Specifying a layered presentation-oriented editor

Martijn Schrage

Utrecht University

26-1-2005

Overview

- Proxima
  - Introduction to architecture
  - Extra state
  - Demo

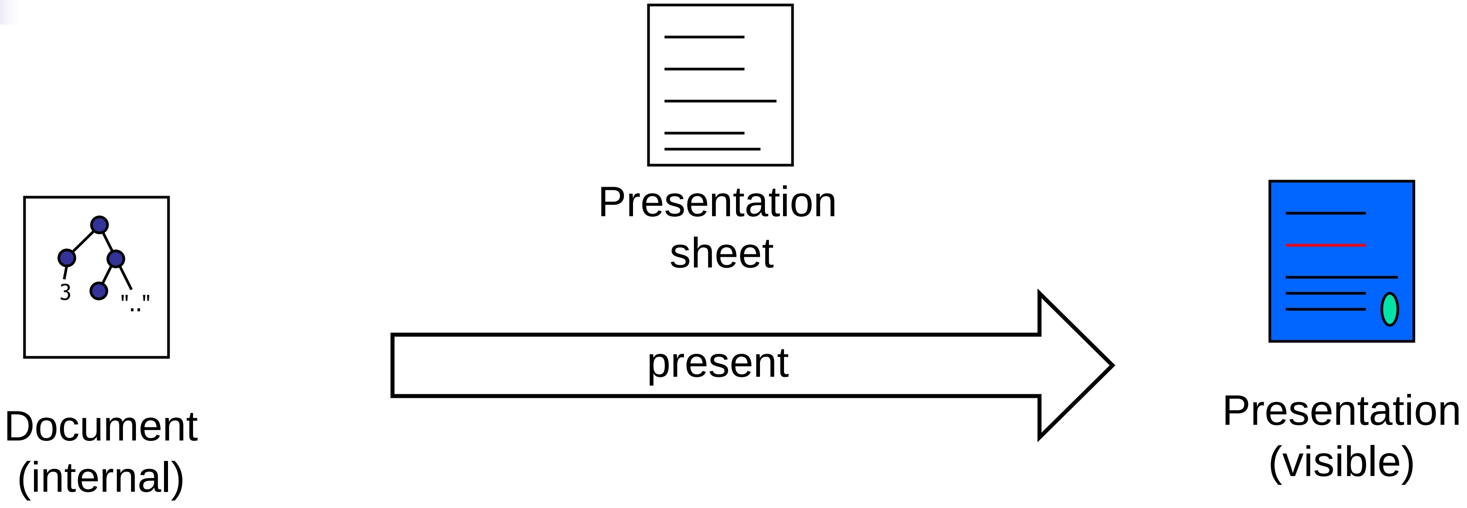
- Specification
  - Chapter 5 of

    Martijn M. Schrage.

    Proxima - A presentation-oriented editor for structured documents. PhD thesis, Utrecht University, 2004.
  - Joint work with Lambert Meertens
  - Stepwise

1. Proxima

Edit model



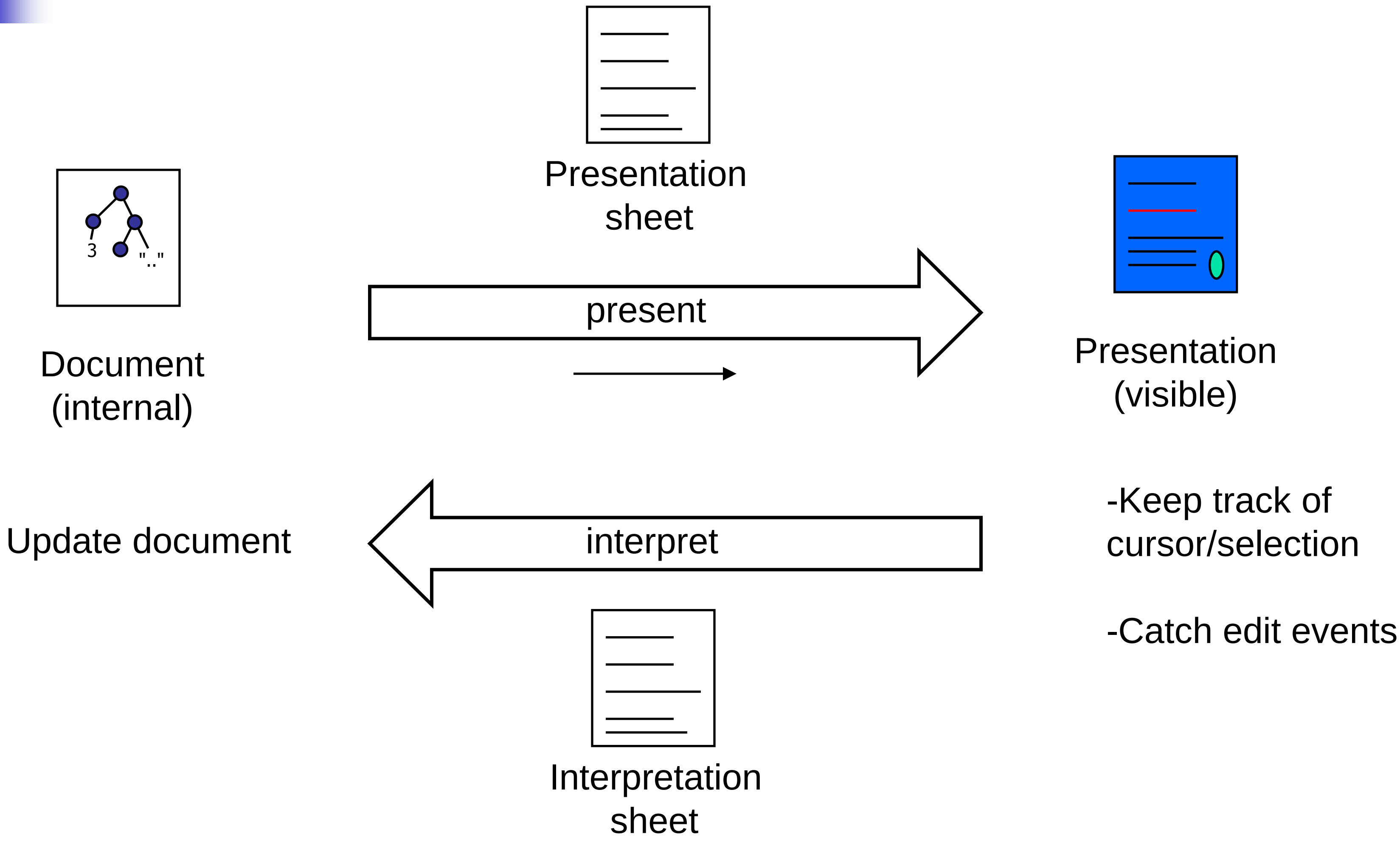
Document
(internal)

