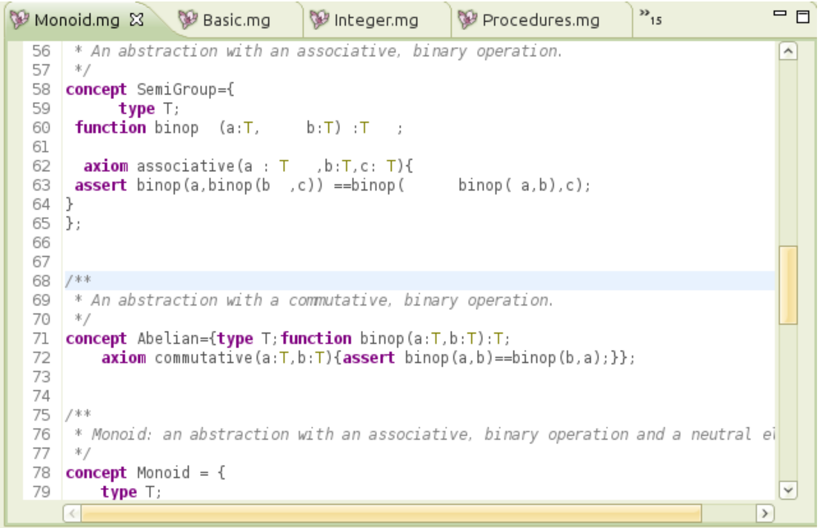
A Pretty Good Formatting Pipeline

Anya Helene Bagge and Tero Hasu

University of Bergen, Norway

SLE'13

Problem



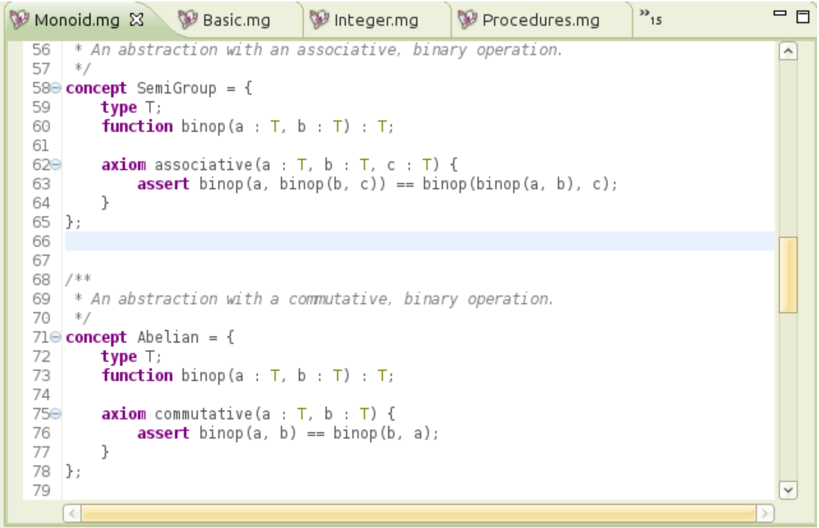
```
 * An abstraction with an associative, binary operation.
 */
concept SemiGroup={
      type T;
 function binop  (a:T,     b:T) :T   ;

  axiom associative(a : T   ,b:T,c: T){
 assert binop(a,binop(b  ,c)) ==binop(      binop( a,b),c);
}
};


/**
 * An abstraction with a commutative, binary operation.
 */
concept Abelian={type T;function binop(a:T,b:T):T;
    axiom commutative(a:T,b:T){assert binop(a,b)==binop(b,a);}};


/**
 * Monoid: an abstraction with an associative, binary operation and a neutral e
 */
concept Monoid = {
    type T;
```

Solution



```
 * An abstraction with an associative, binary operation.
 */
concept SemiGroup = {
    type T;
    function binop(a : T, b : T) : T;

    axiom associative(a : T, b : T, c : T) {
        assert binop(a, binop(b, c)) == binop(binop(a, b), c);
    }
};


/**
 * An abstraction with a commutative, binary operation.
 */
concept Abelian = {
    type T;
    function binop(a : T, b : T) : T;

    axiom commutative(a : T, b : T) {
        assert binop(a, b) == binop(b, a);
    }
};

