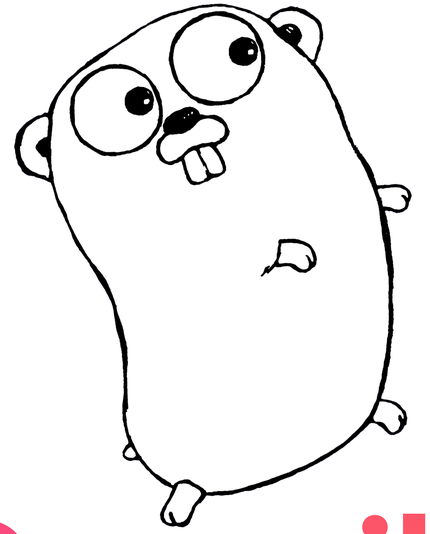


A Look Inside the Go Compiler

... what the hell is actually happening under the hood?

The (Standard) Compiler Pipeline

What's the point?

Compiler's goal is to...

transform source language into *semantically equivalent* target language.

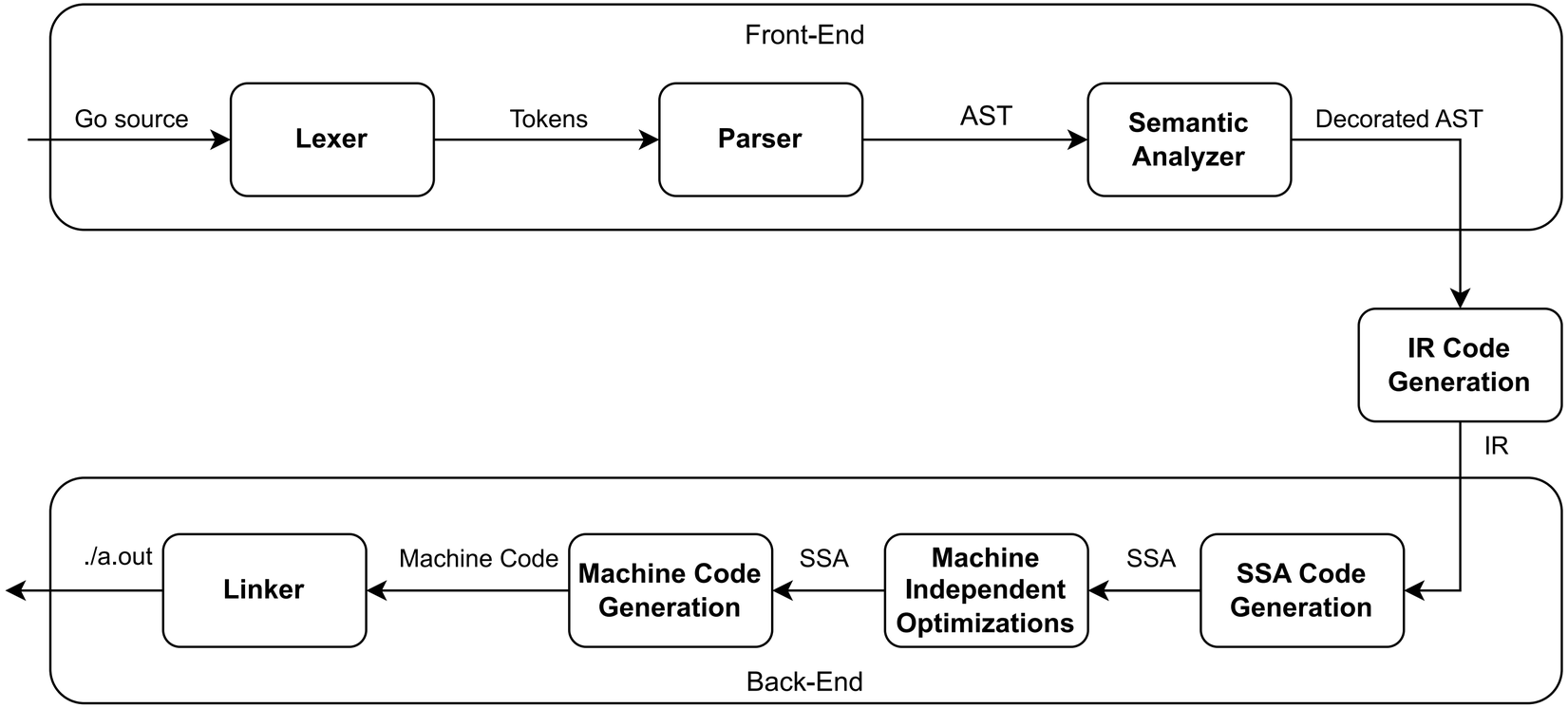
In the case of Go compiler...

source language is the Go source code,

target language is the assembly.