Presentation
sheet

present

Presentation
(visible)

Edit model



Presentation sheet

Document (internal)

present

Presentation (visible)

Update document

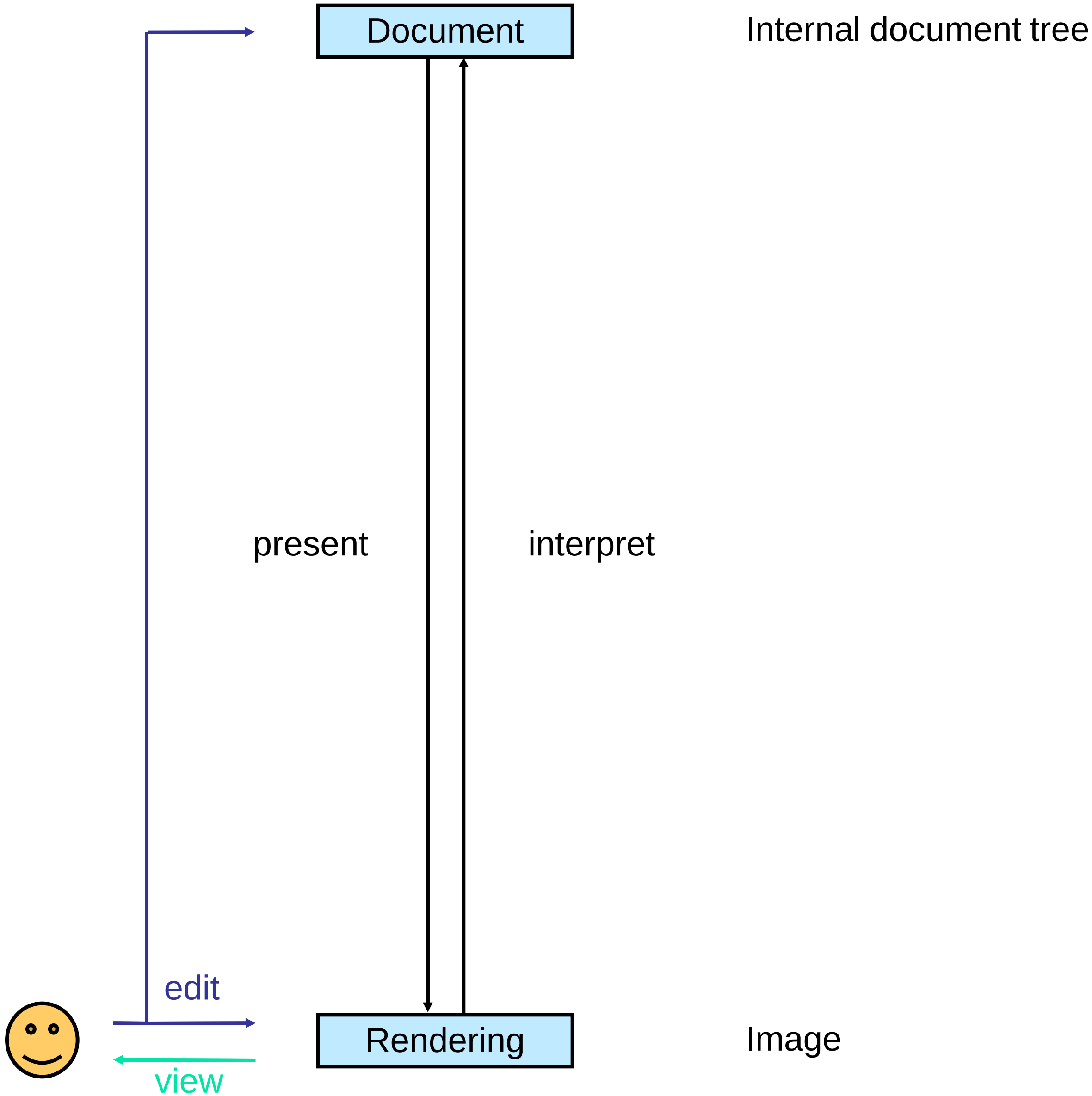
interpret

-Keep track of cursor/selection

-Catch edit events

Interpretation sheet

Proxima architecture



Document

Internal document tree

present

interpret

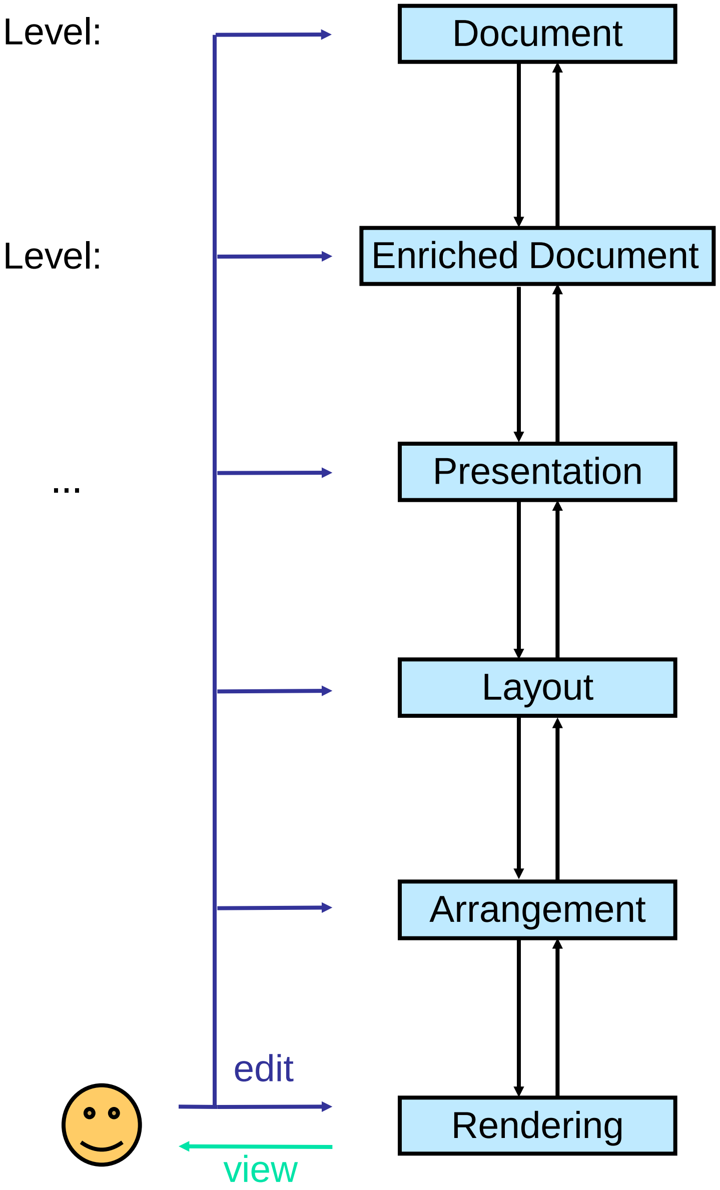
edit

view

Rendering

Image

Proxima architecture: Levels



Level: Document

Internal document tree

Level: Enriched Document

Document + derived values

... Presentation

Logical presentation: rows, columns,
tokens, etc.

Layout

Presentation + explicit whitespace

Arrangement

Presentation with exact positions & sizes

edit

view

Rendering

Image

Presentation process

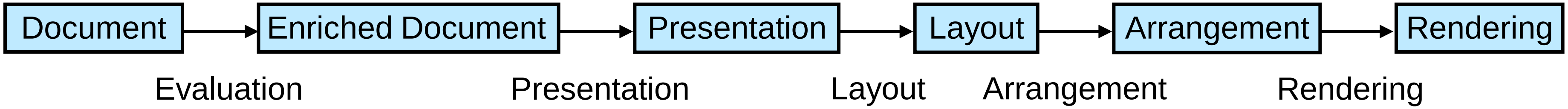
Document:

```
Root [ Comment ["This", "is", "a", "simple", "expression"]
     , Decl "simple1"
            (IfExp (BoolExp True) (IntExp 1) (IntExp 0))
     ]
```

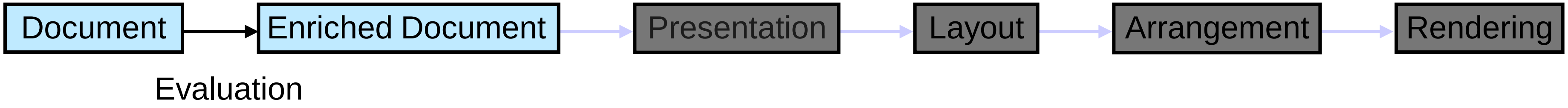
Rendering:

This is a simple expression

```