```

Observations

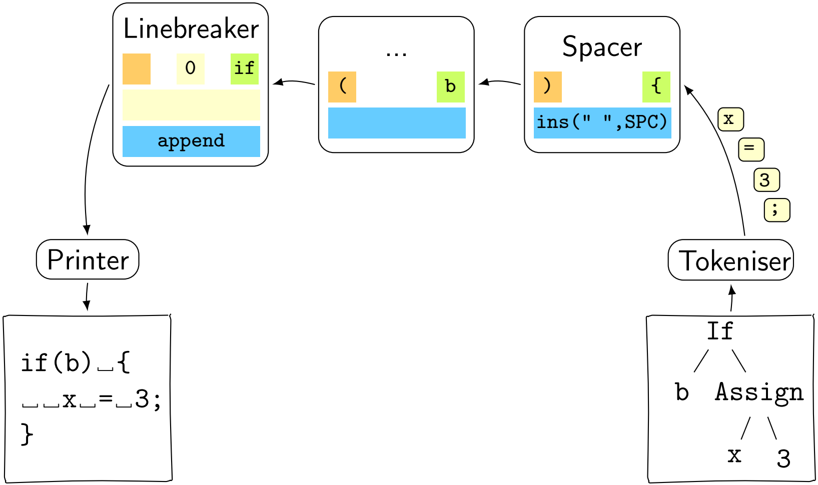
Good code formatting encompasses multiple concerns:

- Inter-word (horizontal) spacing
- Line breaking
- Vertical spacing
- Indentation
- Colouring

Rules differ according to user preference

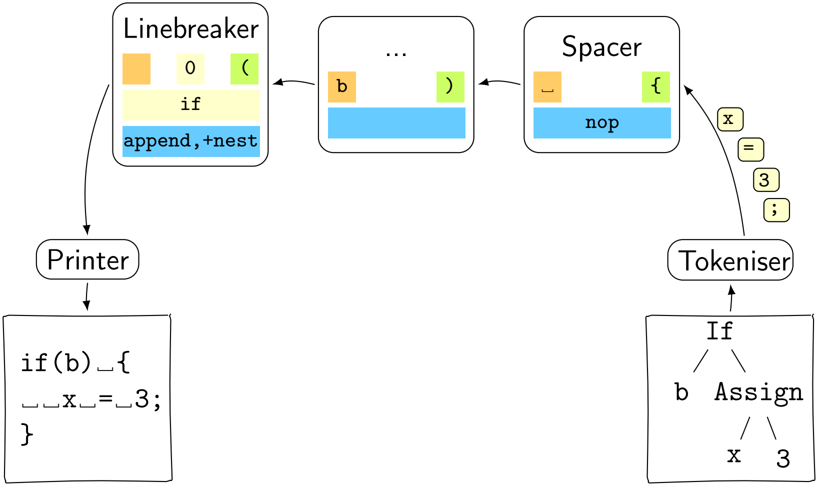
Many languages have similar rules

Architecture



```
if(b)␣{
␣␣x␣=␣3;
}
```

Architecture



Linebreaker

0 (

if

append,+nest

…

b)

Spacer

␣ {

nop

x

=

3

;