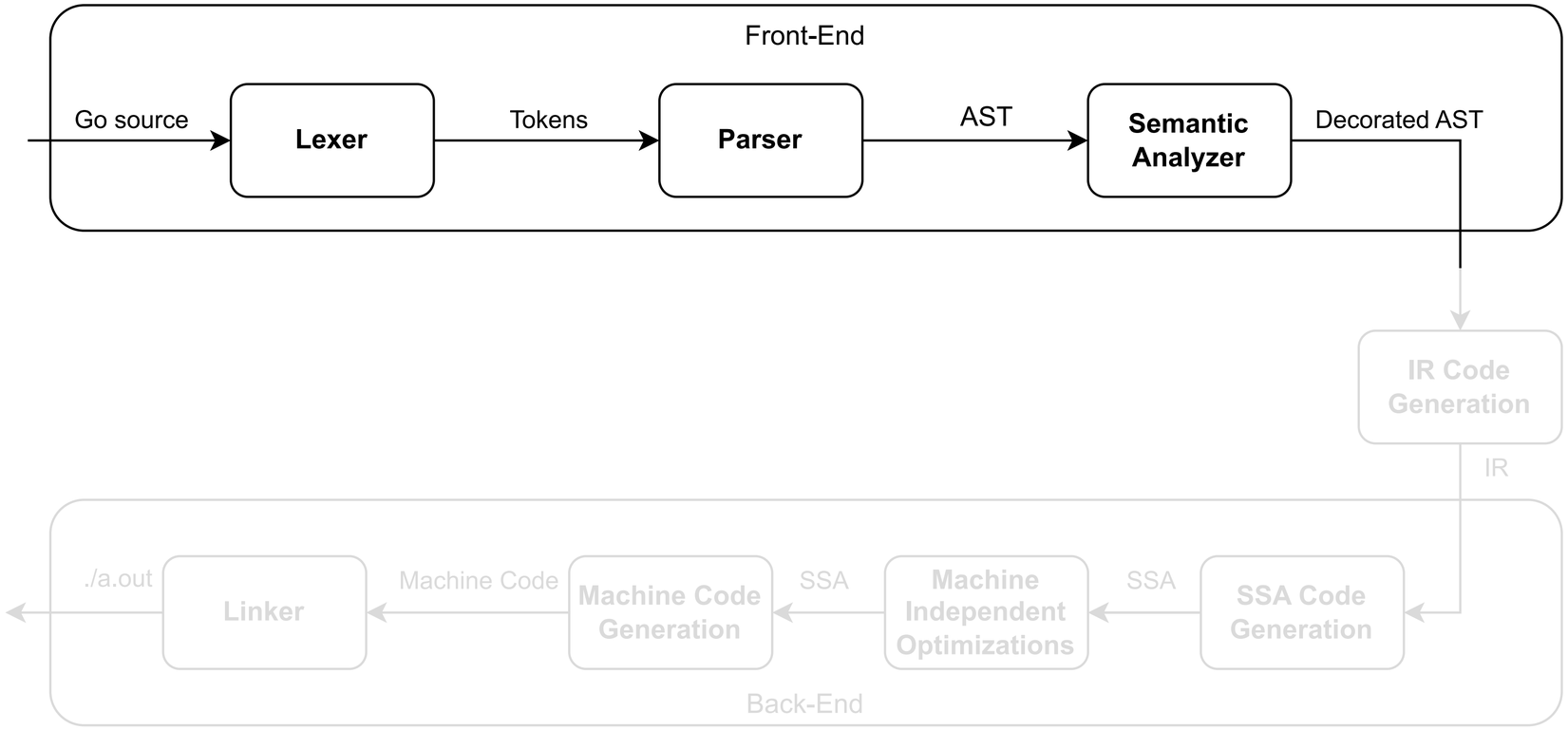
The (Standard) Compiler Pipeline

High-Level Overview



Frontend

Ensures your code makes sense. Then molds it into backend-ready form.



