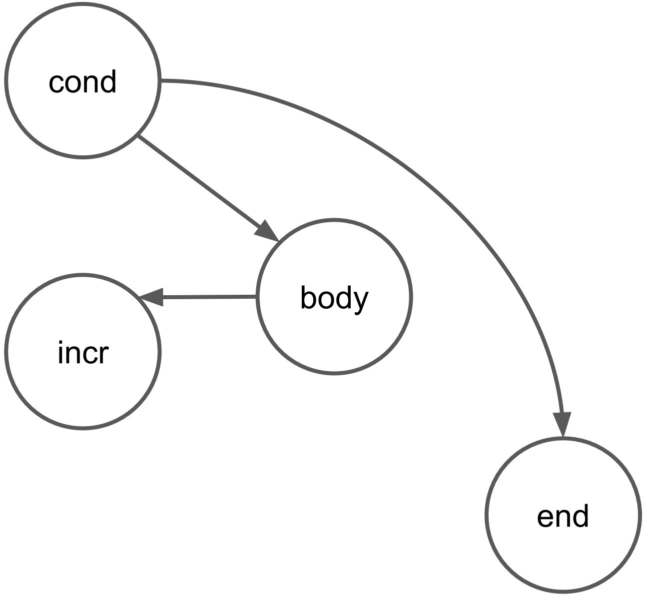
```
// set up for continue/break in body
prevContinue := s.continueTo
prevBreak := s.breakTo
s.continueTo = bIncr
s.breakTo = bEnd

// generate body
s.startBlock(bBody)
s.stmtList(n.Body)
```



