gofmt 的文化演变
The Cultural Evolution of gofmt

Robert Griesemer
Google, Inc.

gofmt

- Go源代码格式化工具
- 定义了“标准“格式
- golang.org代码库中所有提交的Go代码都必须通过gofmt格式化过
- 除了gofmt之外，相同功能可以通过go/format库获得
- 不需要设置!

初衷

- 代码审查是软件工程的最佳实践
- 代码审查是基于代码规范和正规格式的
- **太多时间浪费在审查格式上而不是代码本身了**
- 但是这工作对机器来说是最好不过了的
- 第一个决定就是要写一个好的格式美化器

历史

- 格式美化器和代码美化工具在计算机发展的早期就已出现
- 对于产生可读的Lisp代码很重要的:

```
GRINDEF  (Bill Gosper, 1967)          第一个计算行长度
```

- 其他:

```
SOAP     (R. Scowen et al, 1969)      简化了晦涩的算法程序
NEATER2  (Ken Conrow, R. Smith, 1970) PL／1格式器，作为（早期的）纠错工具
cb       (Unix Version 7, 1979)       C程序美化器
indent   (4.2 BSD, 1983)              缩进和格化化C代码
等等
```

- 最近的:

```
ClangFormat                           C/C++/Objective-C 格式器
Uncrustify                            C, C++, C#, ObjectiveC, D, Java, Pawn and VALA的美化器
等等
```

事实上

- 在2007年，没人喜欢代码格式器
- 例外：IDE强制的格式化
- 但是：很多程序员不用IDE...
- 问题：如果是格式化太具有毁坏性，那么就没有人会用
- 被忽视的观点：“刚刚好“的，统一化的格式是好过于各种不同的格式的。
- 规范的价值在于：整齐划一，而不是完美

好的格式美化器的问题

- 当越多人思考他们自己的格式风格的时候，他们就变得更加固执于此了
- 错误的结论：自动格式器必须要有很多选项!
- 但是有很多选项的格式器其实违背他们的目的
- 此外，支持很多选项是难的
- 尊重用户的想法是最关键的
- 处理注释是很难的
- 语言本身也会增加很多额外的复杂度（比如，C的宏）

格式化Go

尽量保证其简单

- 小的语言能让事情变得简单
- 不要为行长度烦恼
- 相反的，尊重用户：考虑原有代码中的断行
- 不要支持任何选项
- 使其使用傻瓜化

一个格化标准搞定所有!

gofmt的基本结构

- 源代码的处理
- 基本的格式化
- 附加：注释的处理
- 完善：代码和注释的对齐
- 但是，没有牛X的通用布局算法
- 相反的：基于节点的精细优化

处理源代码

- 使用`go/scanner`, `go/parser`及其相关的库
- 给每一个go文件生成一个抽象语法树
- 每一个语法结构都有相应的AST节点

```
// Syntax of an if statement.
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .

// An IfStmt node represents an if statement.
IfStmt struct {
    If   token.Pos // position of "if" keyword
    Init Stmt      // initialization statement; or nil
    Cond Expr      // condition
    Body *BlockStmt
    Else Stmt // else branch; or nil
}
```

- AST节点有（选择性的）位置信息。

基本的格式化

- 遍历AST然后打印每个节点

```
case *ast.IfStmt:
    p.print(token.IF)
    p.controlClause(false, s.Init, s.Cond, nil)
    p.block(s.Body, 1)
    if s.Else != nil {
        p.print(blank, token.ELSE, blank)
        switch s.Else.(type) {
        case *ast.BlockStmt, *ast.IfStmt:
            p.stmt(s.Else, nextIsRBrace)
        default:
            p.print(token.LBRACE, indent, formfeed)
            p.stmt(s.Else, true)
            p.print(unindent, formfeed, token.RBRACE)
        }
    }
```

- 打印器（`p.print`）接收包括位置和空格符等的一系列记号

细致的调节

- 基于优先级安排操作数之间的空格.
- 提高表达式的可读性.

```
x = a + b
x = a + b*c
if a+b <= d {
if a+b*c <= d {
```

- 使用位置信息决定何时换行.
- 其他一些策略.