Frontend

Lexing (aka tokenizing aka scanning)

Scans the code, slices it into stream of *tokens*, and keeps track of where everything came from.

- Token is a smallest meaningful unit in the source code.
- It has a *type*, *value*, and *position info*.

Frontend

Token Definiton

```go
// go/src/go/token/token.go

// Token is the set of lexical tokens of the Go programming language.
type Token int

// The list of tokens.
const (
    // Special tokens
    ILLEGAL Token = iota
    EOF
    COMMENT

    literal_beg
    // Identifiers and basic type literals
    // (these tokens stand for classes of literals)
    IDENT  // main
    INT    // 12345
    FLOAT  // 123.45
    IMAG   // 123.45i
    CHAR   // 'a'
    STRING // "abc"
    literal_end
```

```go
// go/src/go/token/position.go

// Position describes an arbitrary source position
// including the file, line, and column location.
// A Position is valid if the line number is > 0.
type Position struct {
    Filename string // filename, if any
    Offset   int    // offset, starting at 0
    Line     int    // line number, starting at 1
    Column   int    // column number, starting at 1 (byte count)
}
```

```go
// go/src/go/scanner/scanner.go

func (s *Scanner) Scan() (pos token.Pos, tok token.Token, lit string)
```

Frontend

Example: Scanning

```
1  package main
2
3  const ANSWER = 42
4
5  // Greet the user depending on the parameter `x`.
6  func greet(x int) bool {
7      // Greet the user only if the argument is
8      // the answer to everything.
9      if x == ANSWER {
10         println("hello, gophercamp!")
11         return true
12     }
13
14     return false
15 }
16
17 func main() {
18     greet(42)
19 }
20
```


```
("main.go:1:1", PACKAGE, "package")
```

Frontend

Example: Scanning

```
1 package main
2
3 const ANSWER = 42
4
5 // Greet the user depending on the parameter `x`.
6 func greet(x int) bool {
7     // Greet the user only if the argument is
8     // the answer to everything.
9     if x == ANSWER {
10        println("hello, gophercamp!")
11        return true
12    }
13
14    return false
15 }
16
17 func main() {
18    greet(42)
19 }
20
```

Frontend

Example: Scanning

```
1  package main
2
3  const ANSWER = 42
4
5  // Greet the user depending on the parameter `x`.
6  func greet(x int) bool {
7      // Greet the user only if the argument is
8      // the answer to everything.
9      if x == ANSWER {
10         println("hello, gophercamp!")
11         return true
12     }
13
14     return false
15 }
16
17 func main() {
18     greet(42)
19 }
20
```

Frontend

Example: Scanning

```
1 package main
2
3 const ANSWER = 42
4
5 // Greet the user depending on the parameter `x`.
6 func greet(x int) bool {
7     // Greet the user only if the argument is
8     // the answer to everything.
9     if x == ANSWER {
10        println("hello, gophercamp!")
11        return true
12    }
13
14    return false
15 }
16
17 func main() {
18    greet(42)
19 }
20
```

s.Scan() → ("main.go:3:1", CONST, "const")

Frontend

Example: Scanning

```
1  package main
2
3  const ANSWER = 42
4
5  // Greet the user depending on the parameter `x`.
6  func greet(x int) bool {
7      // Greet the user only if the argument is
8      // the answer to everything.
9      if x == ANSWER {
10         println("hello, gophercamp!")
11         return true
12     }
13
14     return false
15 }
16
17 func main() {
18     greet(42)
19 }
20
```

Frontend

Example: Scanning

```
1 package main
2
3 const ANSWER = 42
4
5 // Greet the user depending on the parameter `x`.
6 func greet(x int) bool {
7     // Greet the user only if the argument is
8     // the answer to everything.
9     if x == ANSWER {
10        println("hello, gophercamp!")
11        return true
12    }
13
14    return false
15 }
16
17 func main() {
18    greet(42)
19 }
20
```


```
("main.go:3:14", ASSIGN, "=")
```

Frontend

Example: Scanning

```
1  package main
2
3  const ANSWER = 42
4
5  // Greet the user depending on the parameter `x`.
6  func greet(x int) bool {
7      // Greet the user only if the argument is
8      // the answer to everything.
9      if x == ANSWER {
10         println("hello, gophercamp!")
11         return true
12     }
13
14     return false
15 }
16
17 func main() {
18     greet(42)
19 }
20
```

Frontend

Parser

Builds structure from tokens, turning flat code into an *abstract syntax tree* according to the grammar.