simple1 :: Int
simple1 =
  if True then 1
          else 0
```


Document → Enriched Document → Presentation → Layout → Arrangement → Rendering

Evaluation, Presentation, Layout, Arrangement, Rendering



Document → Enriched Document → Presentation → Layout → Arrangement → Rendering

Evaluation

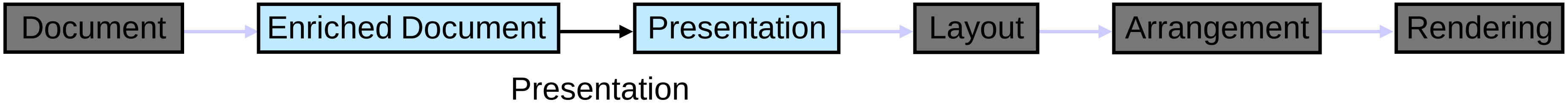
Evaluation

Document:

```
Root [ Comment ["This", "is", "a", "simple", "expression"]
     , Decl "simple1"
            (IfExp (BoolExp True) (IntExp 1) (IntExp 0))
     ]
```

Enriched Document:

```
Root [ Comment ["This", "is", "a", "simple", "expression"]
     , TypeDecl "simple1" IntType
     , Decl "simple1"
            (IfExp (BoolExp True) (IntExp 1) (IntExp 0))
     ]
```


Document → Enriched Document → Presentation → Layout → Arrangement → Rendering

Presentation

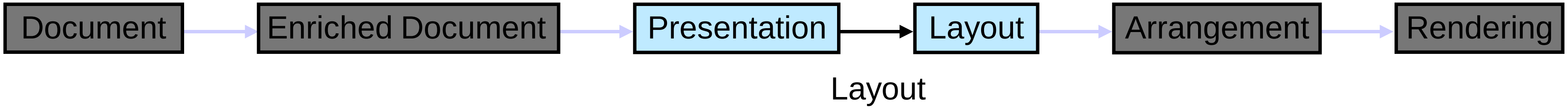
Presentation

Enriched Document:

```
Root [ Comment ["This", "is", "a", "simple", "expression"]
     , TypeDecl "simple1" IntType
     , Decl "simple1"
              (IfExp (BoolExp True) (IntExp 1) (IntExp 0))
     ]
```