注释的处理

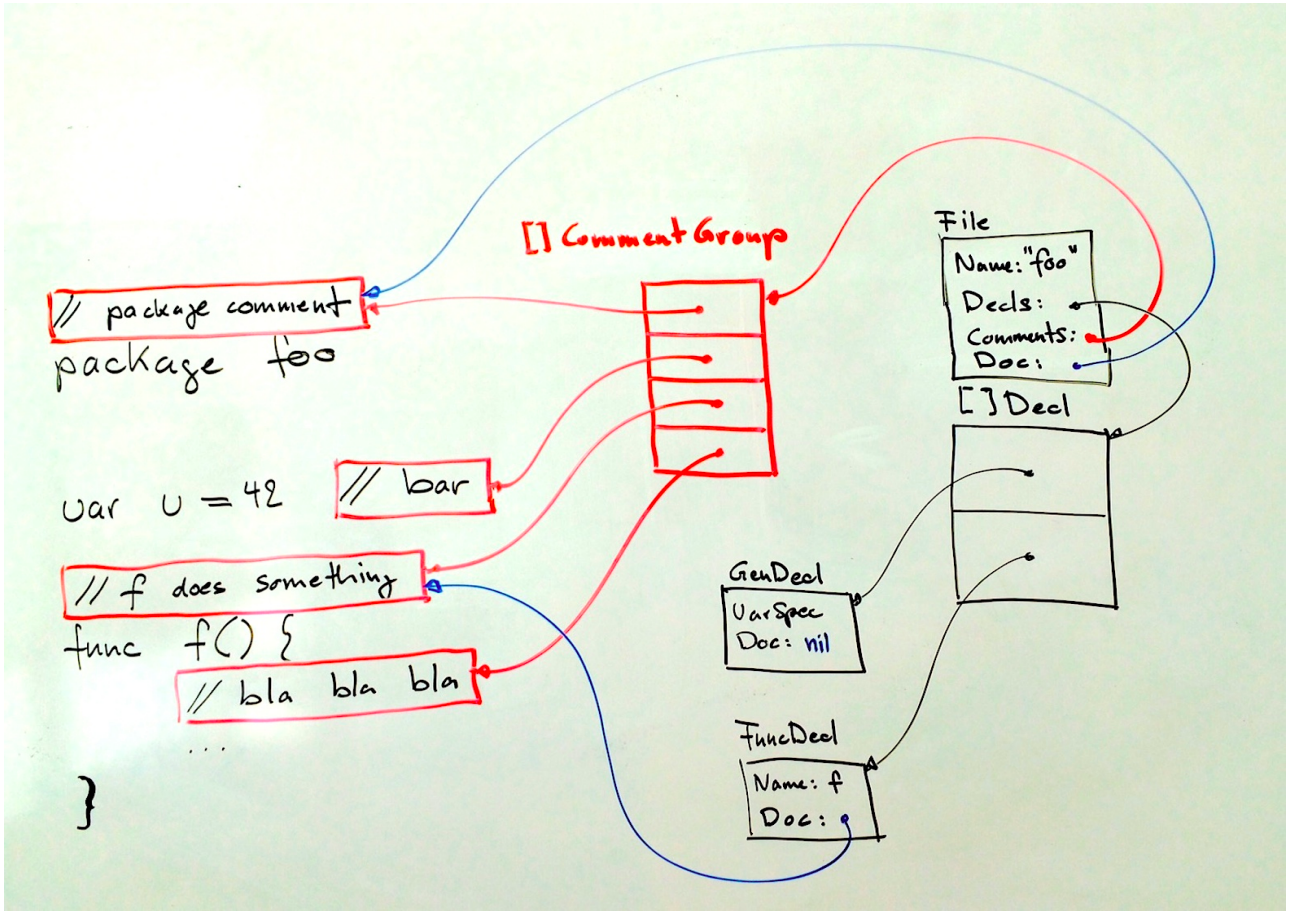
- 注释可以出现在程序的任何两个词汇之间.
- 通常情况下，不能很明显的知道注释属于哪个 AST 节点.
- 注释经常是成组出现:

```
// A CommentGroup represents a sequence of comments
// with no other tokens and no empty lines between.
//
type CommentGroup struct {
    List []*Comment // len(List) > 0
}
```

- 成组的注释被处理为一个大的注释.

