
Desmistificando o compilador Go: A jornada do func main() até o go run



Alex Sandro Garzão

TcheLinux POA - Dezembro 2019

Agenda (ou alinhando expectativas)

O que é um compilador

Compilador Go (Histórico, arquitetura, etapas de compilação, ...)

Hacking (Alegrias, tristezas, trechos de código, ...)

Considerações finais

Não dá tempo para tudo....



Quem sou

- Engenheiro de Software na Zenvia
- Minhas paixões (algumas)
 - Linguagens de programação, compiladores, máquinas virtuais
 - Domínios complexos, algoritmos, processamento em tempo real
 - Go, Python, C, C++, ...
 - Falar sobre tecnologia
 - Código bem feito :-)
- Já atuei com “Toy compilers”
 - HoloC, UbiC, G-Portugol, Pascal para bytecode JVM
 - [ST \(IEC 61131-3\)](#) para ASM (80C51), ...

Vamos nos conhecer um pouco...

Quem tem conhecimento sobre como um compilador funciona?

Quem já atuou na implementação de um compilador (toy ou não)?

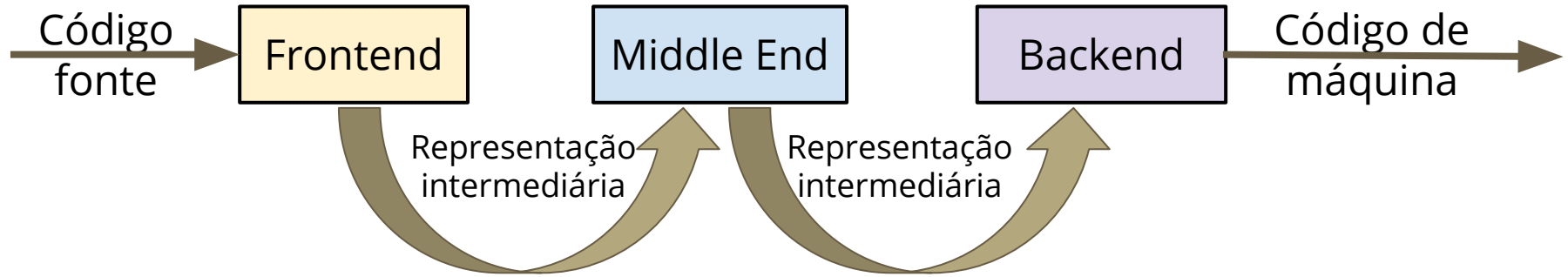
Quem já olhou e/ou analisou o código de um compilador?

O que é um compilador?

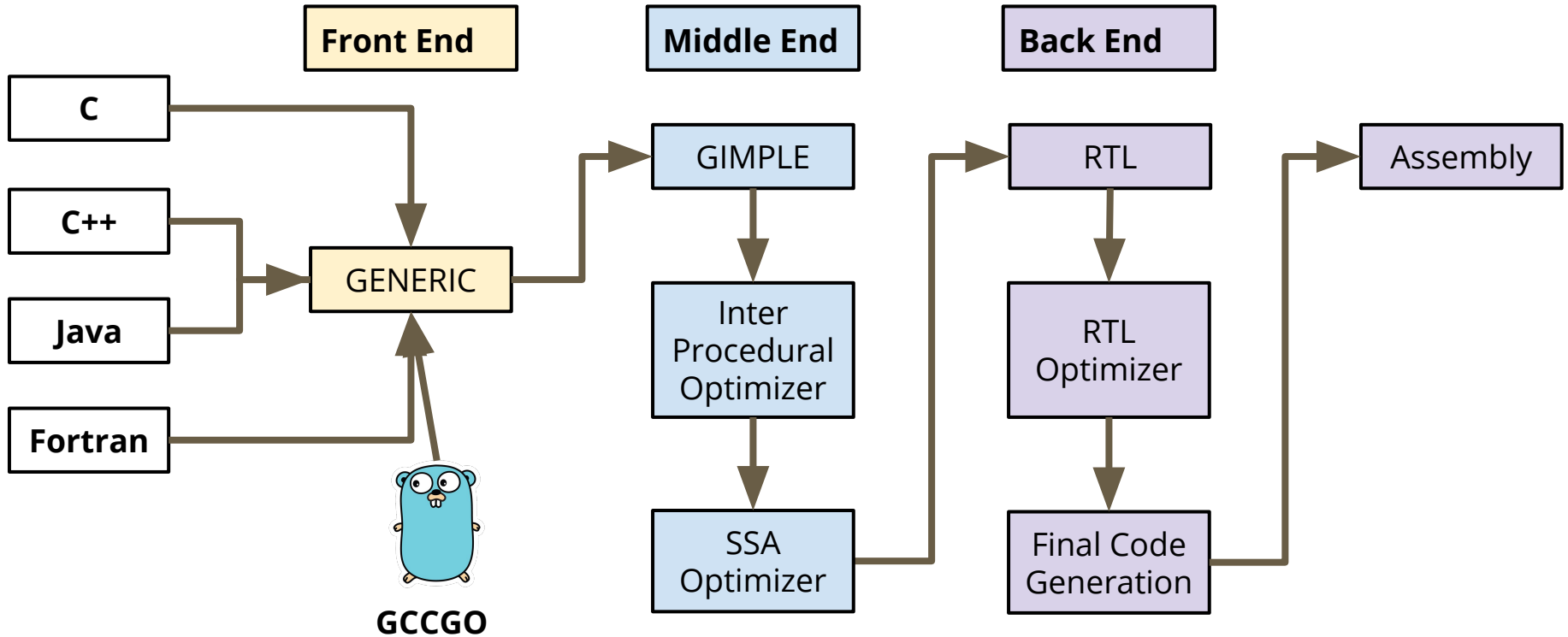


“Um compilador é um programa que consegue ler um programa em uma linguagem (linguagem de origem) e traduzir para um programa equivalente em outra linguagem (linguagem destino)” [Aho, 2a edição]