Presentation:

```
Col [ With {font}
       (Formatter [ "This", "is", "a", "simple", "expression" ])
    , With {font}
       (Tokens [ Token (1,0) "simple1", Token (0,1) "::"
               , Token (0,1) "Int" ])
    , With {font}
       (Tokens [ Token (1,0) "simple1", Token (0,1) "="
               , Token (1,2) "if", ... ]) ]
```



Document
Enriched Document
Presentation
Layout
Arrangement
Rendering
Layout

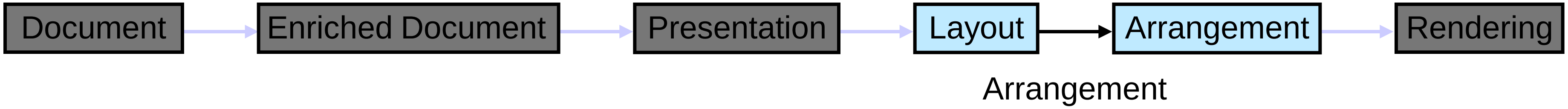
Layout

Presentation:

```
Col [ ...
    , With {font}
      (Tokens [ Token (1,0) "simple1", Token (0,1) "::"
              , Token (0,1) "Int" ])
    , With {font}
      (Tokens [ Token (1,0) "simple1", Token (0,1) "="
              , Token (1,2) "if", ... ]) ]
```

Layout:

```
Col [ ...
    , With {font}
      (Col [ ""
           , Row [ "simple1", " ", "::", " ", "Int" ] ])
    , With {font}
      (Col [ Row [ "simple1", " ", "=" ]
           , Row [ "  ", "if", " ", "True", " ", "then" ]
           , Row [ "           ", "else", ... ] ) ]
```



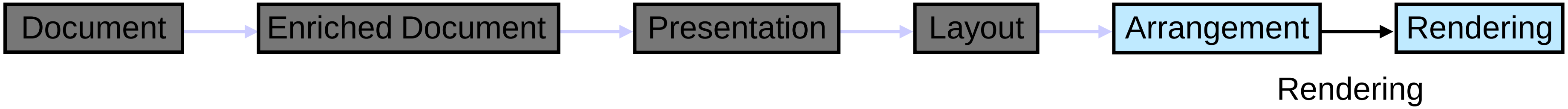
Arrangement

Layout:

```
Col [ With {font}
      (Formatter [ "This", "is", "a", "simple", "expression" ])
    , With {font}
      (Col [ ""
           , Row [ "simple1", " ", "::", " ", "Int" ] ])
    , With {font}
      (Col [ Row [ "simple1", " ", "=" ], ... ]) ]
```

Arrangement:

```
Col(0,0)(80×84)
   [ Col(0,0)(80×24)   [ Row(0,0)(80×12)  [ "This"(0,0)(17×12), "is"(25,0)(6×12), ... ]
                       , Row(0,12)(80×12) [ "expression"(53,0)(27×12)  ]
   , Col(0,24)(75×24)  [ ""(0,0)(0×12)
                       , Row{..}  [ "simple1"{..}, " "{..}, "::"{..}, ... ] ]
   , Col(0,48)(80×36)  [ Row{..}  [ "simple1"{..}, " "{..}, "="{..} ]
                       , ... ] ]
```



Rendering

Arrangement:

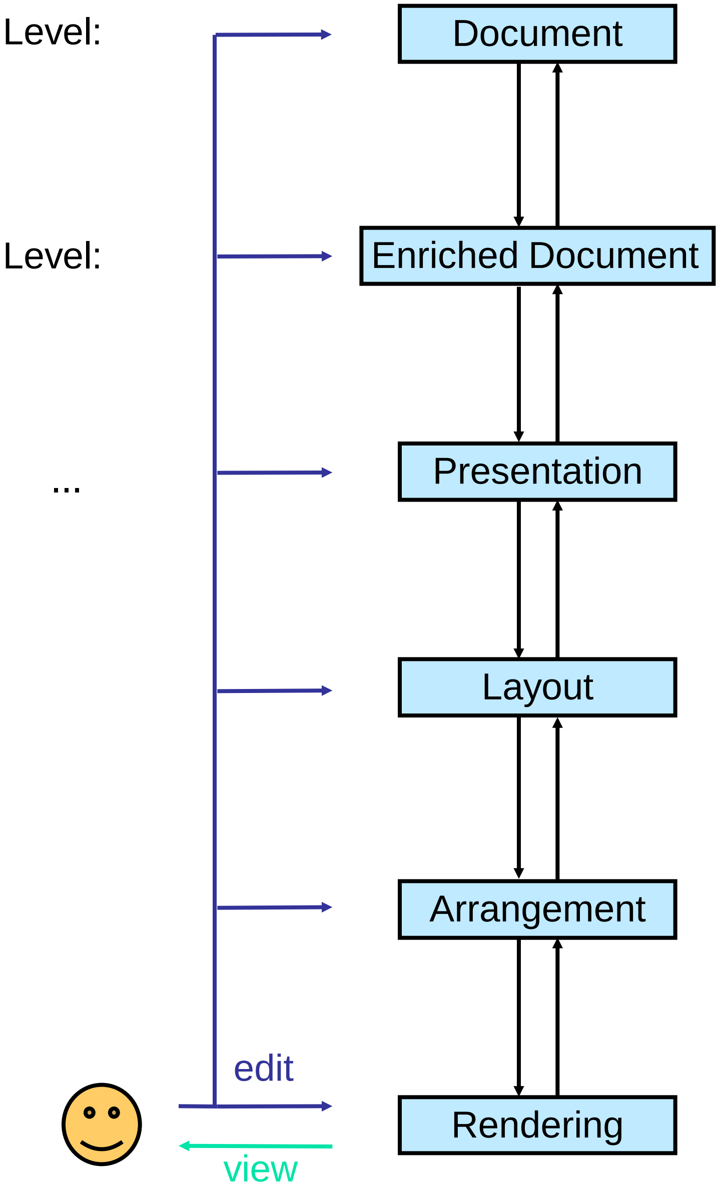
```
Col(0,0)(80×84)
     [ Col(0,0)(80×24)    [ Row(0,0)(80×12)  [ "This"(0,0)(17×12),  "is"(25,0)(6×12),  ... ]
                          , Row(0,12)(80×12) [ "expression"(53,0)(27×12)  ]
     , Col(0,24)(75×24)   [ Row{..}  [ ]
                          , Row{..}  [ "simple1"{..}, " "{..}, "::"{..}, ... ] ]
     , Col(0,48)(80×36)   [ Row{..}  [ "simple1"{..}, " "{..}, "="{..} ]
                          , ... ] ]
```