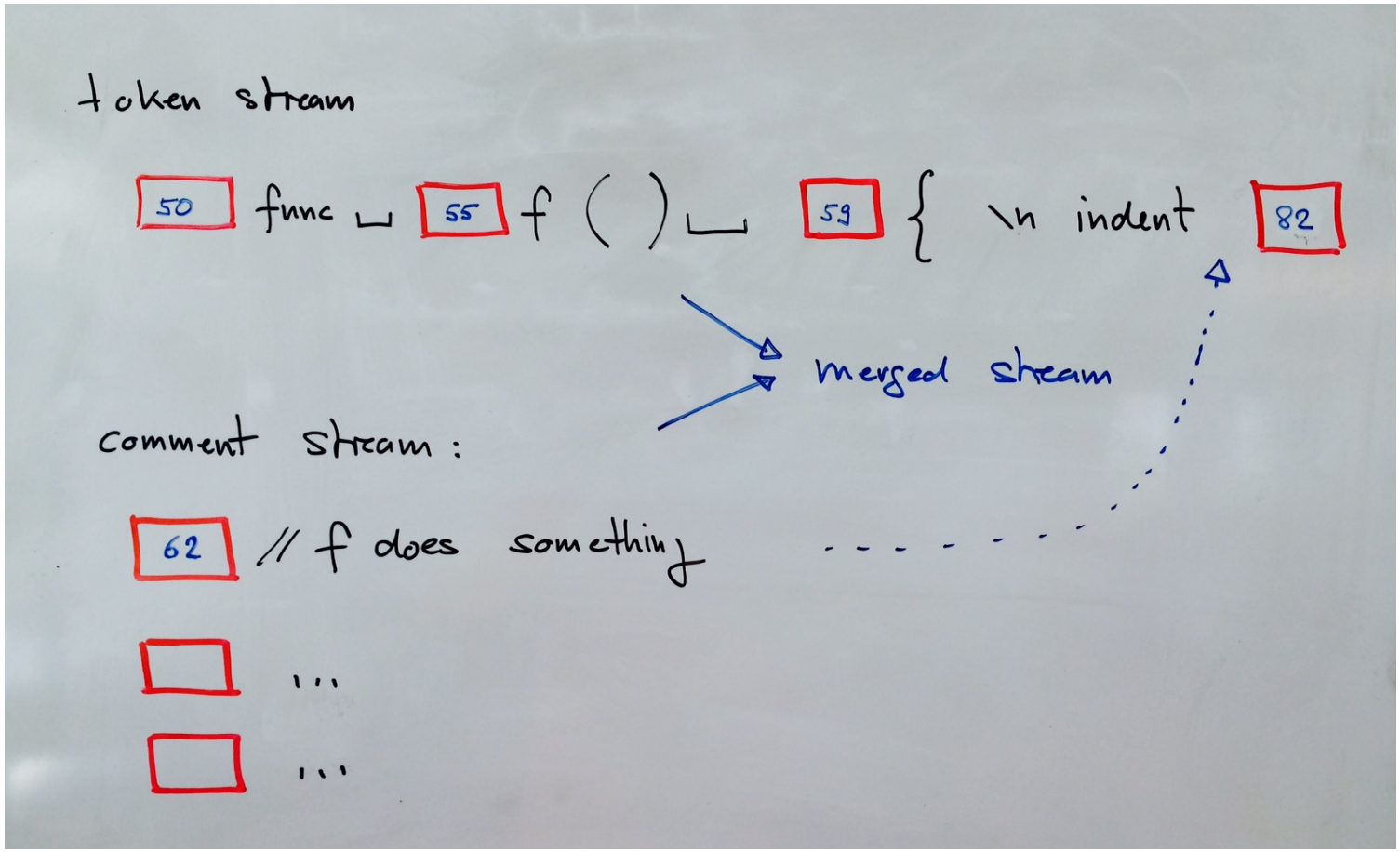
注释在 AST 上的表达

- 注释组的连续列表被连接到 AST 的文件节点.
- 另外，一些被标示为 *doc strings* 的注释被连接到声明节点.



格式化注释

- 基本的办法：基于位置信息合并词汇流和注释流.