- Hand-written recursive descent parser with limited lookahead.
- Parser closely mirrors the formal grammar of the language.

Frontend

Sample Parser Code

Function declarations

A function declaration binds an identifier, the *function name*, to a function.

```
FunctionDecl = "func" FunctionName [ TypeParameters ] Signature [ FunctionBody ] .
FunctionName = identifier .
FunctionBody = Block .
```

https://go.dev/ref/spec#Function_declarations

```go
// go/src/go/parser/parser.go

func (p *parser) parseFuncDecl() *ast.FuncDecl {
    // ...
    pos := p.expect(token.FUNC)
    // ...
    ident := p.parseIdent()
    // ...
    params := p.parseParameters(false)
    results := p.parseParameters(true)

    var body *ast.BlockStmt
    switch p.tok {
    case token.LBRACE:
        body = p.parseBody()
        p.expectSemi()
    case token.SEMICOLON:
        p.next()
        // ...
    default:
        p.expectSemi()
    }

    decl := &ast.FuncDecl{
        Doc:  doc,
        Recv: recv,
        Name: ident,
        Type: &ast.FuncType{
            Func:       pos,
            TypeParams: tparams,
            Params:     params,
            Results:    results,
        },
        Body: body,
    }
    return decl
}
```

Frontend

Parsed AST

```
 1 package main
 2
 3 const ANSWER = 42
 4
 5 // Greet the user depending on the parameter `x`.
 6 func greet(x int) bool {
 7     // Greet the user only if the argument is
 8     // the answer to everything.
 9     if x == ANSWER {
10         println("hello, gophercamp!")
11         return true
12     }
13
14     return false
15 }
16
17 func main() {
18     greet(42)
19 }
```

```
 0  *ast.File {
 1  .  Package: main.go:1:1
 2  .  Name: *ast.Ident {
 3  .  .  NamePos: main.go:1:9
 4  .  .  Name: "main"
 5  .  }
 6  .  Decls: []ast.Decl (len = 3) {
 7  .  .  0: *ast.GenDecl {
 8  .  .  .  TokPos: main.go:3:1
 9  .  .  .  Tok: const
10  .  .  .  Lparen: -
11  .  .  .  Specs: []ast.Spec (len = 1) {
12  .  .  .  .  0: *ast.ValueSpec {
13  .  .  .  .  .  Names: []*ast.Ident (len = 1) {
14  .  .  .  .  .  .  0: *ast.Ident {
15  .  .  .  .  .  .  .  NamePos: main.go:3:7
16  .  .  .  .  .  .  .  Name: "ANSWER"
17  .  .  .  .  .  .  .  Obj: *ast.Object {
18  .  .  .  .  .  .  .  .  Kind: const
19  .  .  .  .  .  .  .  .  Name: "ANSWER"
20  .  .  .  .  .  .  .  .  Decl: *(obj @ 12)
21  .  .  .  .  .  .  .  .  Data: 0
22  .  .  .  .  .  .  .  }
23  .  .  .  .  .  .  }
24  .  .  .  .  .  }
25  .  .  .  .  .  Values: []ast.Expr (len = 1) {
26  .  .  .  .  .  .  0: *ast.BasicLit {
27  .  .  .  .  .  .  .  ValuePos: main.go:3:16
28  .  .  .  .  .  .  .  Kind: INT
29  .  .  .  .  .  .  .  Value: "42"
30  .  .  .  .  .  .  }
31  .  .  .  .  .  }
32  .  .  .  .  }
33  .  .  .  }
34  .  .  .  Rparen: -
35  .  .  }
36  .  .  1: *ast.FuncDecl {
            //  ...
234  }
```