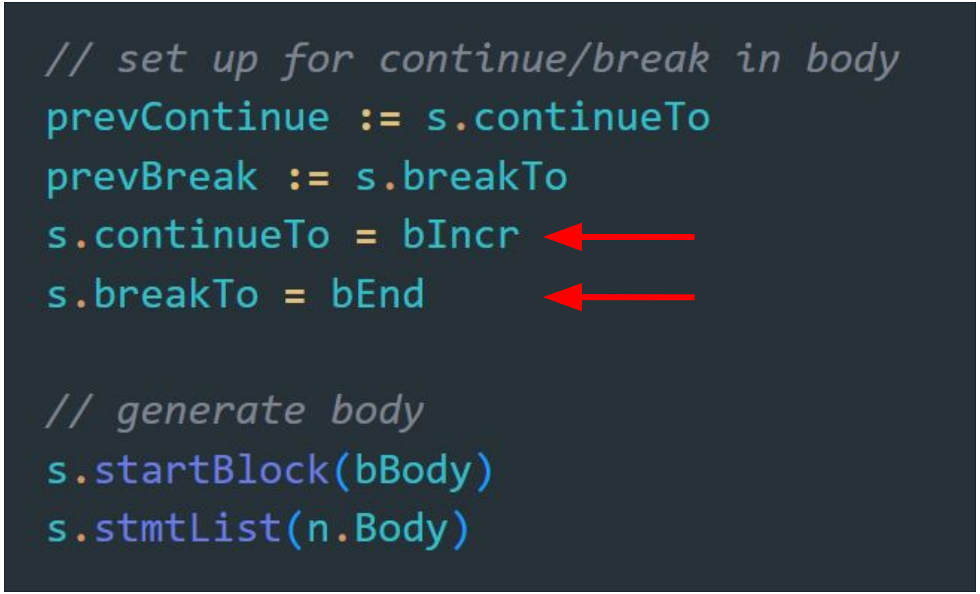
SSA Generation



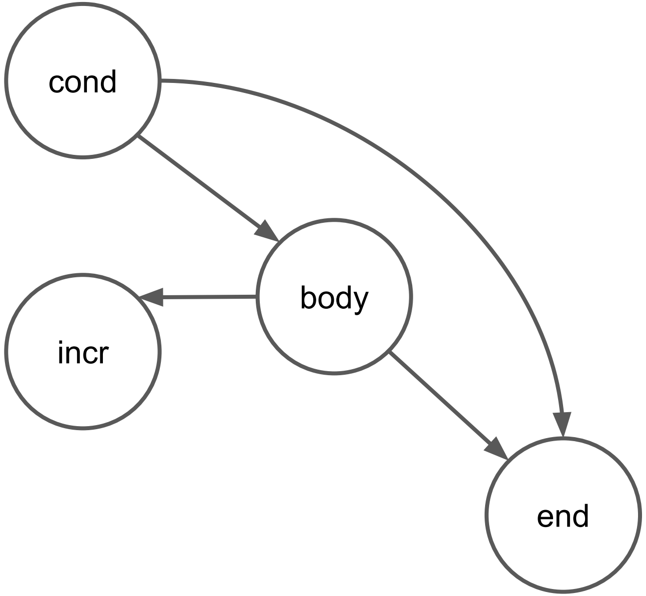
```
// set up for continue/break in body
prevContinue := s.continueTo
prevBreak := s.breakTo
s.continueTo = bIncr
s.breakTo = bEnd

// generate body
s.startBlock(bBody)
s.stmtList(n.Body)
```



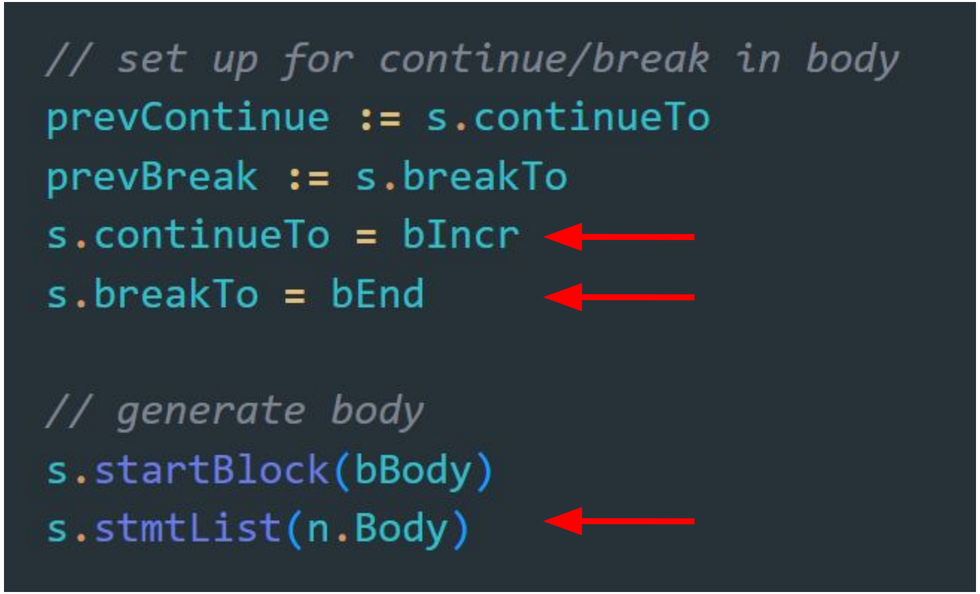
SSA Generation



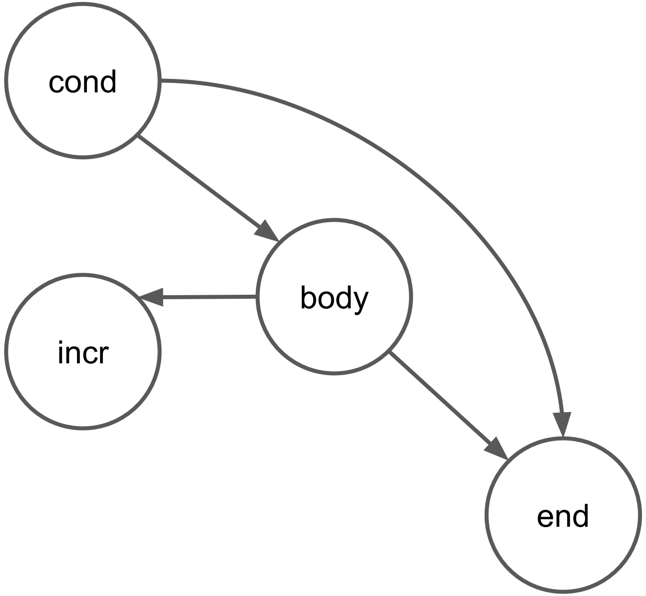
```
// set up for continue/break in body
prevContinue := s.continueTo
prevBreak := s.breakTo
s.continueTo = bIncr
s.breakTo = bEnd

// generate body
s.startBlock(bBody)
s.stmtList(n.Body)
```