魔鬼就在细节中

- 在源代码中估计当前的位置.
- 比较当前的位置和注释的位置去决定下一个是什么.
- 词汇也包含了空格词汇 - 注释必须被合理的分布!
- 维持一个未被打印的空格缓冲区，在下一个词汇之前输出，然后分布注释.
- 多种策略得以正确地处理空格.
- 很多次的尝试和错误.

格式化单独的注释

- 区分代码行和注释.
- 努力对多行注释进行合理的缩进.

```
func f() {                  func() {
 /*                                 /*
  * foo                               * foo
  * bar          ==>                 * bar
  * bal                               * bal
 */                                  */
       if ...                        if  ...
}                           }
```

- 但并不总是能够处理正确.
- 想达到两个效果：注释能够缩进，注释的内容不进行处理。还没有好的解决办法.

对齐

- 仔细选择的对齐可以让代码更容易阅读.

```
var (                                   var (
        x, y int = 2, 3 // foo                  x, y int     = 2, 3 // foo
        z float32 // bar          ==>           z    float32        // bar
        s string // bal                        s    string         // bal
)                                       )
```

- 很难进行手工维护 (制表符并不能够做到）.
- 但是却非常适合使用格式化工具.

灵活的制表符宽度

通常的制表符把当前的写位置移动到下一个固定的位置.

基本的办法：让制表符宽度更加灵活.

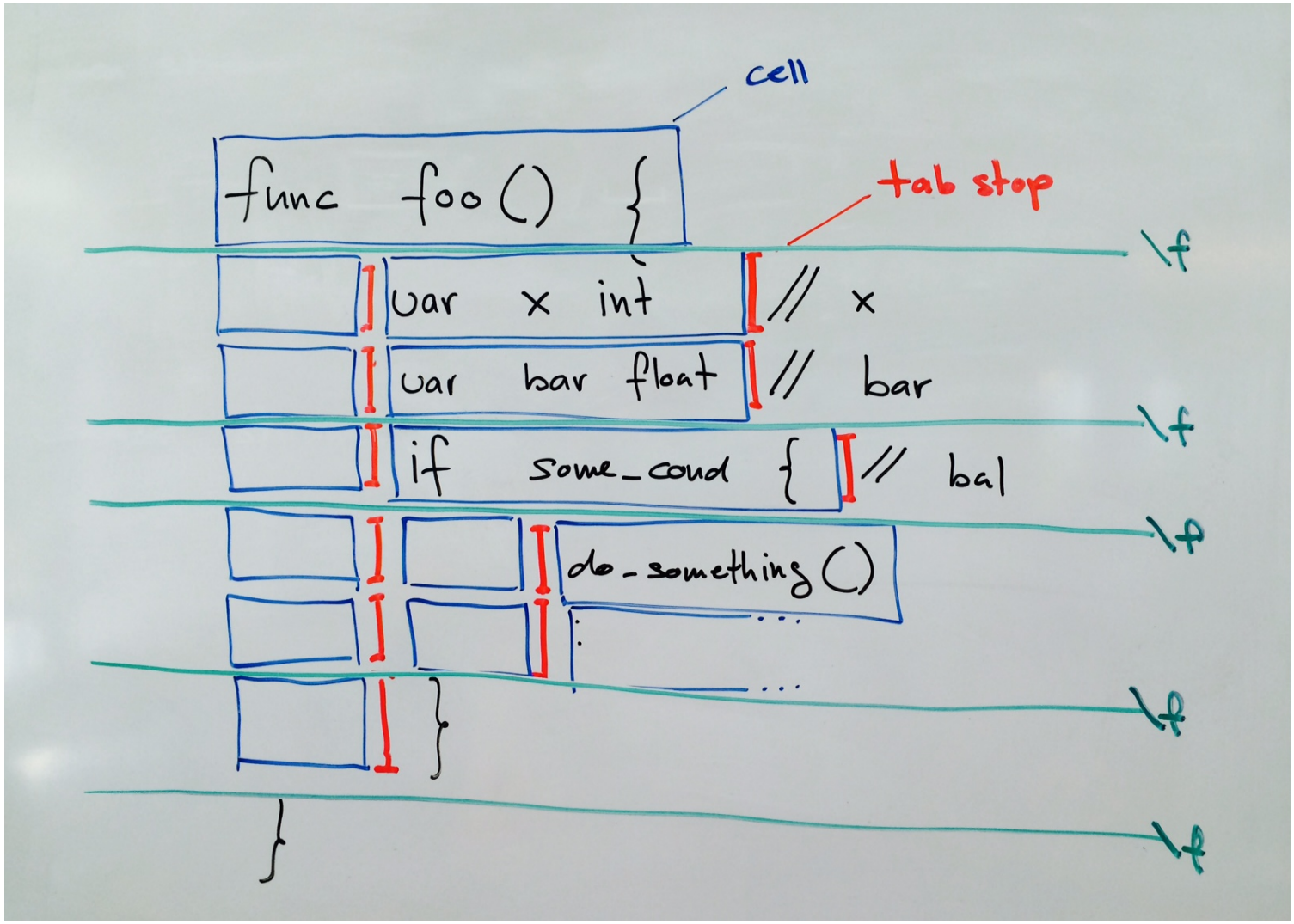
- 制表符可以标示一个文本单元的结束位置.
- 一个列块是一个连续的相邻的单元.
- 一个列块的宽度可以到达多个单元里最宽文本的宽度.