Frontend

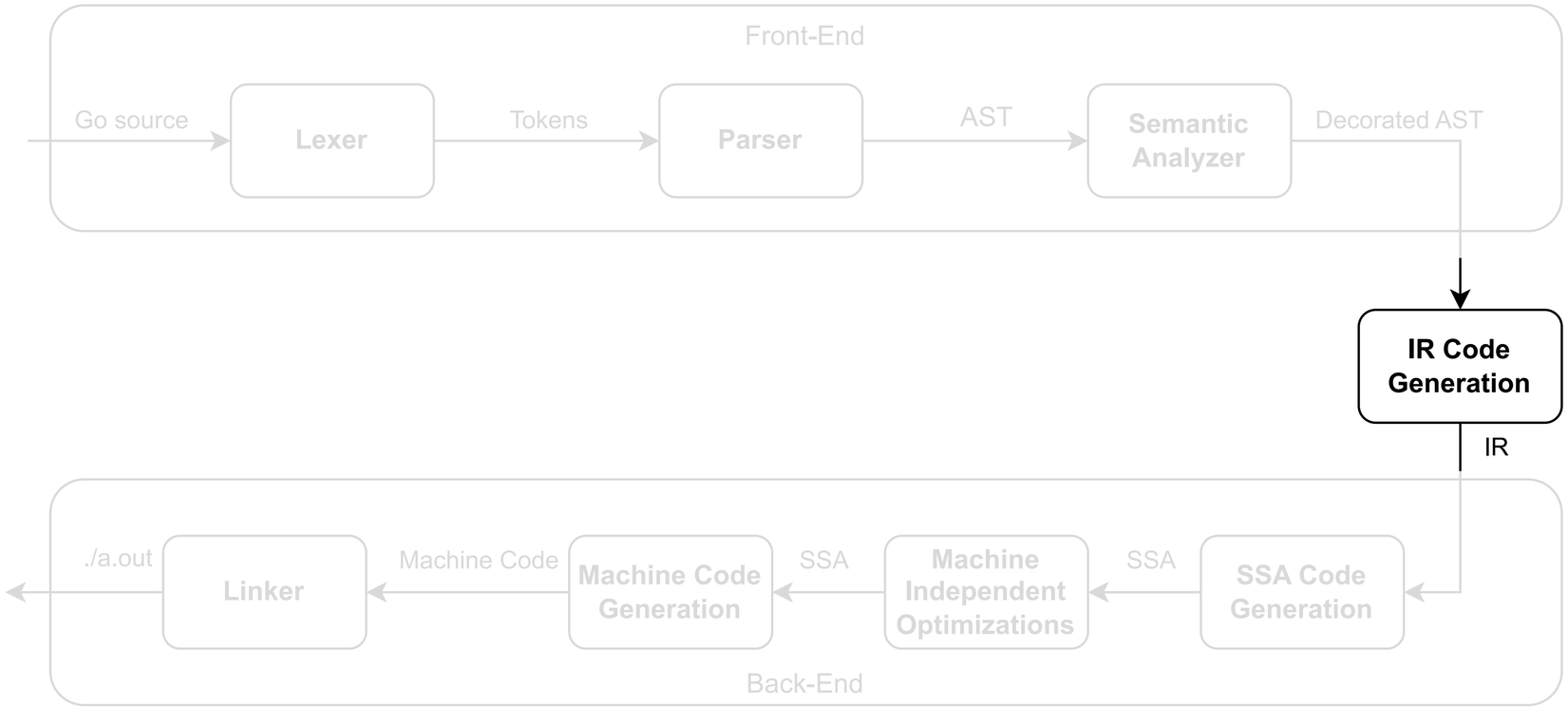
Semantic Analyzer

The code may look right—this phase makes sure it *is* right.

- Type inference, type checking and monomorphization of generics
- Scope, definition, name resolution
- Return statement checking
- noding: lowering to *Intermediate Representation*

Middle-end

Perform Go specific optimizations & desugar Go constructs.



Middle-end

IR Optimization

- Function call inlining
- Devirtualization of known interface method calls
- Escape analysis