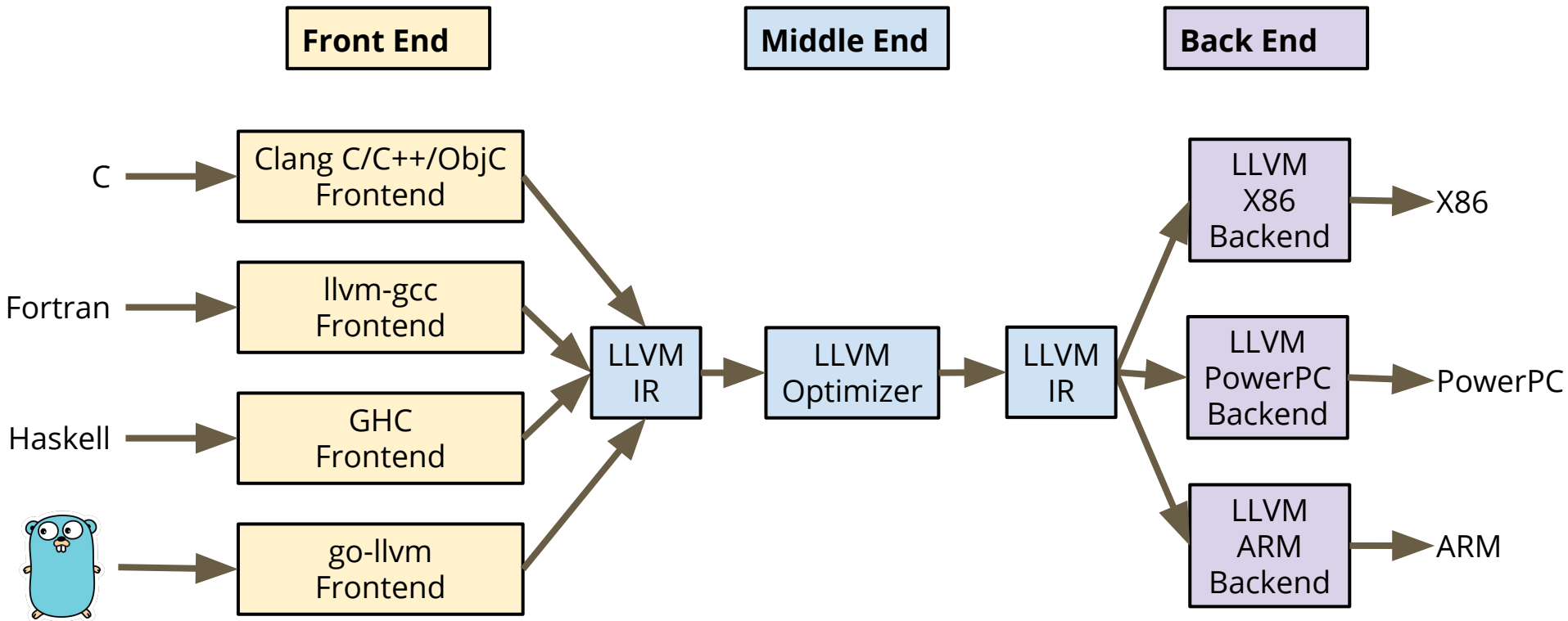
Arquitetura clássica de um compilador



Arquitetura do GCC



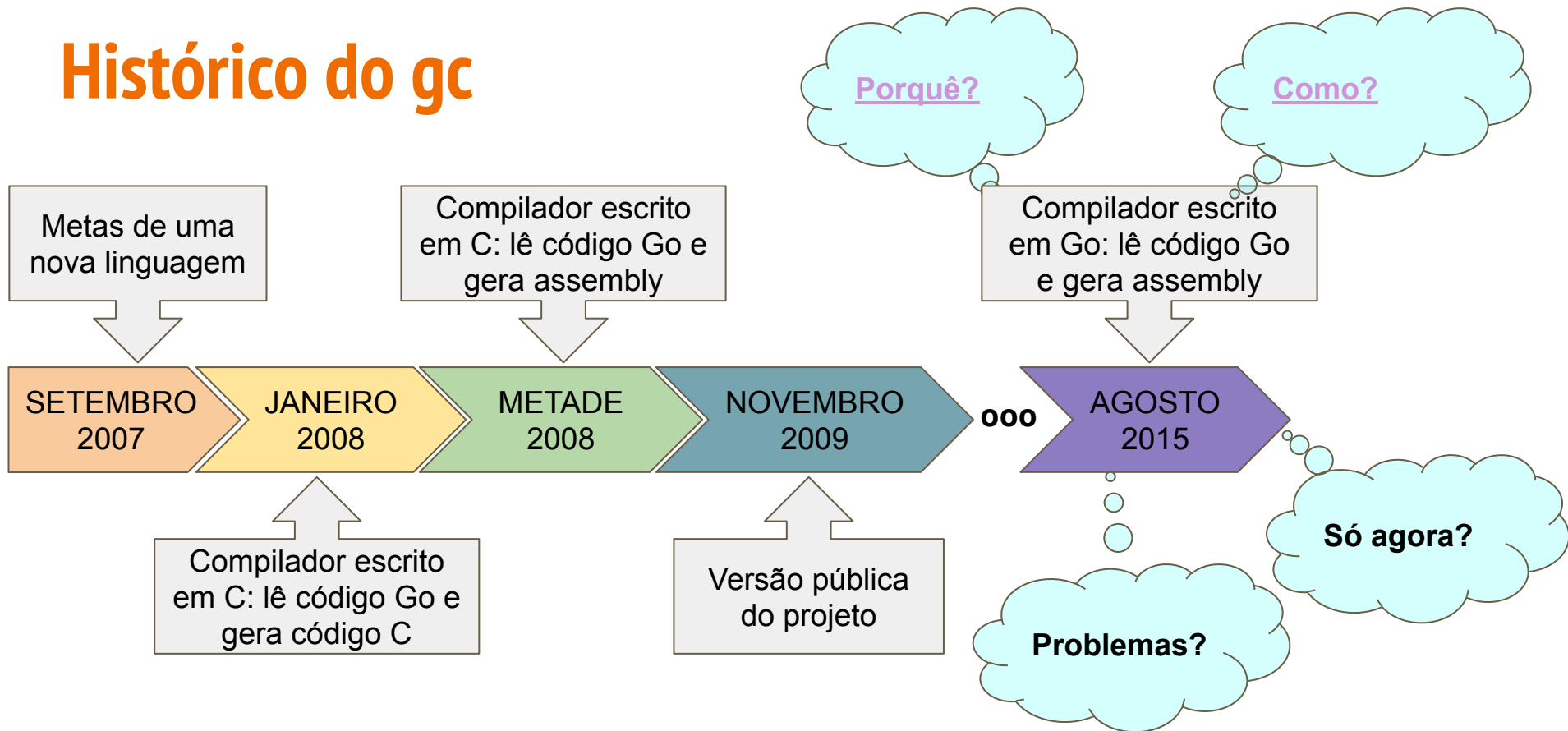
Arquitetura do LLVM



Quem é o gc?

- gc (minúsculo) é o compilador de go
- Sem confusões com o GC (maiúsculo), que é o Garbage Collector
- Pacote go/lexer não faz parte do gc
 - go/lexer é usado por ferramentas como gofmt, golint, ...

Histórico do gc



O que o compilador do Go faz?