SSA Generation

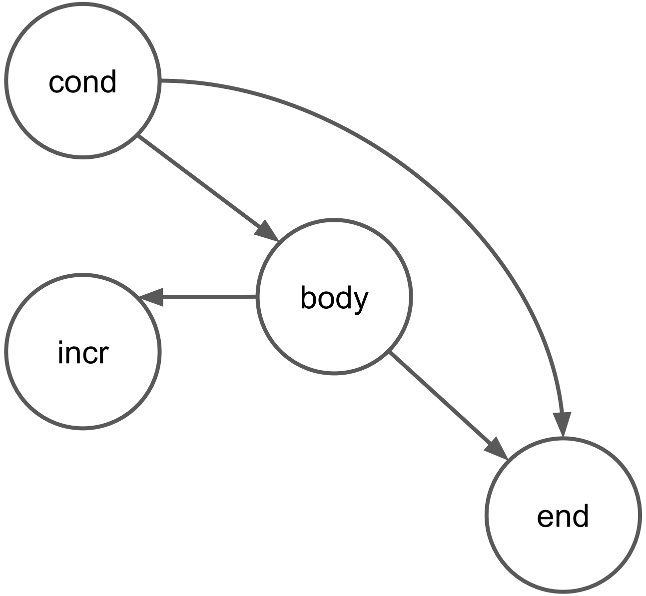


```
// set up for continue/break in body
prevContinue := s.continueTo
prevBreak := s.breakTo
s.continueTo = bIncr
s.breakTo = bEnd

// generate body
s.startBlock(bBody)
s.stmtList(n.Body)
```