被 Nick Gravgaard 提出于2006

nickgravgaard.com/elastic-tabstops/ (http://nickgravgaard.com/elastic-tabstops/)

实现在 `text/tabwriter` 包中.

灵活制表符宽度的展示


cell
tab stop
func foo() {
var x int // x
var bar float // bar
if some_cond { // bal
do_something()
}
}
\t
\t
\t
\t
\t

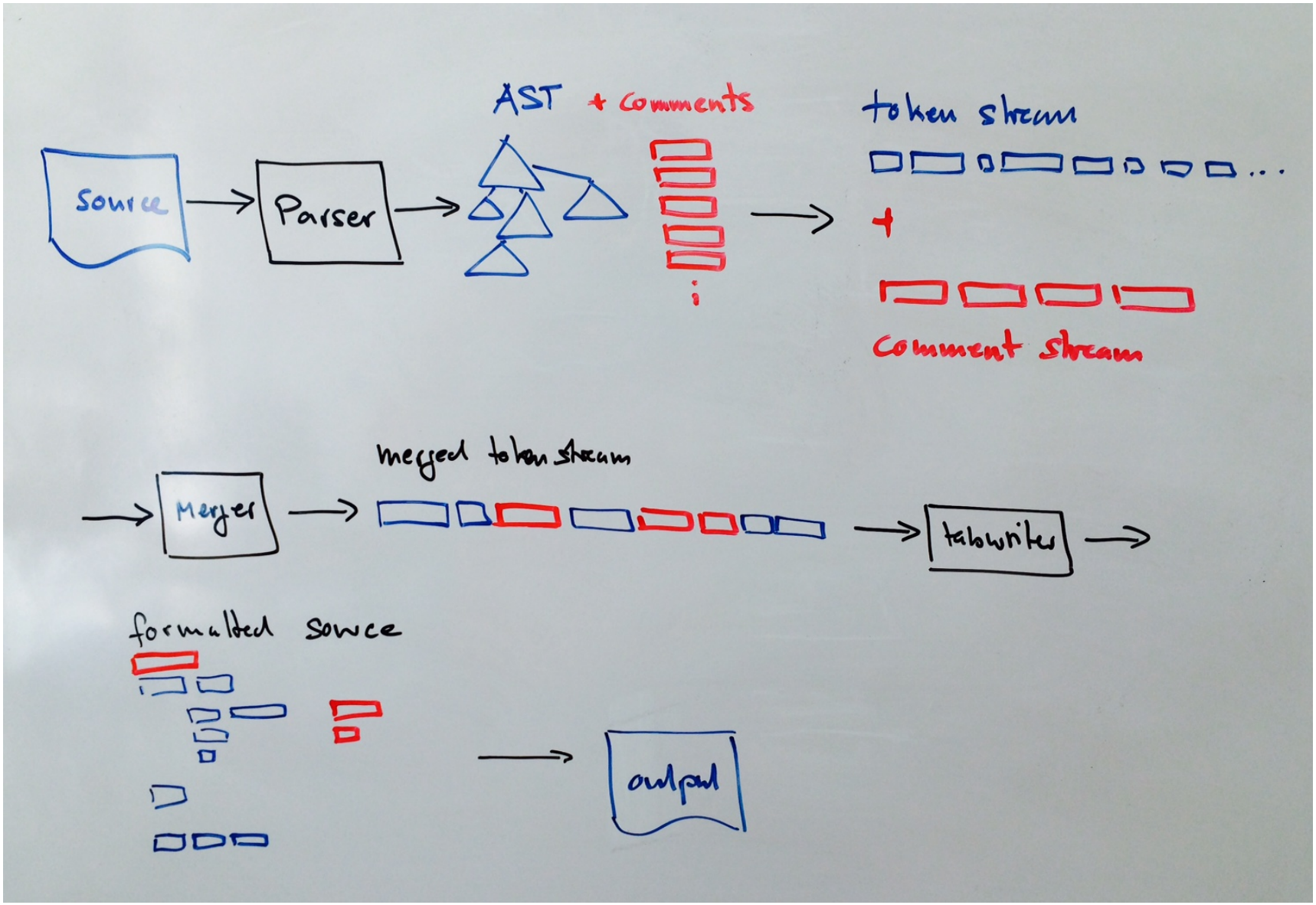
综合在一起 (1)