Rendering:

This is a simple
expression

```
simple1 :: Int
simple1 =
  if True then 1
          else 0
```

Proxima architecture: levels



Level:

Document

Internal document tree

Level:

Enriched Document

Document + derived values

...

Presentation

Logical presentation: rows, columns, tokens, etc.

Layout

Presentation + explicit whitespace

Arrangement

Presentation with exact positions & sizes

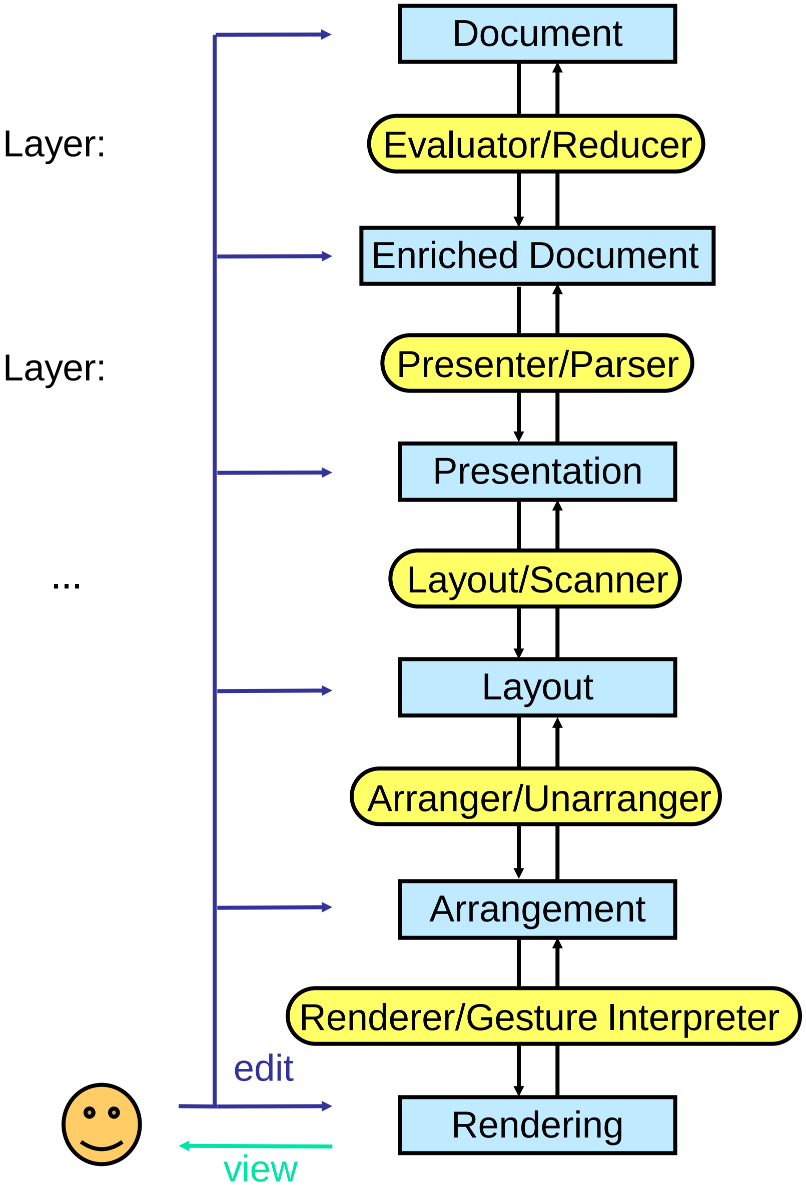
edit

view

Rendering

Image

Proxima architecture: layers



Document — Internal document tree

Layer: Evaluator/Reducer

Enriched Document — Document + derived values

Layer: Presenter/Parser

Presentation — Logical presentation: rows, columns, tokens, etc.