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Println("Hello!")
```

```
}
```



```
XORPS X0, X0
```

```
MOVUPS X0, 0x38(SP)
```

```
LEAQ 0x38(SP), AX
```

```
MOVQ AX, 0x30(SP)
```

```
TESTB AL, 0(AX)
```

```
LEAQ 0x10998(IP),
```

```
MOVQ CX, 0x38(SP)
```

```
LEAQ main.static
```

```
MOVQ CX, 0x40(SP)
```

```
TESTB AL, 0(AX)
```

```
JMP 0x486a9d
```

```
MOVQ AX, 0x48(SP)
```

```
MOVQ $0x1, 0x50($
```

```
MOVQ $0x1, 0x58($
```

```
MOVQ AX, 0(SP)
```

```
MOVQ $0x1, 0x8(SP)
```

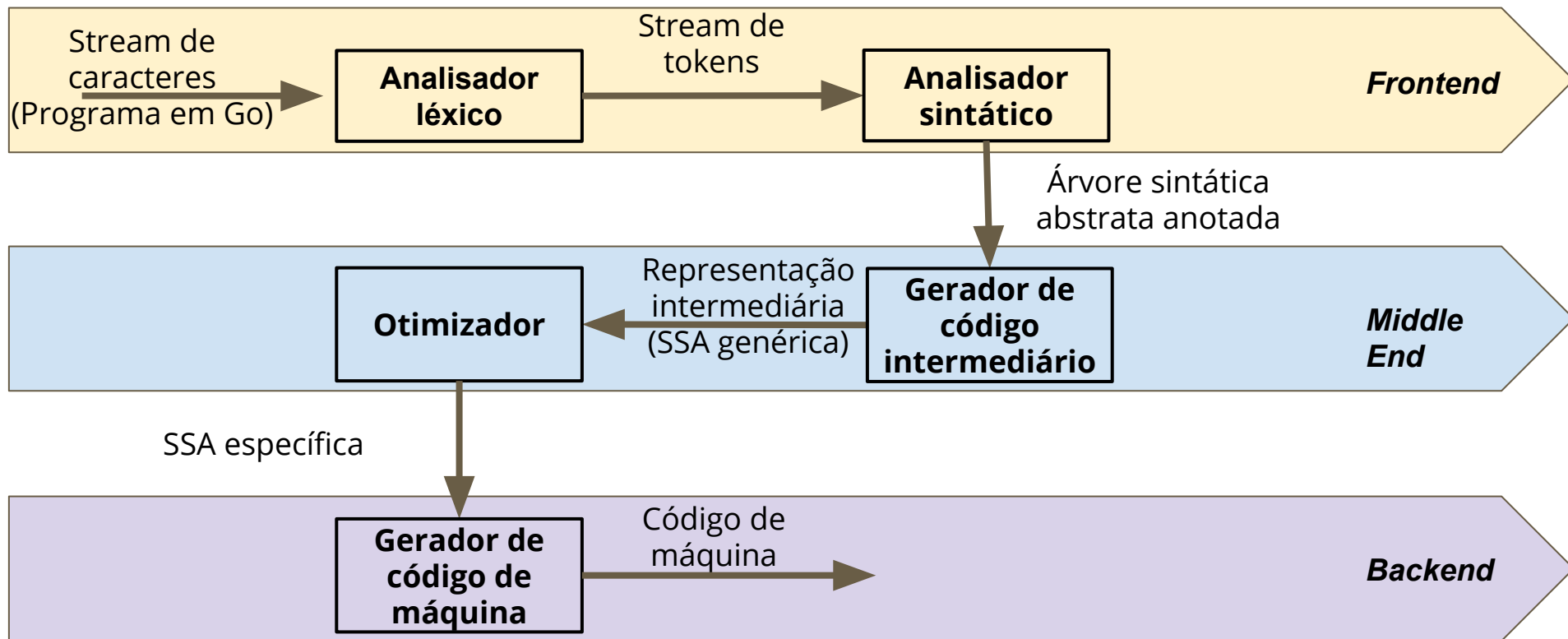
```
MOVQ $0x1, 0x10(SP)
```

```
CALL fmt.Println(SB)
```

+ RUNTIME

- garbage collector
- stack management
- concurrency
- stdlib
- ...

Arquitetura do gc / etapas de compilação




Mas como isso acontece?

```
package main

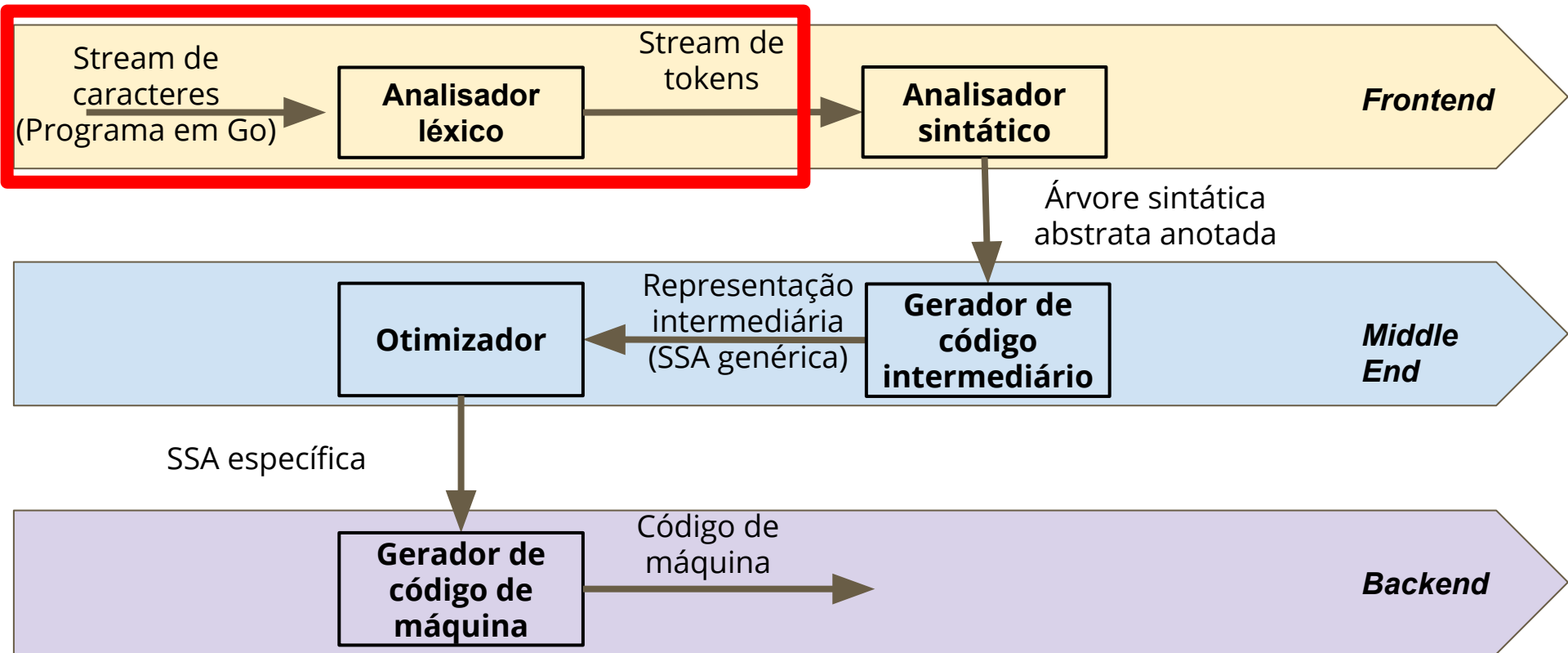
import "fmt"