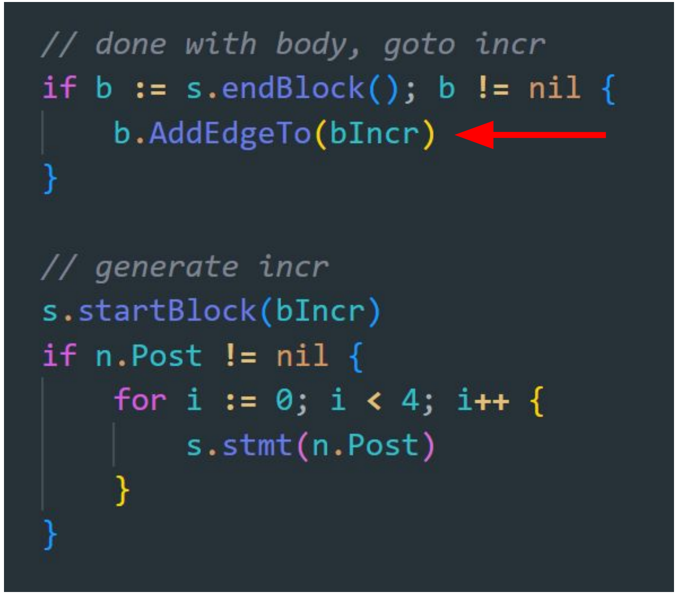


SSA Generation

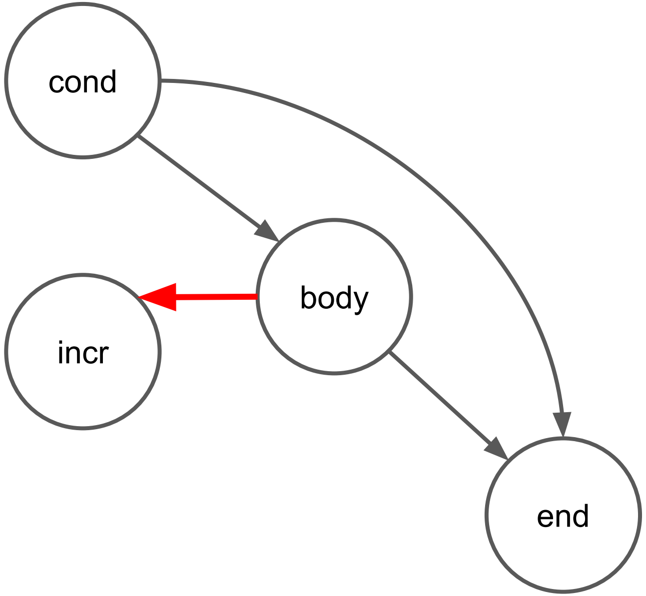
```
// done with body, goto incr
if b := s.endBlock(); b != nil {
    b.AddEdgeTo(bIncr)
}

// generate incr
s.startBlock(bIncr)
if n.Post != nil {
    for i := 0; i < 4; i++ {
        s.stmt(n.Post)
    }
}
```



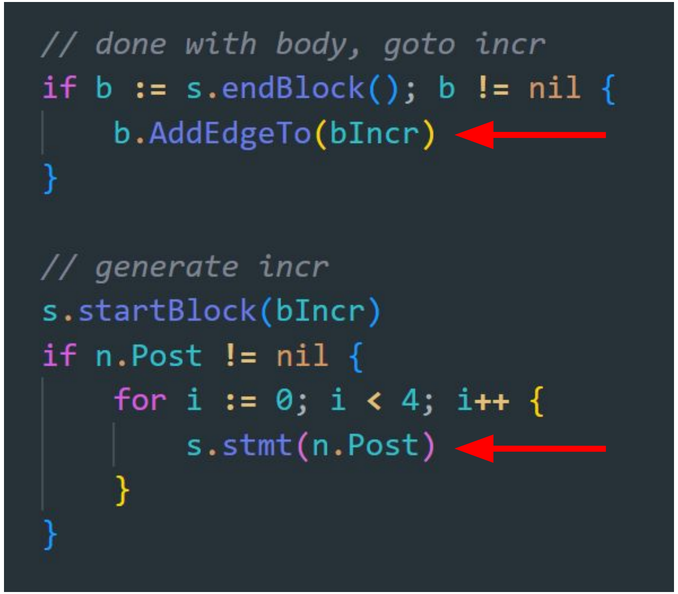
SSA Generation



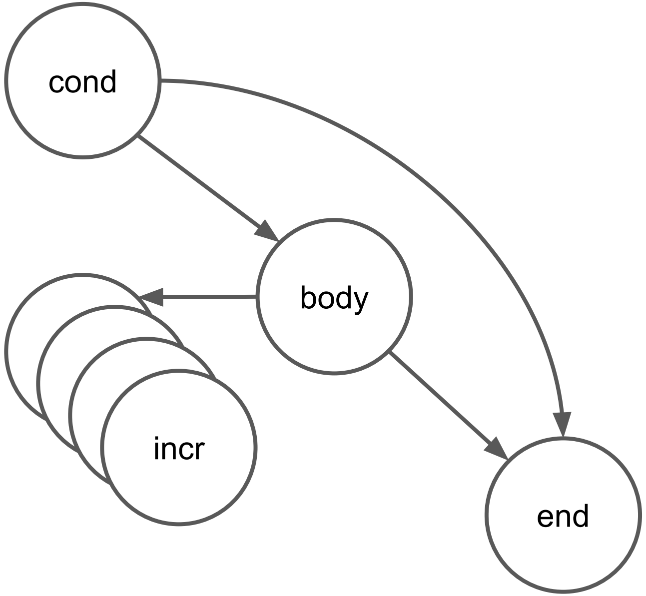
```
// done with body, goto incr
if b := s.endBlock(); b != nil {
    b.AddEdgeTo(bIncr)
}

// generate incr
s.startBlock(bIncr)
if n.Post != nil {
    for i := 0; i < 4; i++ {
        s.stmt(n.Post)
    }
}
```