... Layout/Scanner

Layout — Presentation + explicit whitespace

Arranger/Unarranger

Arrangement — Presentation with exact positions & sizes

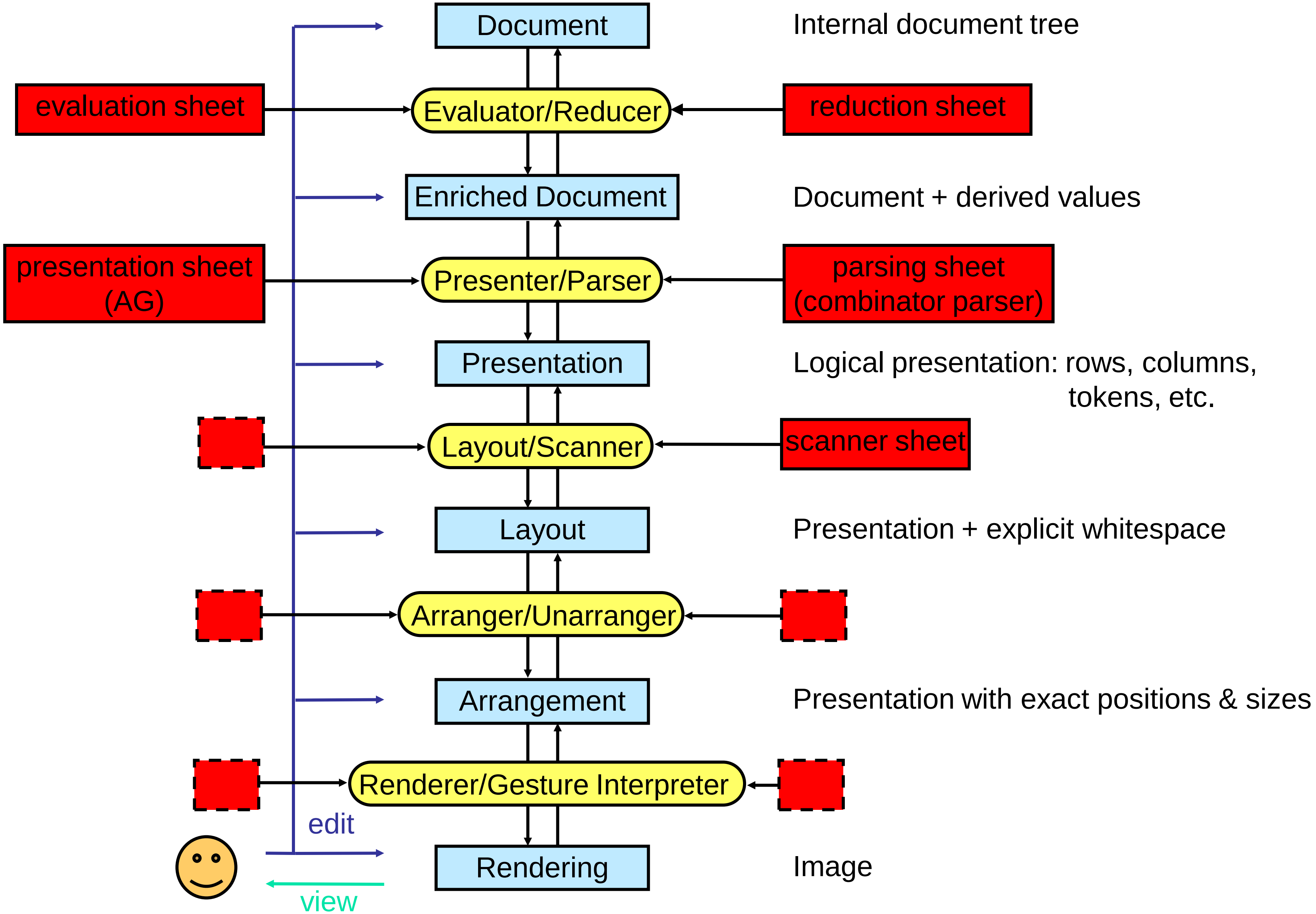
Renderer/Gesture Interpreter