func main() {
    x := 1
    y := 2
    fmt.Println(x+y)
}
```



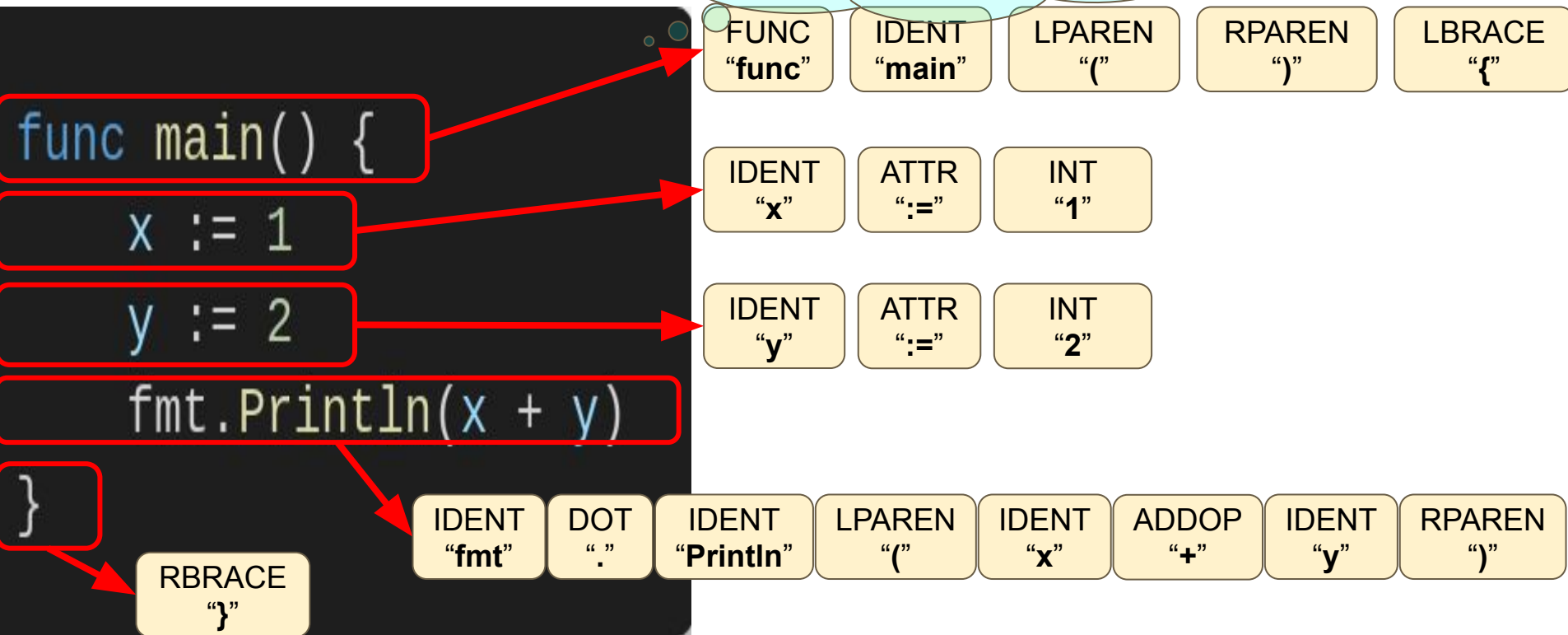
**Por que não um
“Hello World”?**

Analizador léxico

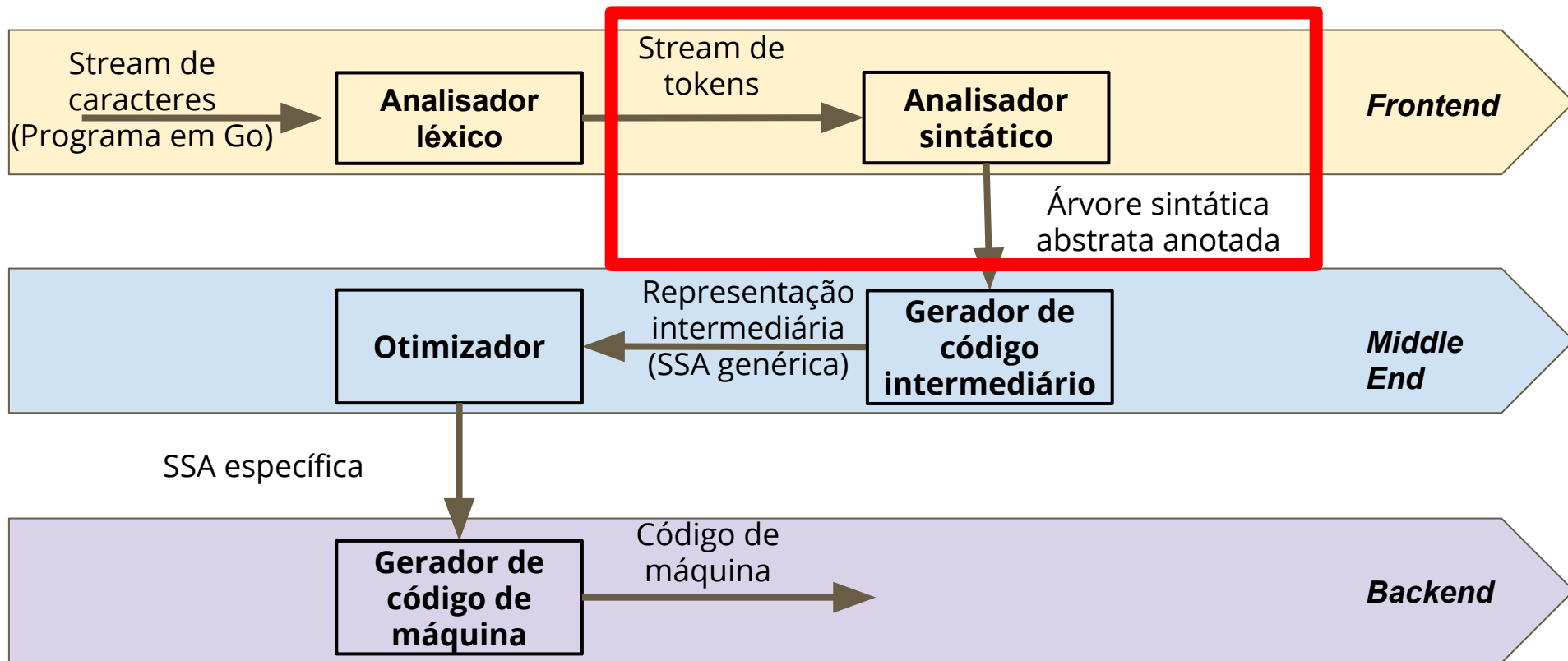


Analizador léxico

Espaçamento? Tabulação?
Salto de linha?