Printer

```
if(b)␣{
␣␣x␣=␣3;
}
```

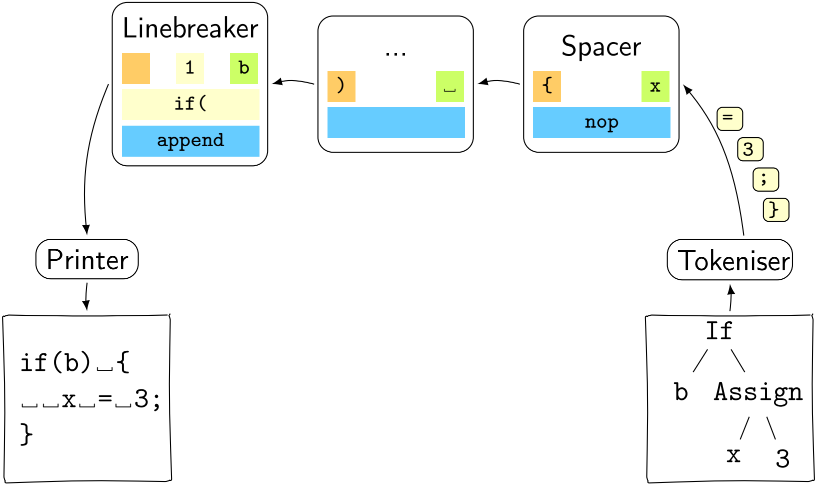
Tokeniser

If

b Assign

x 3

Architecture



Linebreaker

1 b

if(

append

...

) ␣

Spacer

{ x

nop

=

3

;

}

Tokeniser

If

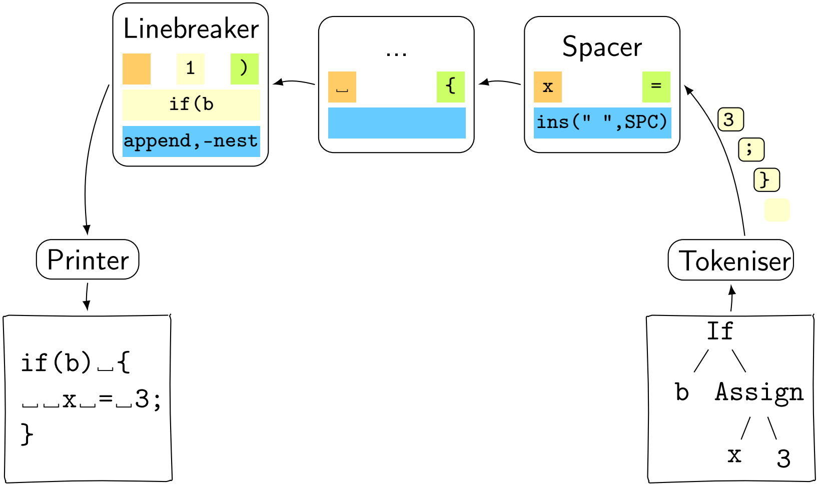
b Assign

x 3

Printer

```
if(b)␣{
␣␣x␣=␣3;
}
```

Architecture



Linebreaker

1)

if(b

append,-nest

…

␣ {

Spacer

x =

ins(" ",SPC)

3

;

}

Printer

Tokeniser

If

b Assign

x 3

```
if(b)␣{
␣␣x␣=␣3;
}
```

In this Talk

- Tokens, categories and token processors
- Spacing
- Indentation and Line-Breaking
- Plumbing

Token Stream Processors

- Formatter is divided into token processors
- Processors are connected in a pipeline
- Inputs and outputs are streams of tokens
- Reconfigurable:
    - Spacing, indentation and line breaking
    - Just fix spaces, don't touch line breaks
    - Just do indentation, don't touch other spaces
    - Just break lines and indent, don't touch spaces
    - ...

Categorising Tokens

- Decisions are made based on token categories

  if (*b*) ␣ { \n *x* =
  3 ; \n }

- Every token belongs to one category
- That category may give membership in other (super)categories

Categorising Tokens

- Decisions are made based on token categories

  if:KW   *(*:LPAR   *b*:ID   *)*:RPAR   ␣:SPC   *{*:LBRC   \n:NL   *x*:ID   *=*:OP
  3:NUM   *;*:SEMI   \n:NL   *}*:RBRC

- Every token belongs to one category
- That category may give membership in other (super)categories

Token Hierarchy

- For example, the category of { is LBRC:
  - Any LBRC is also a BRC and a LGRP.
  - Any BRC and LGRP is also a GRP.
  - Any non-space token is a member of TXT.
  - All tokens are members of TOKEN.