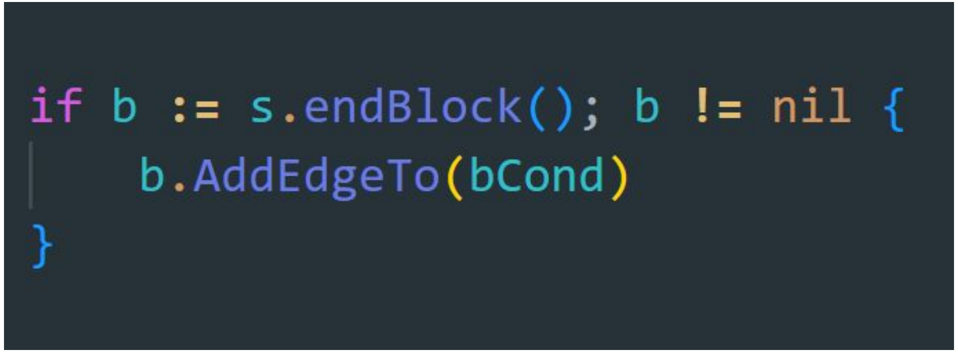
SSA Generation



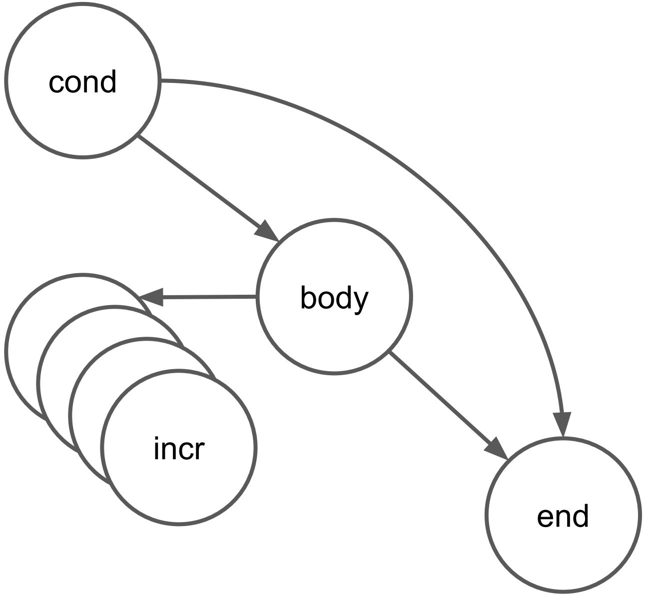
```
// done with body, goto incr
if b := s.endBlock(); b != nil {
    b.AddEdgeTo(bIncr)
}

// generate incr
s.startBlock(bIncr)
if n.Post != nil {
    for i := 0; i < 4; i++ {
        s.stmt(n.Post)
    }
}
```