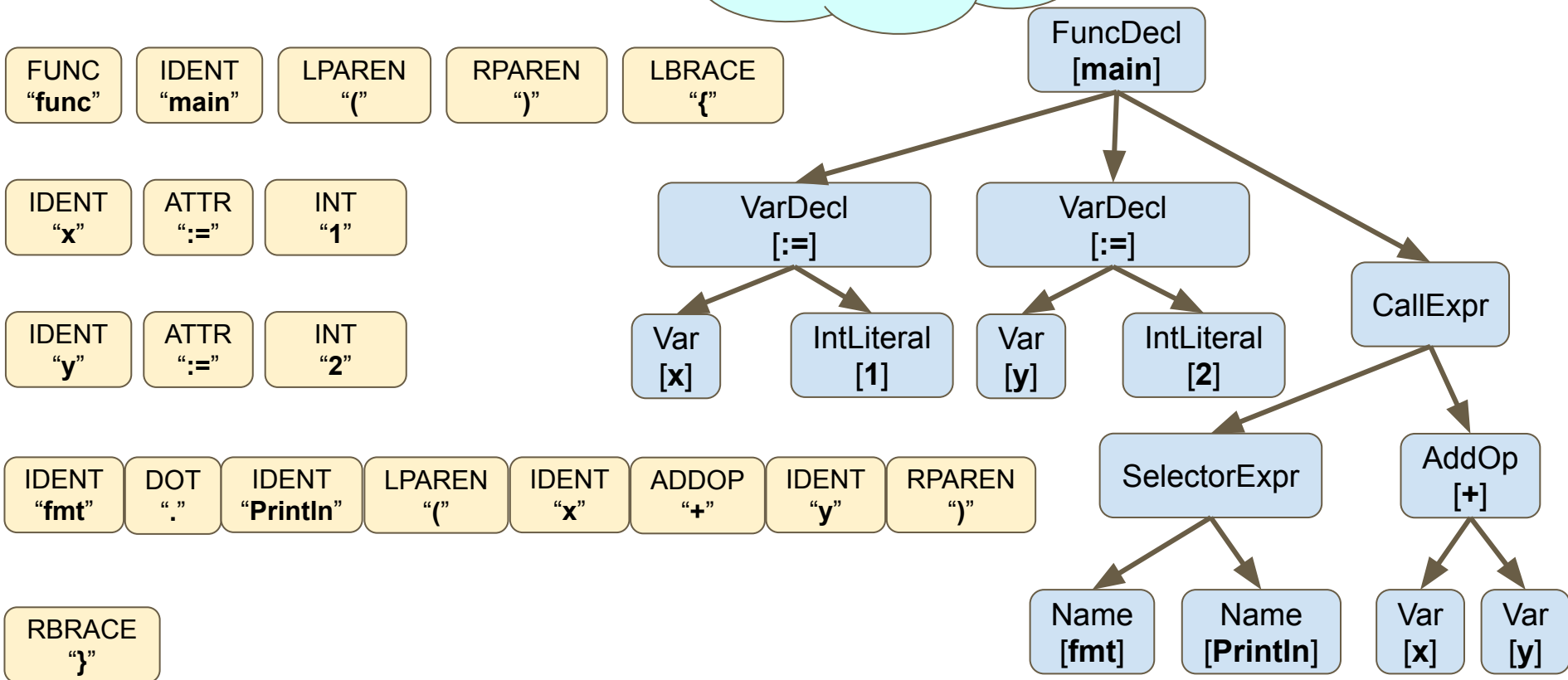


Analizador sintático

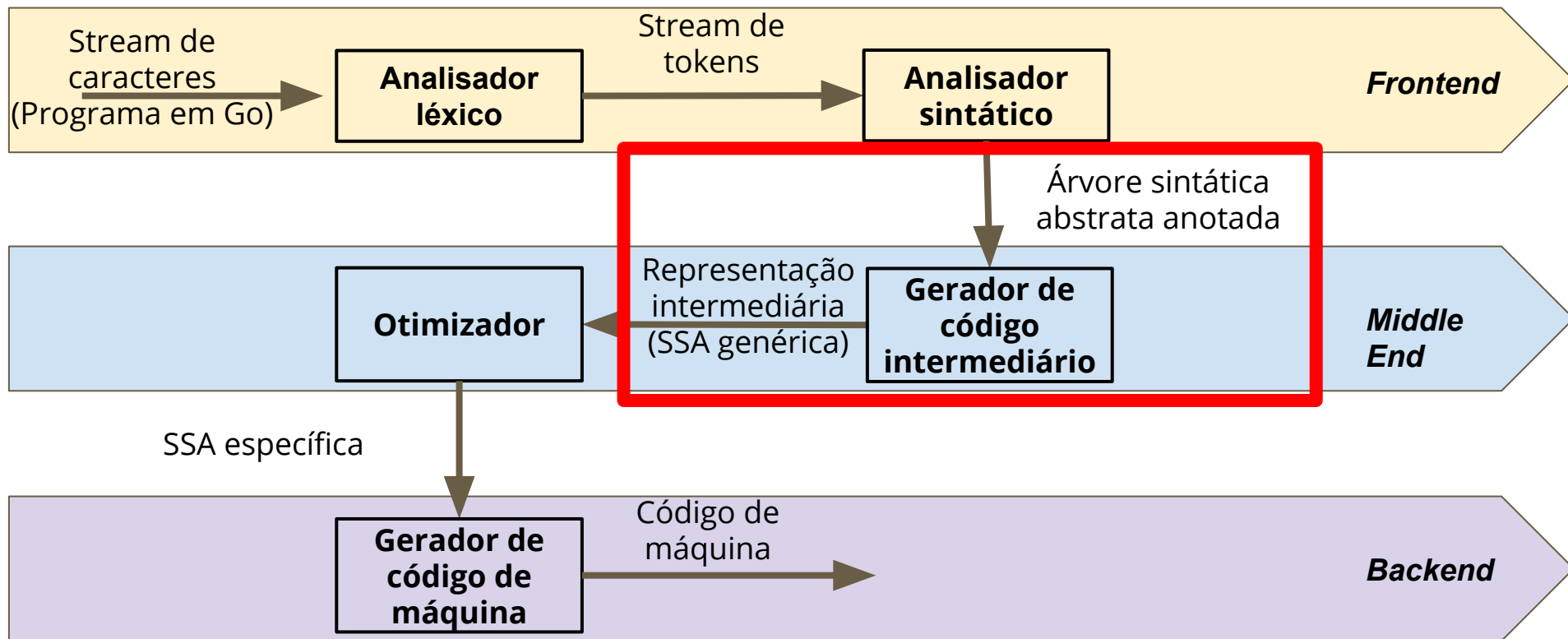


Analizador sintático

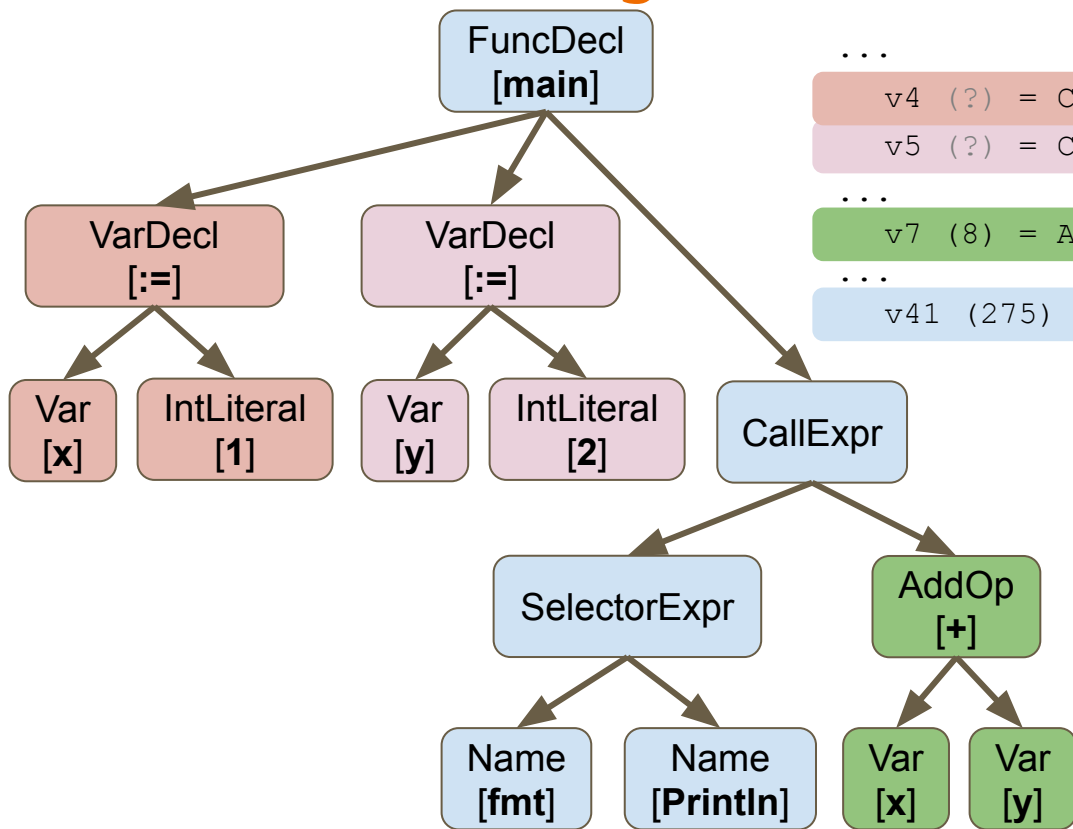
Delimitadores?



Gerador de código intermediário



Gerador de código intermediário



...