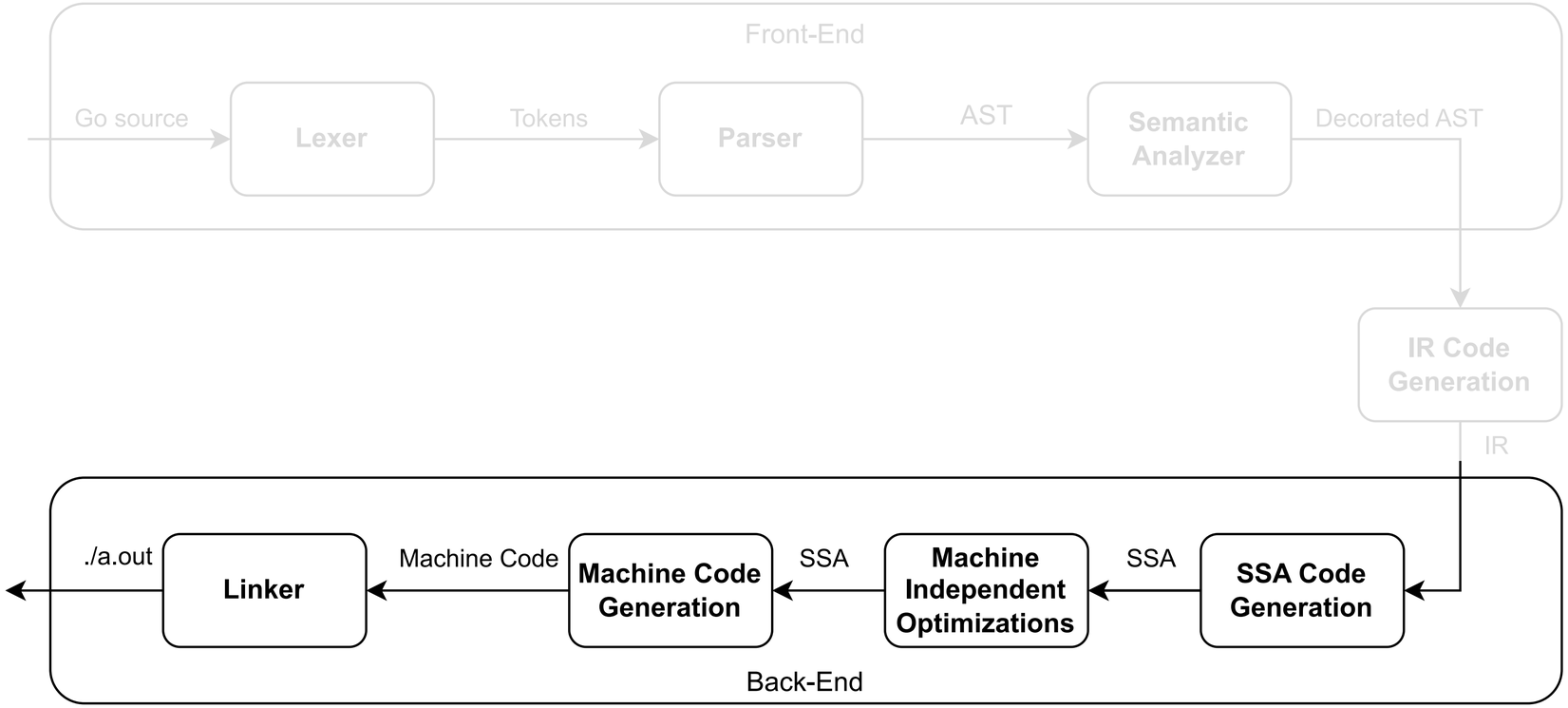
Middle-end

Walking the IR

- Decompose complex statements into simpler ones.
- Desugar higher-level constructs:
  - Switch statements into binary search or jump tables.
  - Operations on maps and channels replaced with runtime code.

Backend

Lowers, optimizes, and assembles your code for the target architecture.



Backend

Why Custom Backend?

"It's a small toolchain that we can keep in our heads and make arbitrary changes to, quickly and easily. Honestly, if we'd built on GCC or LLVM, we'd be moving so slowly I'd probably have left the project years ago.

[...] Certainly there is a room for improvement, but the custom toolchain is one of the key reasons we've accomplished so much in so little time."

- Russ Cox in Hacker News thread[1]

[1]https://news.ycombinator.com/item?id=8817990

Backend

Lowering to Static Single Assignment

Static Single Assignment: IR in which each variable is assigned to *exactly once*.