- Used in formatting rules:
  - LGRP increases nesting, RGRP decreases
  - Break line after/before LBRC/RBRC
  - Always space around BINOP
  - No space after/before LGRP/RGRP

Control Tokens

- May also use control tokens
    - Begin/end of nested expressions
    - Switch formatting rule sets (for different languages)
    - Indentation control (e.g., indent to level of opening paren)

Tokenising Parse Trees

- A full parse tree contains both lexical and structural information
    - All you need for beautiful formatting!
- Transforming to a token stream is easy
    - categorise based on sorts (from grammar), regexes, hand-implemented rules
    - can include structural info (e.g., expression nesting level)
    - could also include extra goodies (e.g., type annotations)
- We can auto-tokenise parse trees in UPTR (Rascal) and AsFix2 (SDF2/SGLR) formats
    - Language-specific tuning categorise tokens

Example: Tokenisation Config for Java-like Language

- Nesting non-terminal sorts: Expr, Stat, Decl*
- Identifiers (ID) look like: `[_a-zA-Z][_a-zA-Z0-9]*`
- Numbers (NUM) look like: `[0-9]+`
- Alphabetic literal strings are keywords (KW)
- Any non-space layout is a comment (COM)
- Parens, braces, bracket and punctation follow normal rules

Spacing

- The spacer is a token processor
- Goal: insert/remove horizontal space according to rules
- For example:

```
axiom cutSalaries ( c:Company , n:Name ){
    assert salaryOf( findEmployee( cut(c),n))
      == halve(salaryOf(findEmployee(c,n))); }
```

  to

```
axiom cutSalaries(c : Company, n : Name) {
    assert salaryOf(findEmployee(cut(c), n))
      == halve(salaryOf(findEmployee(c, n))); }
```

- Can be done using simple rule-based automaton
  - Looking at previous token, and next 1–2 tokens

Spacing Rules

- First, remove all existing spaces
- Then, for each token, decide whether to insert space before it:
  - No spaces on the inner side of parentheses:

    ```
    addRule(after(LPAR), nop);
    addRule(at(PAR), nop);
    ```

  - Always (or never) space between an `if` and the parenthesis:

    ```
    addRule(after(IF).at(LPAR), space);
    ```

  - Always space after a comma, never before:

    ```
    addRule(at(COMMA), nop);
    addRule(after(COMMA), space);
    ```

  - ...
  - Fallback: Always spaces between any non-space tokens:

    ```
    addRule(after(TXT).at(TXT), space);
    ```

- Rules for different languages seem similar. Sharing possible?

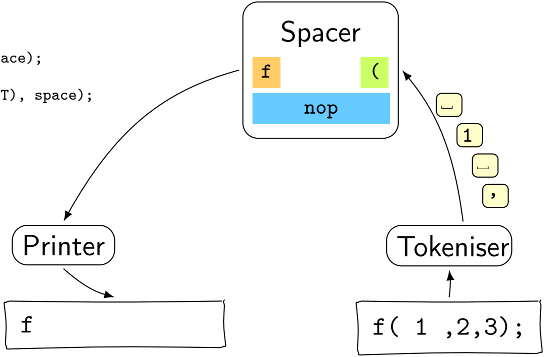
Spacing Example

```
addRule(at(SPC), delete);

addRule(after(LPAR), nop);
addRule(at(PAR), nop);

addRule(at(COMMA), nop);
addRule(after(COMMA), space);

addRule(after(TXT).at(TXT), space);
```



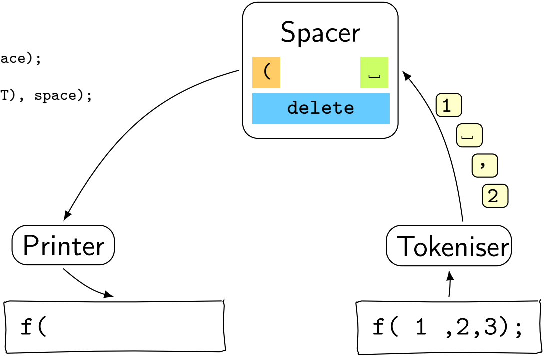
Spacing Example

```
addRule(at(SPC), delete);

addRule(after(LPAR), nop);
addRule(at(PAR), nop);

addRule(at(COMMA), nop);
addRule(after(COMMA), space);

addRule(after(TXT).at(TXT), space);
```