- 分析器生成 AST.
- 打印工具递归地打印AST，使用制表符去灵活的标示制表符的位置.
- 产生的词汇，位置和空格流会和注释流进行合并.
- 词汇会扩展为字符串，所有的文本流将会被制表符写入器处理.
- 制表符写入器会将制表符替换为合适数量的空格.

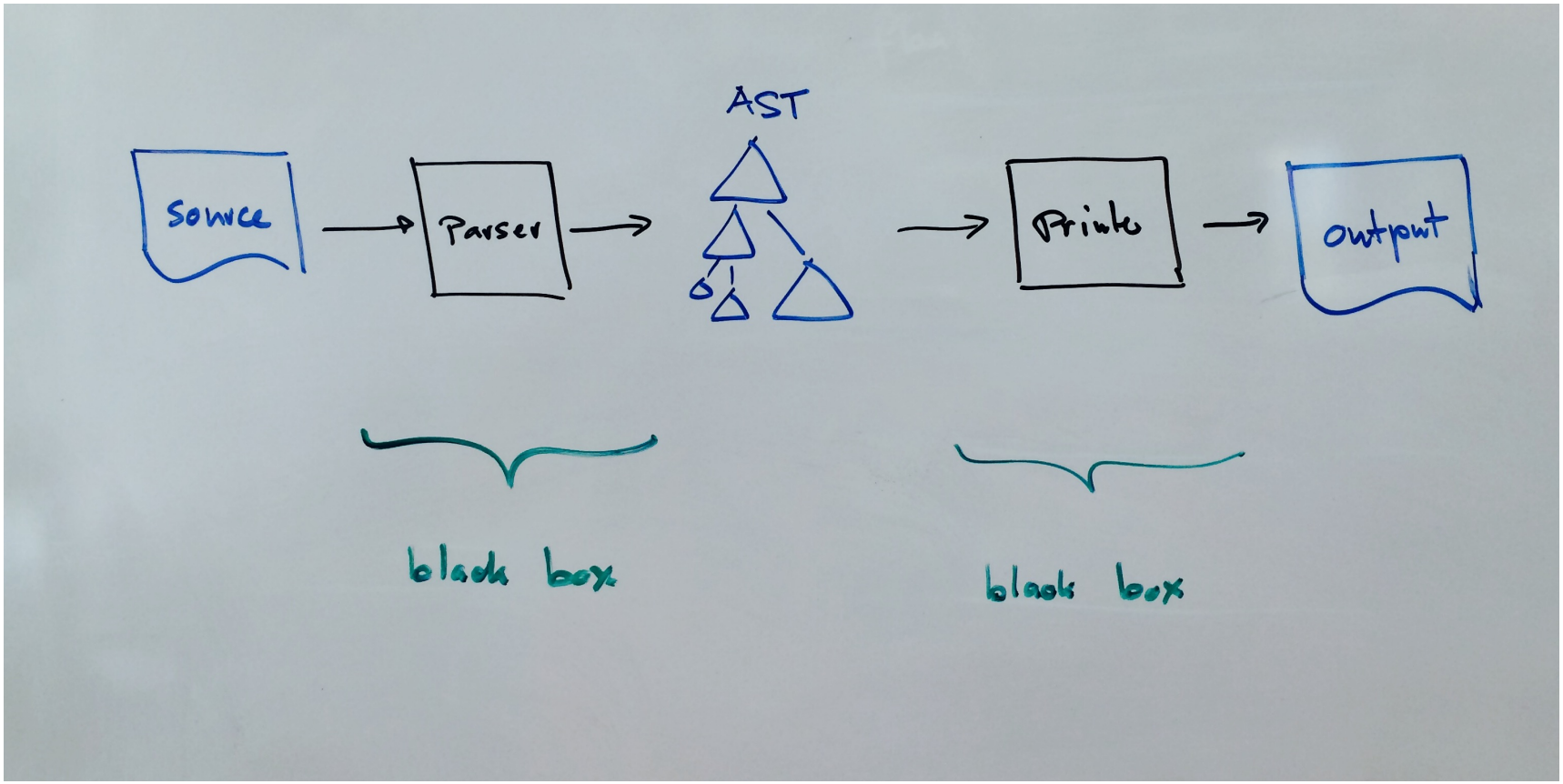
对于固定宽度的字体，处理的很好.