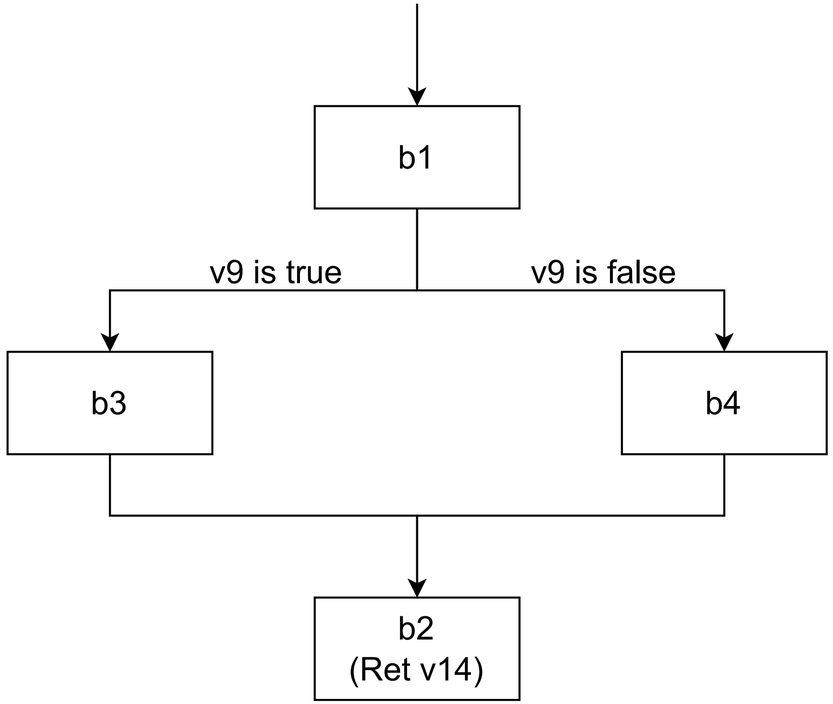
```
1  func example() int {
2      a := 10
3      b := a * 10
4
5      c := 0
6      if b == 100 {
7          c = b
8      } else {
9          c = 42
10     }
11
12     return c
13 }
```

```
1  b1:
2      v1 (?) = InitMem <mem>
3      v2 (?) = SP <uintptr>
4      v3 (?) = SB <uintptr>
5      v4 (?) = LocalAddr <*int> {~r0} v2 v1
6      v5 (?) = Const64 <int> [0] (c[int])
7      v6 (?) = Const64 <int> [10] (a[int])
8      v7 (24) = Mul64 <int> v6 v6 (b[int])
9      v8 (?) = Const64 <int> [100]
10     v9 (27) = Eq64 <bool> v7 v8
11     v11 (?) = Const64 <int> [42] (c[int])
12 If v9 → b3 b4 (27)
13
14 b2: ← b3 b4
15     v12 (33) = Phi <int> v10 v11 (c[int])
16     v13 (33) = Copy <mem> v1
17     v14 (33) = MakeResult <int,mem> v12 v13
18 Ret v14 (+33)
19
20 b3: ← b1
21     v10 (28) = Copy <int> v7 (b[int], c[int])
22
23 Plain → b2 (28)
24
25 b4: ← b1
26
27 Plain → b2 (33)
```

Backend

SSA & Control-Flow Graph

```
b1:
    v1 (?) = InitMem <mem>
    v2 (?) = SP <uintptr>
    v3 (?) = SB <uintptr>
    v4 (?) = LocalAddr <*int> {~r0} v2 v1
    v5 (?) = Const64 <int> [0] (c[int])
    v6 (?) = Const64 <int> [10] (a[int])
    v7 (24) = Mul64 <int> v6 v6 (b[int])
    v8 (?) = Const64 <int> [100]
    v9 (27) = Eq64 <bool> v7 v8
    v11 (?) = Const64 <int> [42] (c[int])
If v9 → b3 b4 (27)

b2: ← b3 b4
    v12 (33) = Phi <int> v10 v11 (c[int])
    v13 (33) = Copy <mem> v1
    v14 (33) = MakeResult <int,mem> v12 v13
Ret v14 (+33)

b3: ← b1
    v10 (28) = Copy <int> v7 (b[int], c[int])

Plain → b2 (28)

b4: ← b1

Plain → b2 (33)
```



Backend

Machine-Independent Optimization

- Optimizations performed in multiple (ordered) passes
  - Dead code elimination
  - Removal of unneeded nil checks
  - Constant folding
  - Common sub-expression elimination
- Application of function intrinsics