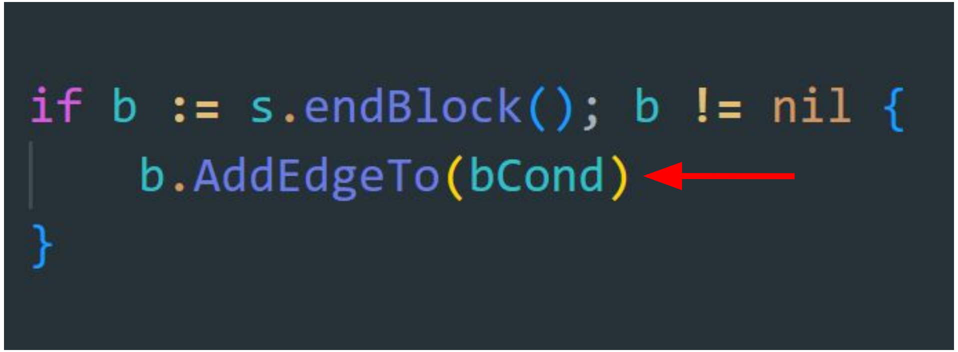
SSA Generation



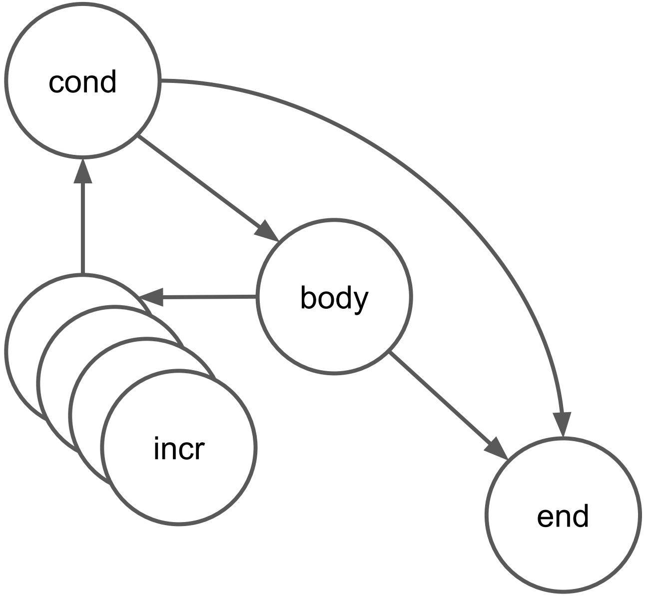
```
if b := s.endBlock(); b != nil {
    b.AddEdgeTo(bCond)
}
```



SSA Generation



```
if b := s.endBlock(); b != nil {
    b.AddEdgeTo(bCond)
}
```



Machine Code Generation

The SSA IR is then “lowered”, rewriting generic values to machine specific ones.

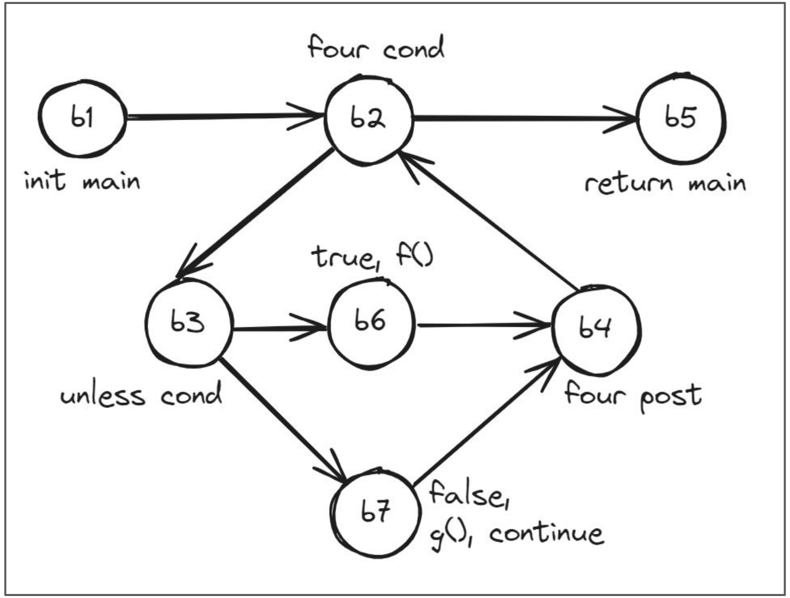
A series of machine-dependent optimization passes are run, e.g.