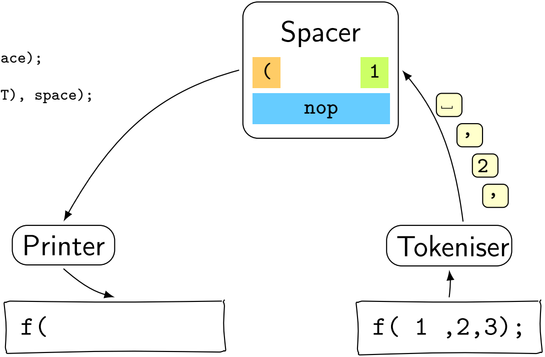
Spacing Example

```
addRule(at(SPC), delete);

addRule(after(LPAR), nop);
addRule(at(PAR), nop);

addRule(at(COMMA), nop);
addRule(after(COMMA), space);

addRule(after(TXT).at(TXT), space);
```



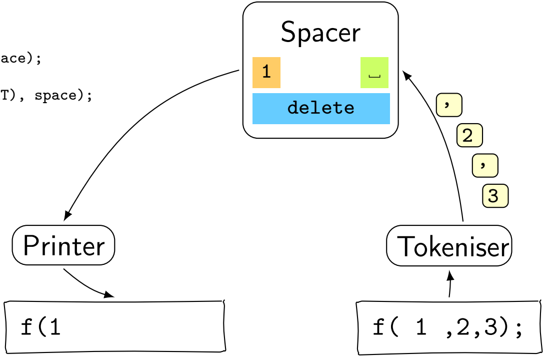
Spacing Example

```
addRule(at(SPC), delete);

addRule(after(LPAR), nop);
addRule(at(PAR), nop);

addRule(at(COMMA), nop);
addRule(after(COMMA), space);

addRule(after(TXT).at(TXT), space);
```



Spacer

1

␣

delete

,

2

,

3

Printer

Tokeniser

f(1

f(1 ,2,3);

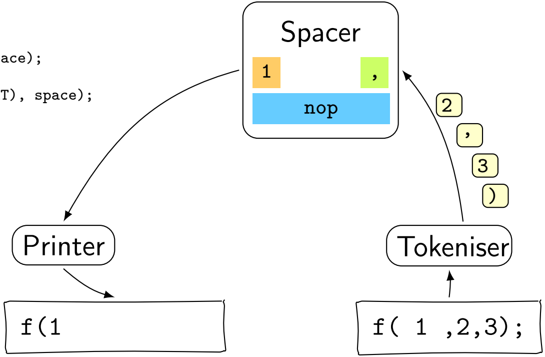
Spacing Example

```
addRule(at(SPC), delete);

addRule(after(LPAR), nop);
addRule(at(PAR), nop);

addRule(at(COMMA), nop);
addRule(after(COMMA), space);

addRule(after(TXT).at(TXT), space);
```



.

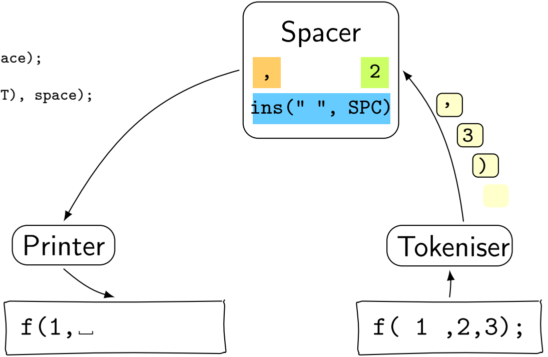
Spacing Example

```
addRule(at(SPC), delete);

addRule(after(LPAR), nop);
addRule(at(PAR), nop);

addRule(at(COMMA), nop);
addRule(after(COMMA), space);

addRule(after(TXT).at(TXT), space);
```