```
v4 (?) = Const64 <int> [1] (x[int])
```

```
v5 (?) = Const64 <int> [2] (y[int])
```

...

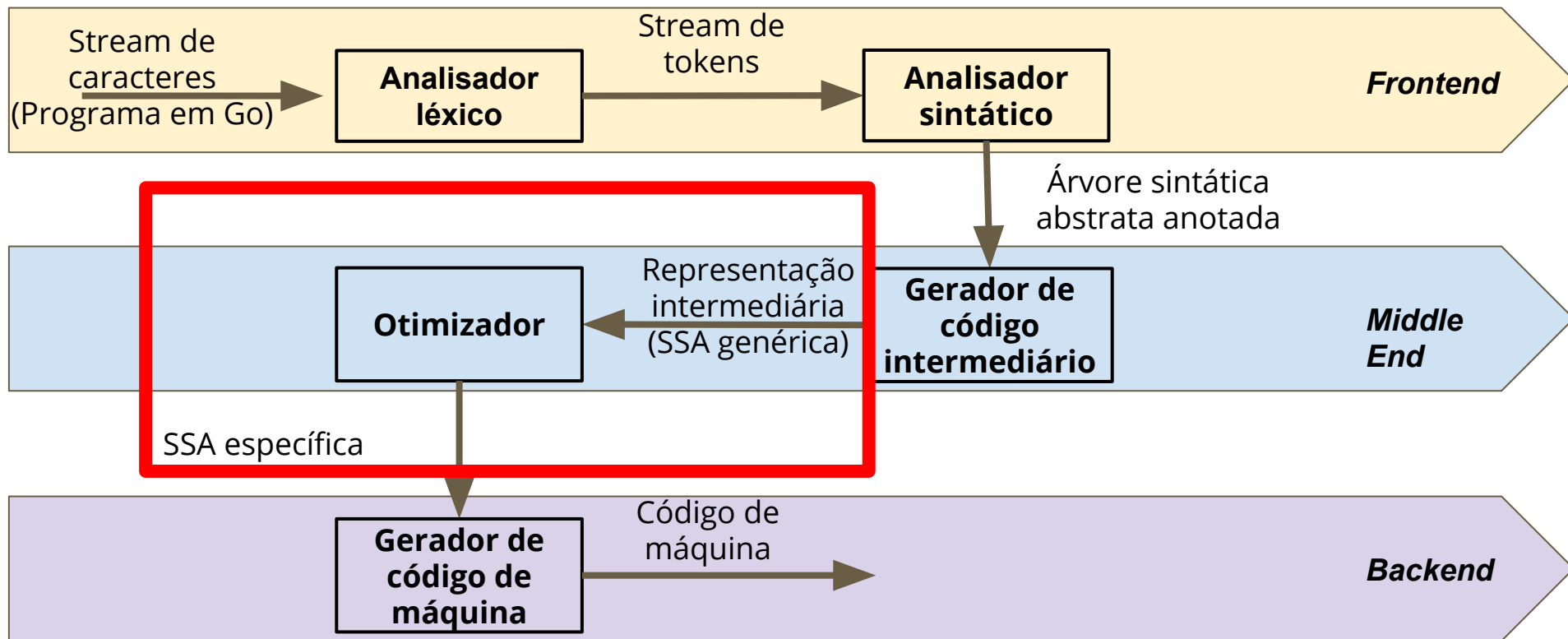
```
v7 (8) = Add64 <int> v4 v5
```

...

```
v41 (275) = StaticCall <mem> {fmt.Fprintln} [64] v40
```

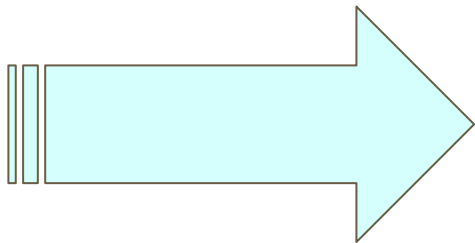
Semântica?

Otimizador



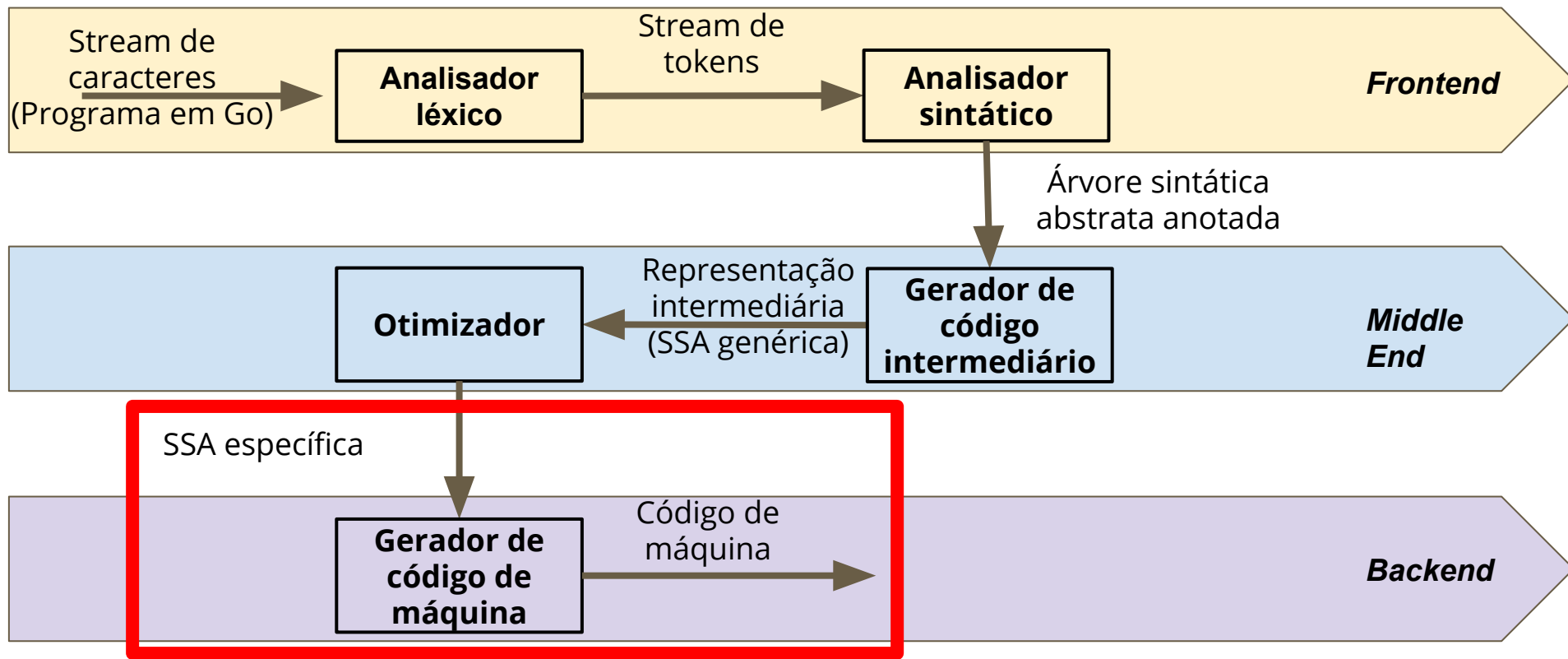
Otimizador

```
...  
v4 (?) = Const64 <int> [1] (x[int])  
v5 (?) = Const64 <int> [2] (y[int])  
...  
v7 (8) = Add64 <int> v4 v5  
  
v41 (275) = StaticCall <mem> {fmt.Fprintln} [64] v40
```



```
...  
v7 (8) = Const64 <int> [3]  
...  
v41 (275) = StaticCall <mem> {fmt.Fprintln} [64] v40
```

Gerador de código de máquina



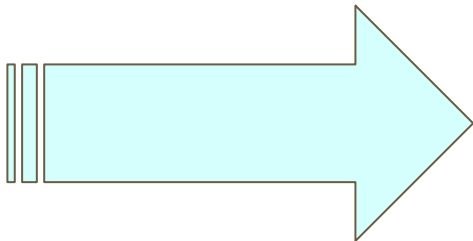
Gerador de código de máquina

...