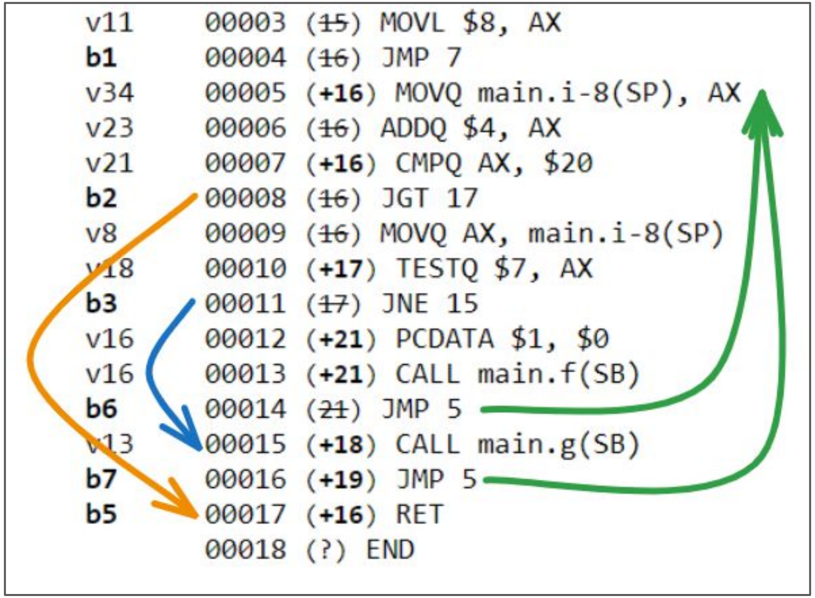
- moving values closer to their uses
- register allocations

The final output is passed to the assembler to generate the actual machine code which will be linked and then can be executed.

Compiler Tooling

```
$ GOSSAFUNC=main go build -gcflags=-l main.go
```



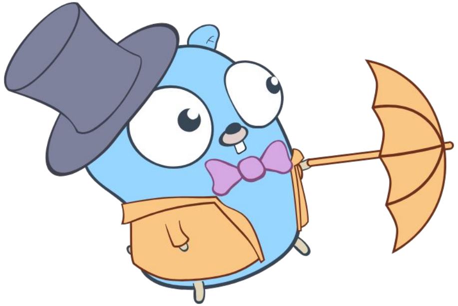


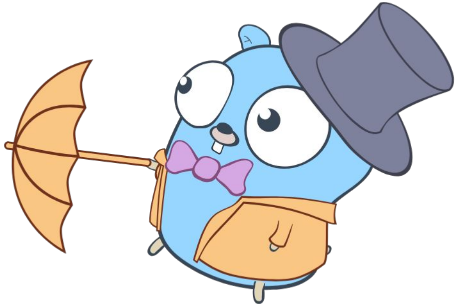
```
v11    00003 (~~15~~) MOVL $8, AX
b1     00004 (~~16~~) JMP 7
v34    00005 (+16) MOVQ main.i-8(SP), AX
v23    00006 (~~16~~) ADDQ $4, AX
v21    00007 (+16) CMPQ AX, $20
b2     00008 (~~16~~) JGT 17
v8     00009 (~~16~~) MOVQ AX, main.i-8(SP)
v18    00010 (+17) TESTQ $7, AX
b3     00011 (~~17~~) JNE 15
v16    00012 (+21) PCDATA $1, $0
v16    00013 (+21) CALL main.f(SB)
b6     00014 (~~21~~) JMP 5
v13    00015 (+18) CALL main.g(SB)
b7     00016 (+19) JMP 5
b5     00017 (+16) RET
       00018 (?) END
```

Summary

- Compilation can be broken down into 3 stages, each with a different representation
- Implement changes in the middle end by desugaring complex statements into simpler ones
- Implement changes in the backend by generating SSA code
- How to analyze the compilation process and output with GOSSAFUNC

Thanks for listening!





Scan to see all of the code