比例大小的字体也可以被编辑器支持，如果这个编辑器可以支持灵活的制表符宽度.

综合在一起 (2)


AST + comments
token stream
Source
Parser
Comment stream
merged token stream
Merger
tabwriter
formatted source
output

从宏观上看



gofmt 的应用

gofmt 作为源代码变换工具

- 改写 Go 的代码 (Russ Cox)， `gofmt -r`

```
gofmt -w -r 'a[i:len(x)] -> a[i:]' *.go
```

- 简化 Go 的代码， `gofmt -s`
- 更新 API (Russ Cox), `go fix`
- 改变语言 （去掉分号，其它）
- goimport (Brad Fitzpatrick)

大家的反应

- Go 项目要求所有提交的源代码都用 gofmt 的格式。
- 一开始，大家都抱怨：`gofmt` 不知道怎样格式成我的风格！
- 慢慢地，大家不作声了：Go 项目组一定要用 gofmt！
- 最后，大家看清了：gofmt 不是任何人的风格，但所有人都喜欢 gofmt 的风格。
- 现在，大家都赞扬： gofmt 是大家喜欢 Go 的一个原因。

现在，格式已经不是一个问题。

其它语言也在向我们学习

- Google 的 BUILD 文件现在也有格式器 (Russ Cox).
- Java 格式器
- Clang 格式器
- Dartfmt

  www.dartlang.org/tools/dartfmt/ (https://www.dartlang.org/tools/dartfmt/)

- 等等

现在，任何语言都被要求带有自动的源代码格式器。

总结

编程文化的演变

- gomft 是 Go 语言的一个重要的卖点
- 大家渐渐达成共识：一致的“足够好“的格式很有好处
- 这种在 AST-级别上的源代码操作带动了一系列的新的工具。
- 其它语言也在向我们学习：编程的文化在慢慢演变。

至今的收获：应用程序

- 一开始，基本的源代码格式化是一个很好的目标。
- 但是，真正的用处在于源代码的变换工具。
- 不要给大家有选择格式的机会。
- 越简单越好。

我们想要：

- Go 分析器：源代码 => 语法树
- 尽可能让语法树的操作变得容易。
- Go 打印器：语法树 => 源代码

至今的收获：实现过程

- 最初的版本有很多的尝试和失败。
- 最大的错误：注释没有连到 AST-节点上.

=> 现在的设计使得操作 AST 和保持注释在正确的地方十分困难。

- 很混乱：ast.CommentMap

我们想要：

- 容易操作语法树，连带注释。

将来的计划

- 正在设计新的语法树（仍在试验阶段）
- 语法树操作起来更加简单和容易（例如：声明结点）
- 更快和更容易地使用分析器和打印器。
- 让工具用起来可靠并且快。其它一概不理。

Thank you

Robert Griesemer
Google, Inc.
gri@golang.org (mailto:gri@golang.org)