```
v7 (8) = Const64 <int> [3]
```

...

```
v41 (275) = StaticCall <mem> {fmt.Fprintln} [64] v40
```



```
# hello.go
```

```
00000 (5) TEXT    "".main(SB), ABIInternal
```

```
...
```

```
v8 00007 (+8) MOVQ  $3, (SP)
```

```
v9 00008 (8) CALL  runtime.convT64(SB)
```

```
...
```

```
# $GOROOT/src/fmt/print.go
```

```
...
```

```
v41 00036 (275) CALL fmt.Fprintln(SB)
```

```
b4 00037 (?) RET
```

```
00038 (?) END
```


Falar é fácil.... show me the code :-)

- Dois hacks feitos
 - while
 - operador ternário (teve [proposta](#) para Go 2.0, mas não evoluiu...)

Hacking (while)

```
package main

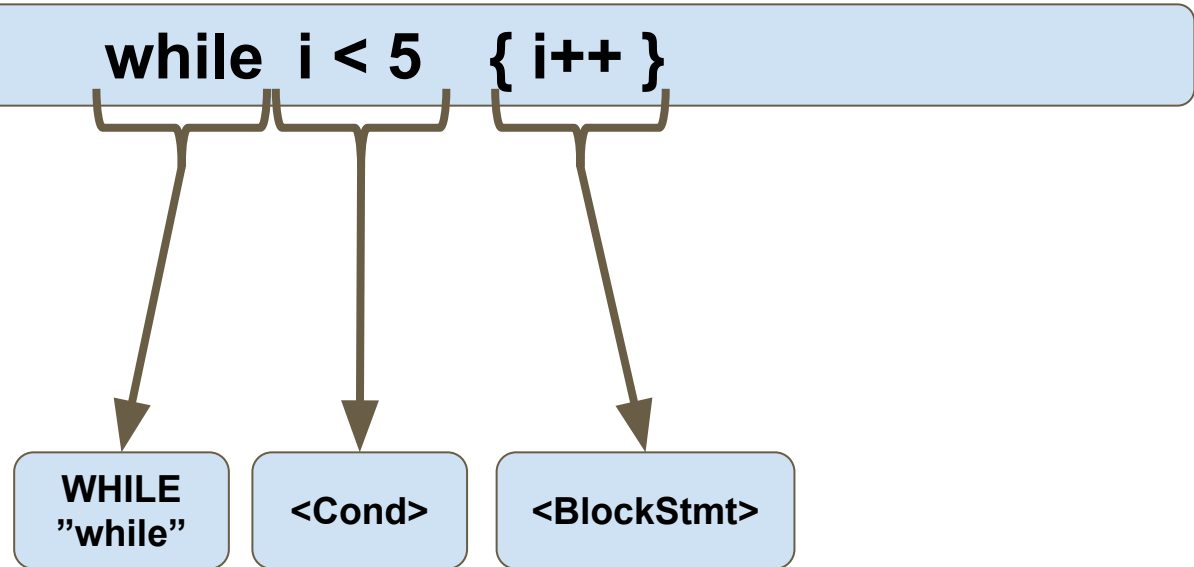
import "fmt"

func main() {
    i := 0
    while i < 5 {
        i++
        fmt.Println("Numero: ", i)
    }
}
```

Hacking (while)

\$ cat && go run while.go

Hacking 1 (while) - Novo statement




Hacking 1 (while) - Novo statement

Statement???

parser.go

```
// Statement =  
// Declaration | LabeledStmt | SimpleStmt |  
// GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |  
// FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt |  
// ForStmt | WhileStmt | DeferStmt .  
func (p *parser) stmtOrNil() Stmt {  
    if trace {  
        defer p.trace("stmt " + p.tok.String())()  
    }  
  
    ...  
  
    case _For:  
        return p.forStmt()  
  
    case _While:  
        return p.whileStmt()  
  
    case _Switch:  
        return p.switchStmt()
```

Hacking 1 (while) - Novo token



tokens.go

```
// keywords
_Break      // break
_Case       // case
_Chan       // chan
_Const      // const
_Continue   // continue
_Default    // default
_Defer      // defer
_Else       // else
_Fallthrough // fallthrough
_For        // for
_While      // while
_Func       // func
_Go         // go
```


Hacking 1 (while) - func whileStmt

ForStmt ???

func whileStmt em
parser.go

```
func (p *parser) whileStmt() Stmt {  
    if trace {  
        defer p.trace("whileStmt")()  
    }  
  
    s := new(ForStmt)  
    s.pos = p.pos()  
  
    s.Init, s.Cond, s.Post = p.header(_while)  
    s.Body = p.blockStmt("while clause")  
  
    return s  
}
```

Hacking 1 (while) - Define while tem condition



Ajuste para
parser
entender que
while tem
“condition”

```
func (p *parser) header(keyword token) (init SimpleStmt, cond Expr, post SimpleStmt) {
    p.want(keyword)

    if p.tok == _Lbrace {
        if keyword == _If {
            p.syntaxError("missing condition in if statement")
        }
        if keyword == _While {
            p.syntaxError("missing condition in while statement")
        }
    }
    return
}
```


Hacking 1 (while) - gc usa perfect hash



scanner.go

```
// hash is a perfect hash function for keywords.
// It assumes that s has at least length 2.
func hash(s []byte) uint {
    return (uint(s[0])<<4 ^ uint(s[1]) + uint(len(s))) & uint(len(keywordMap)-1)
}

var keywordMap [1 << 7]token // size must be power of two

func init() {
    // populate keywordMap
    for tok := _Break; tok = _Var; tok++ {
        h := hash([]byte(tok.String()))
        if keywordMap[h] != 0 {
            panic("imperfect hash")
        }
        keywordMap[h] = tok
    }
}
```

Hacking 1 (while) - Gerando o novo gc

- Gerar os tokens novamente (???)
 - `$ stringer -type token -linecomment tokens.go`
- Gerar os binários do compilador
 - `$./all.bash` (gera binário, bibliotecas, executa testes)
- `$./bin/go run hacks/while.go`

Hacking 1 (while) - O que NÃO foi feito?

- Não foi criado WhileStmt
- Analisador sintático transforma o while em um ForStmt
- As outras etapas acham que é um for...

Hacking 1 (while) - SSA.html

Compilador gera informações sobre todas as etapas de compilação :-)

Hacking 2 (operador ternário) - Expectativa...

```
package main

import "fmt"

func main() {
    production := false
    port := 80 if production else 8080
    fmt.Printf("Production: %v port: %d\n", production, port)

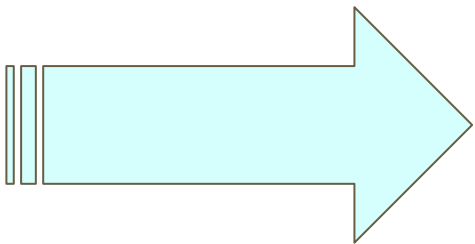
    production = true
    port = 80 if production else 8080
    fmt.Printf("Production: %v port: %d\n", production, port)
}
```

Qual a “treta”?

- gc é LL(1)
 - L: parser Left to Right
 - L: derivação mais à esquerda
 - (1): indica o look ahead (número de símbolos parser utiliza para tomar uma decisão)
- “Ser” LL1 não é ruim, e para Go está certo...
- Porém, eu queria transformar o operador ternário em um “if” normal durante o parser

Como eu queria resolver...