Backend

Definition of Passes

```go
// src/cmd/compile/internal/ssa/compile.go

// list of passes for the compiler
var passes = [...]pass{
    {name: "number lines", fn: numberLines, required: true},
    {name: "early phielim and copyelim", fn: copyelim},
    {name: "early deadcode", fn: deadcode},
    {name: "short circuit", fn: shortcircuit},
    {name: "decompose user", fn: decomposeUser, required: true},
```

```go
// src/cmd/compile/internal/ssa/compile.go

// Double-check phase ordering constraints.
// This code is intended to document the ordering requirements
// between different phases. It does not override the passes
// list above.

type constraint struct {
    a, b string // a must come before b
}

var passOrder = [...]constraint{
    {"dse", "insert resched checks"},
    {"insert resched checks", "lower"},
    {"insert resched checks", "tighten"},
```

Backend

Definition of Passes via Rewrite Rules

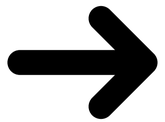
```
1 // src/cmd/compile/internal/ssa/_gen/generic.rules
2
3 // constant folding
4 (Add8   (Const8 [c])   (Const8 [d]))   ⇒ (Const8  [c+d])
5
6 // Convert x * 1 to x.
7 (Mul(8|16|32|64)  (Const(8|16|32|64)  [1]) x) ⇒ x
8
9 // constant floating point comparisons
10 (Eq32F   (Const32F [c]) (Const32F [d])) ⇒ (ConstBool [c == d])
```

Backend

Optimization of SSA

```
1  func example() int {
2      a := 10
3      b := a * 10
4
5      c := 0
6      if b == 100 {
7          c = b
8      } else {
9          c = 42
10     }
11
12     return c
13 }
```

```
1  b1:
2      v1 (?) = InitMem <mem>
3      v2 (?) = SP <uintptr>
4      v3 (?) = SB <uintptr>
5      v4 (?) = LocalAddr <*int> {~r0} v2 v1
6      v5 (?) = Const64 <int> [0] (c[int])
7      v6 (?) = Const64 <int> [10] (a[int])
8      v7 (24) = Mul64 <int> v6 v6 (b[int])
9      v8 (?) = Const64 <int> [100]
10     v9 (27) = Eq64 <bool> v7 v8
11     v11 (?) = Const64 <int> [42] (c[int])
12 If v9 → b3 b4 (27)
13
14 b2: ← b3 b4
15     v12 (33) = Phi <int> v10 v11 (c[int])
16     v13 (33) = Copy <mem> v1
17     v14 (33) = MakeResult <int,mem> v12 v13
18 Ret v14 (+33)
19
20 b3: ← b1
21     v10 (28) = Copy <int> v7 (b[int], c[int])
22
23 Plain → b2 (28)
24
25 b4: ← b1
26
27 Plain → b2 (33)
```



```
1  b2:
2      v1 (?) = InitMem <mem>
3      v7 (+24) = Const64 <int> [100] (c[int], b[int])
4      v14 (+33) = MakeResult <int,mem> v7 v1
5  Ret v14 (+33)
```

Backend

Machine-Specific Optimizations & Machine Code

After machine-independent opts.:

```
1 b2:
2     v1 (?) = InitMem <mem>
3     v7 (+24) = Const64 <int> [100] (c[int], b[int])
4     v14 (+33) = MakeResult <int,mem> v7 v1
5 Ret v14 (+33)
```

After lowering:

```
1 b2:
2     v1 (?) = InitMem <mem>
3     v7 (+24) = MOVDconst <int> [100] (b[int], c[int])
4     v14 (+33) = MakeResult <int,mem> v7 v1
5 Ret v14 (+33)
```

Final machine code:

```
1      00000 (22) TEXT main.simple(SB), ABIInternal
2      00001 (22) FUNCDATA $0, gclocals·FzY36IO2mY0y4dZ1+Izd/w==(SB)
3      00002 (22) FUNCDATA $1, gclocals·FzY36IO2mY0y4dZ1+Izd/w==(SB)
4 v9   00003 (+33) MOVD $100, R0
5 b2   00004 (33) RET
6      00005 (?) END
```

```
go build -gcflags="that's all!"
```

Helpful Resources

- https://go.dev/src/cmd/compile/README
- https://go.dev/src/cmd/compile/internal/ssa/README
- https://eli.thegreenplace.net/2019/go-compiler-internals-adding-a-new-statement-to-go-part-1/
- https://eli.thegreenplace.net/2019/go-compiler-internals-adding-a-new-statement-to-go-part-2/
- https://www.quasilyte.dev/blog/post/go_ssa_rules/
- https://dave.cheney.net/2019/08/20/go-compiler-intrinsics