Line Breaking

- Insert newlines so that all lines fit within some constraint
- Tangled with indentation
- Issues:
  - Fill as much of the line as possible
  - Keep related things on the same line
  - Make code nesting structure easy to see

Indentation

Four ways of controlling indentation:

- Increase Level: normal nesting (in/out)
- Add String: e.g., for breaking line comments
- Absolute Level: e.g., put `#ifdef` in column 0
- Relative Level: e.g., indent to level of last paren

Indentation control can be done as a separate step; indentation itself must be done together with line breaking (if any)

Line Breaking Algorithms

Experiments:

- Wadler's algorithm adapted to streams
- Kiselyov's stream-oriented linear, backtracking-free algorithm
- Our own linear, backtracking-free algorithm
  - discourage breaking at deeply nested points:

x = a * b + c / d + c / d * f + c / d;

```
x = a * b
+ c / d
+ (c / d * f)
+ c / d;
```

```
x = a * b + c
/ d + (c / d
* f) + c / d;
```

Conclusions:

- We don't know which one is best (yet)

Line Breaking Algorithms

Experiments:

- Wadler's algorithm adapted to streams
- Kiselyov's stream-oriented linear, backtracking-free algorithm
- Our own linear, backtracking-free algorithm
  - discourage breaking at deeply nested points:

    x = a * b + c / d + c / d * f + c / d;

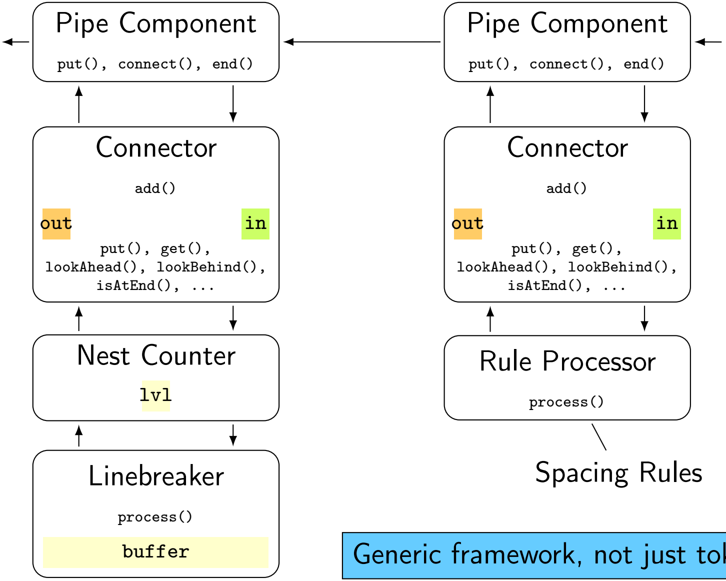
```
x = a * b
+ c / d
+ (c / d * f)
+ c / d;
```

```
x = a * b + c
/ d + (c / d
* f) + c / d;
```

Conclusions:

- We don't know which one is best (yet)

Plumbing for Stream-Based Systems



Pipe Component

put(), connect(), end()

Connector

add()

out

in

put(), get(), lookAhead(), lookBehind(), isAtEnd(), ...

Nest Counter

lvl

Linebreaker

process()

buffer

Pipe Component

put(), connect(), end()

Connector

add()

out

in

put(), get(), lookAhead(), lookBehind(), isAtEnd(), ...

Rule Processor

process()

Spacing Rules

Generic framework, not just tokens!

Status

- Spacing: Works well, needs config system for user control
- Indentation and line breaking: Experimental
- Performance: dominated by parsing and tokenisation
- Code is on GitHub!

Summary

- Code formatting based on token stream processors
- Separation of concerns
  - One processor for each formatting concern
  - Can be plugged together in different ways
- Compatible with Stratego, Rascal, [your system here?]
- Tested on Magnolia and Java code
- Basis for further experimentation

Get it here:
https://github.com/nuthatchery/pgf