```
production := false  
port := 80 if production else 8080
```



```
production := false  
var port int  
if production {  
    port = 80  
} else {  
    port = 8080  
}
```

LL(1)

ATTR...
DEFINE!

IfStmt não é
compatível com
SimpleStmt :-/

```
port := 80 if production else 8080
```

IDENT...

SimpleStmt!

INT

Expectativa vs realidade...

PUDIM



FACEBOOK.COM/REALIDEXEXPECTATIVA

Hacking 2 (operador ternário) - Realidade :-)

```
package main

import "fmt"

func main() {
    production := false
    port := 0
    let port = 80 if production else 8080
    fmt.Printf("Production: %v port: %d\n", production, port)

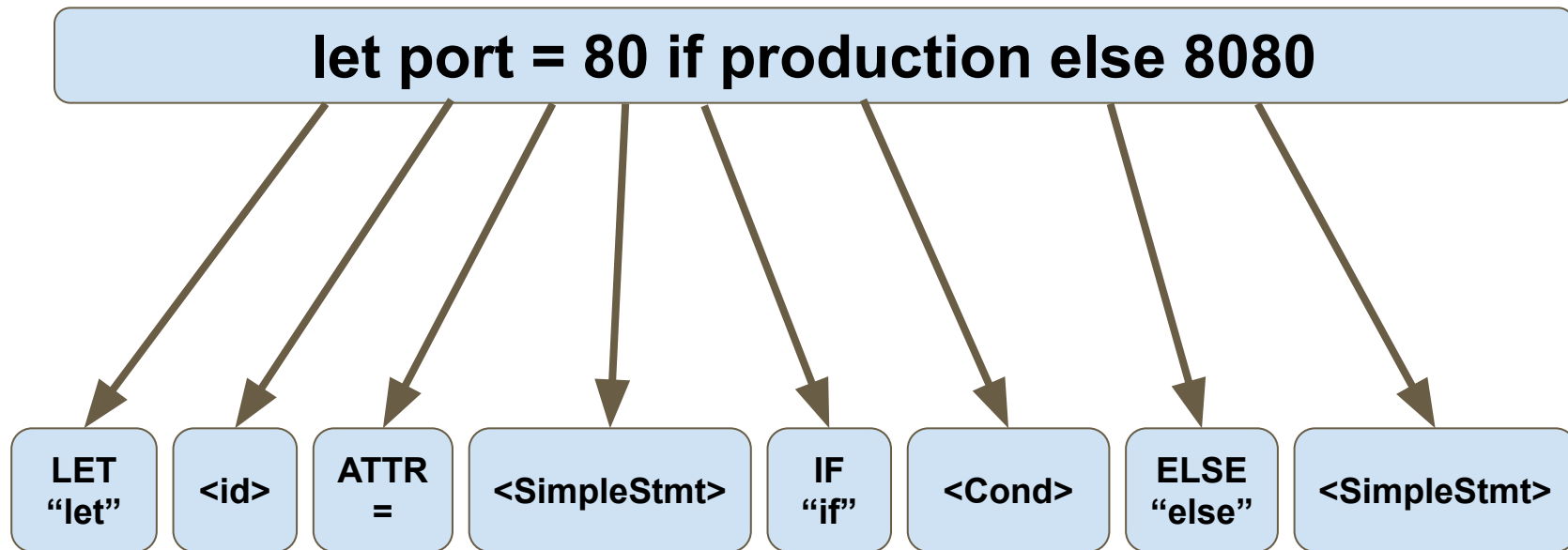
    production = true
    let port = 80 if production else 8080
    fmt.Printf("Production: %v port: %d\n", production, port)
}
```

Porque assim funciona???

Hacking 2 (operador ternário)

```
$ cat && go run ternary_operator.go
```

Hacking 2 (operador ternário) - Novo statement



Hacking 2 (operador ternário) - Novo statement



```
// Statement =  
// Declaration | LabeledStmt | SimpleStmt |  
// GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |  
// FallthroughStmt | Block | IfStmt | LetStmt | SwitchStmt |  
// SelectStmt | ForStmt | WhileStmt | DeferStmt .  
func (p *parser) stmtOrNil() Stmt {  
    if trace {  
        defer p.trace("stmt " + p.tok.String())()  
    }  
  
    ...  
  
    case _Switch:  
        return p.switchStmt()  
  
    ...  
  
    case _If:  
        return p.ifStmt()  
  
    case _Let:  
        return p.letStmt()
```

let <id> = {
<SimpleStmt> {
if <Cond> {
else <SimpleStmt> {

```
// let <id> = <SimpleStmt> if <Cond> else <SimpleStmt>
func (p *parser) letStmt() *IfStmt {
    s := new(IfStmt)
    s.pos = p.pos()
    p.want(_Let)
    id := p.expr()
    ...
    pos := p.pos()
    op := p.op
    p.next()

    thenStmt := p.newAssignStmt(pos, op, id, p.expr())
    bs := new(BlockStmt)
    bs.pos = pos
    bs.List = append(bs.List, thenStmt)
    s.Then = bs
    p.want(_If)
    condStmt := p.simpleStmt(nil, 0)
    ...
    p.want(_Else)
    elseStmt := p.newAssignStmt(pos, op, id, p.expr())
    bs = new(BlockStmt)
    bs.pos = pos
    bs.List = append(bs.List, elseStmt)
    s.Else = bs
```

Hacking 2 (operador ternário) - O que foi feito?

- Sequência similar ao while...
 - Novo token (let)
 - Novo statement
 - Gerar os tokens novamente
 - Gerar a nova versão do compilador

Hacking 2 (operador ternário) - SSA.html

Resumindo....

- gc é um projeto MUITO legal!
- No geral, bem legível
- Boa documentação
- Tem o “legado” da conversão direta de C para Go
- Primeiro compilador “real” que tudo é óbvio :-D

Coisas que (NÃO) vale a pena se preocupar

- Foque em clean code, código organizado, legível, ...
- Programe para outros humanos entenderem, e não para máquinas
 - Código, no geral, é escrito uma vez, mas lido milhares de vezes (por humanos!!!!)
- Pode deixar que o gc lida MUITO bem com otimizações :-)

```
x = y * 4 / 2    ==>    x = y * 2
```

```
...
```

```
y = 4  
x = x / y    ==>    x = x / 4
```

Referências

- Links interessantes sobre o gc
 - [Histórico](#)
 - [Repositório do go](#): gc está [aqui](#)
 - [Introduction to the go compiler](#)
 - [Go contribution guide](#)
 - [SSA rules](#)
- [Porque reescreveram o gc em go](#) e [como](#)
- Adding a new statement: [part 1](#) and [part 2](#)
- [Go toolchain](#)
- [Hacking go compiler internals](#)
- Para gerar SSA: GOSSAFUNC=main go tool compile hello.go (gera ssa.html)
- [Compiler explorer](#)
- [Aho, 2nd edition](#)

Dúvidas????



Alex Sandro Garzão

alexgarzao@gmail.com

<https://www.linkedin.com/in/alexgarzao/>

<https://github.com/alexgarzao>

<https://twitter.com/alexgarzao>