Rendering — Image

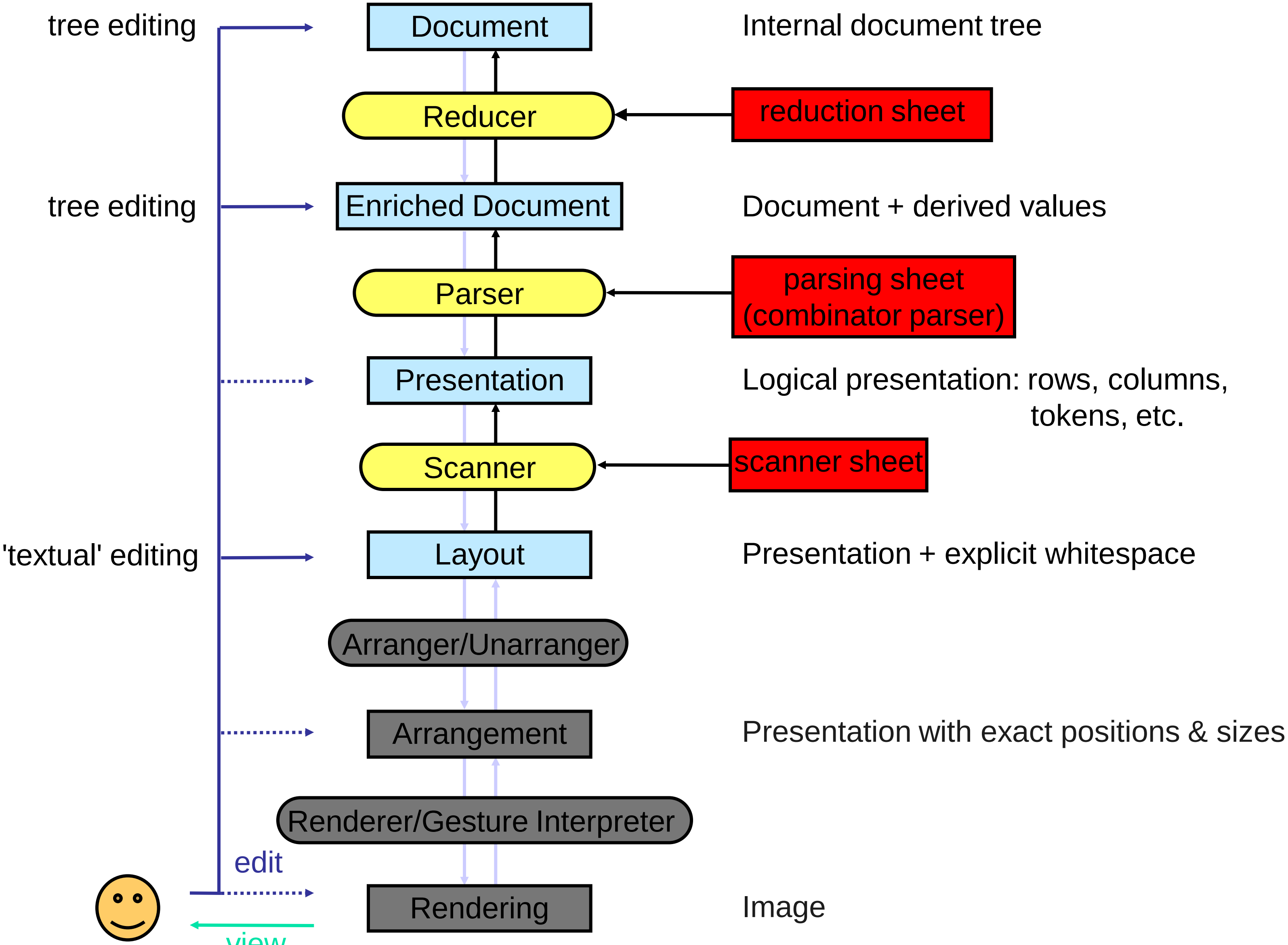
edit

view

Proxima architecture: sheets



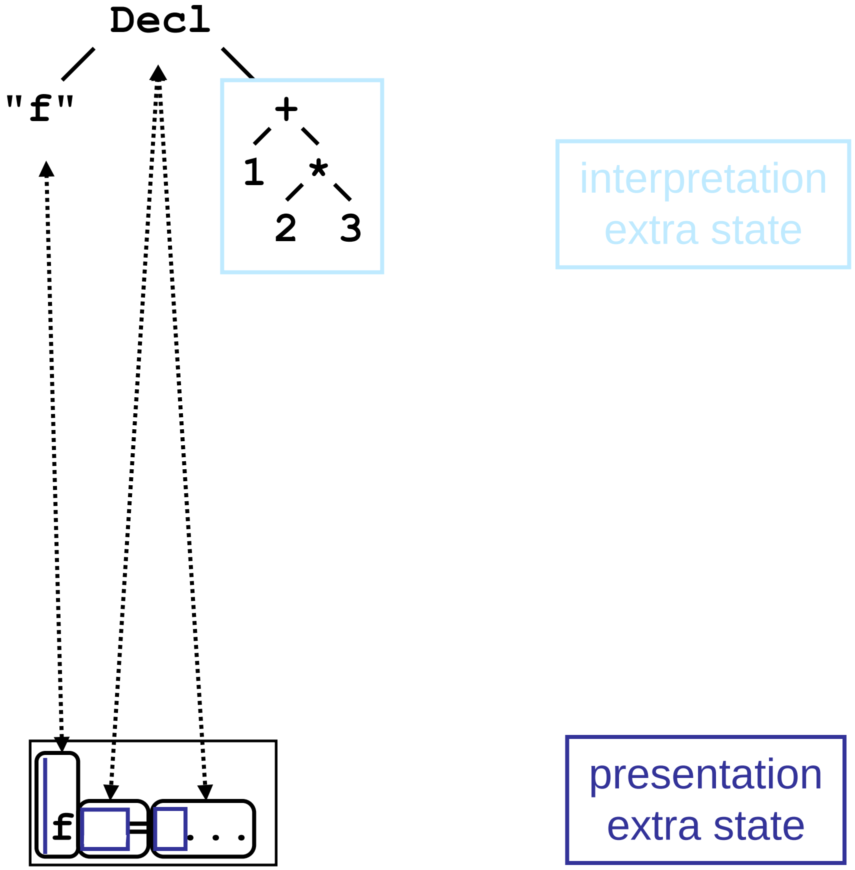
Interpretation process



Extra state (simplified)

Document:

Presentation:

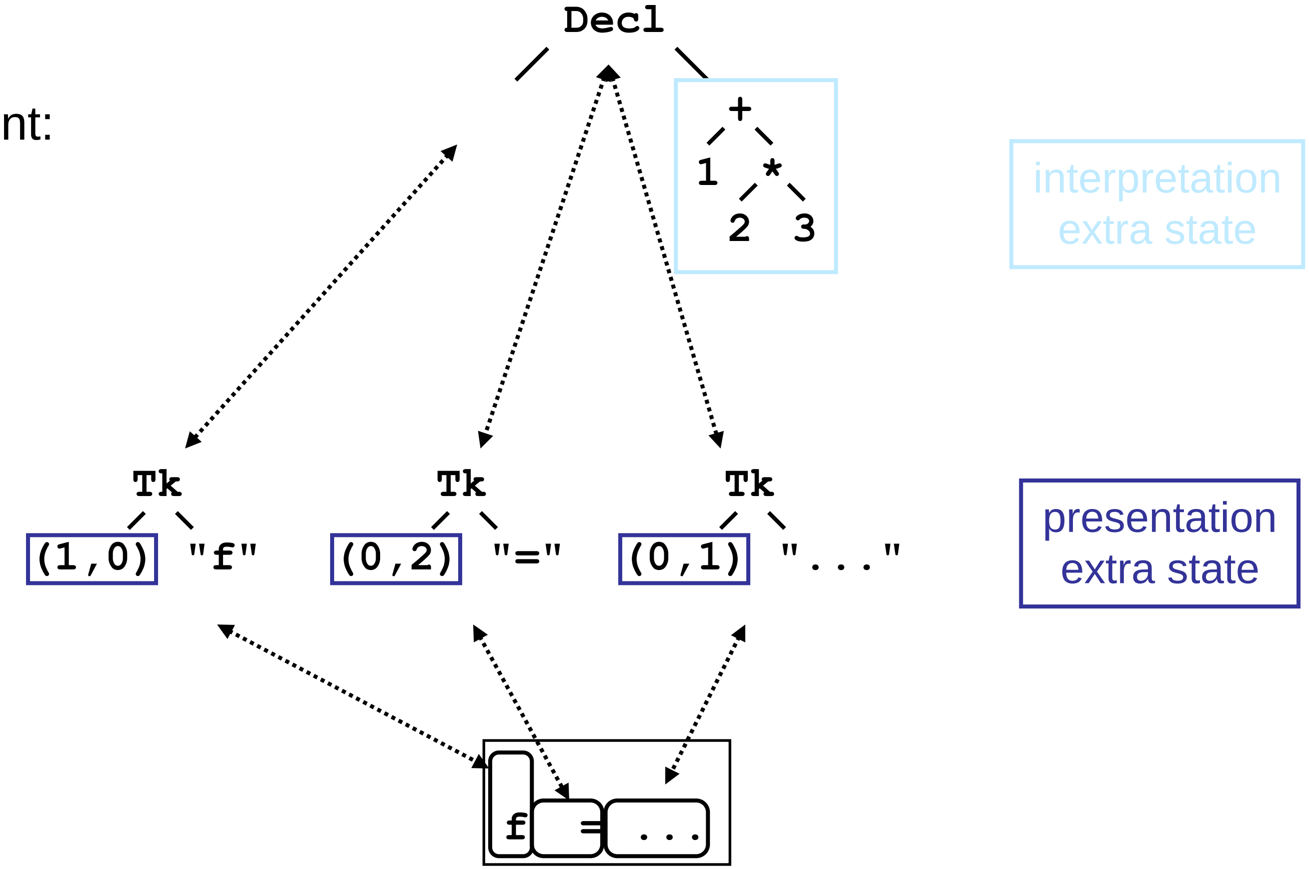


Extra state (actual)

Enriched document:

Presentation:

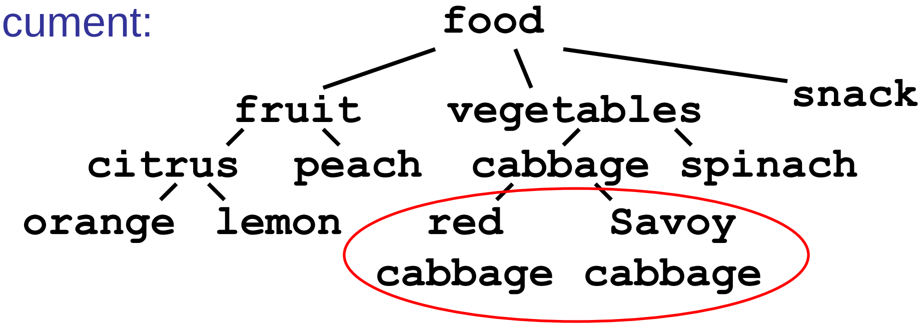
Layout:



Extra state, other examples

- Tree view

Document:



food

fruit vegetables snack

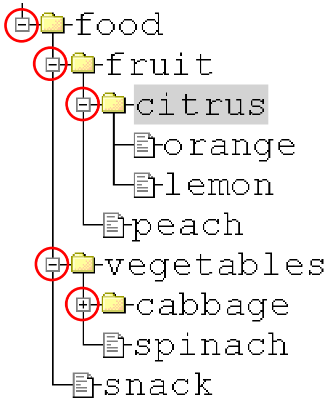
citrus peach cabbage spinach

orange lemon red cabbage Savoy cabbage

present

extra state

Presentation:



```
⊟food
  ⊟fruit
    ⊟citrus
       orange
       lemon
     peach
  ⊟vegetables
    ⊞cabbage
     spinach
   snack
```

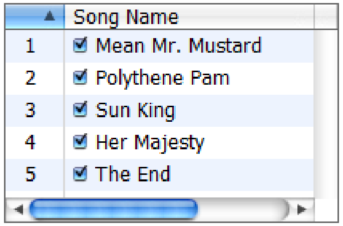
- List order

Document:

```
[ "Sun King"
, "Mean Mr. Mustard"
, "Polythene Pam"
, "The End"
, "Her Majesty"
]
```

present

Presentation:



▲	Song Name
1	☑ Mean Mr. Mustard
2	☑ Polythene Pam
3	☑ Sun King
4	☑ Her Majesty
5	☑ The End

extra state: list order

Demo

- Research prototype
  - Implemented in Haskell (with GUI in wxHaskell)
  - Attribute grammar for presentation
  - Parser combinators for interpretation
- Four editors instantiated
  - Program source (Helium)
  - Slide presentations (a la PowerPoint)
  - Chess board
  - Editor for reversible language Inv
- No optimisations yet
  - On each key press, every pixel is recomputed

DTD - Helium (partial)

concrete
syntax

```
data Document = RootDoc decls:[Decl]

data Decl  = Decl Ident Exp              -- <ident> = <exp>;
           | BoardDecl Board             -- Chess: <board>
           | SlidesDecl Slides           -- Slides: <slides>

data Ident = Ident String

data Exp   = PlusExp  exp1:Exp exp2:Exp  -- <exp> + <exp>
           | TimesExp exp1:Exp exp2:Exp  -- <exp> * <exp>
           | DivExp   exp1:Exp exp2:Exp  -- <exp> / <exp>
           | AppExp   exp1:Exp exp2:Exp  -- <exp> <exp>
           | LamExp   Ident Exp          -- \<ident> → <exp>
           | LetExp   [Decl] Exp         -- let <decls> in <exp>
           | BoolExp  Bool               -- True
           | IntExp   Int                -- 173
           | IdentExp Ident              -- x
           | ...
```

+ ~1500 lines of sheet code

DTD - Slide editor

```
data Slides    = Slides viewType:Bool [Slide]

data Slide     = Slide title:String ItemList

data ItemList = ItemList ListType items:[Item]

data ListType = Bullet
               | Number
               | Alpha

data Item      = StringItem string:String
               | HeliumItem Exp
               | ListItem   ItemList
```

`+ ~200 lines of sheet code`

DTD - Chess

```
data Board = Board r1:Row r2:Row r3:Row r4:Row
                   r5:Row r6:Row r7:Row r8:Row

data Row = Row ca:Square cb:Square cc:Square cd:Square
               ce:Square cf:Square cg:Square ch:Square

data BoardSquare = Queen  color:Bool
                 | King   color:Bool
                 | Bishop color:Bool
                 | Knight color:Bool
                 | Rook   color:Bool
                 | Pawn   color:Bool
                 | Empty
```

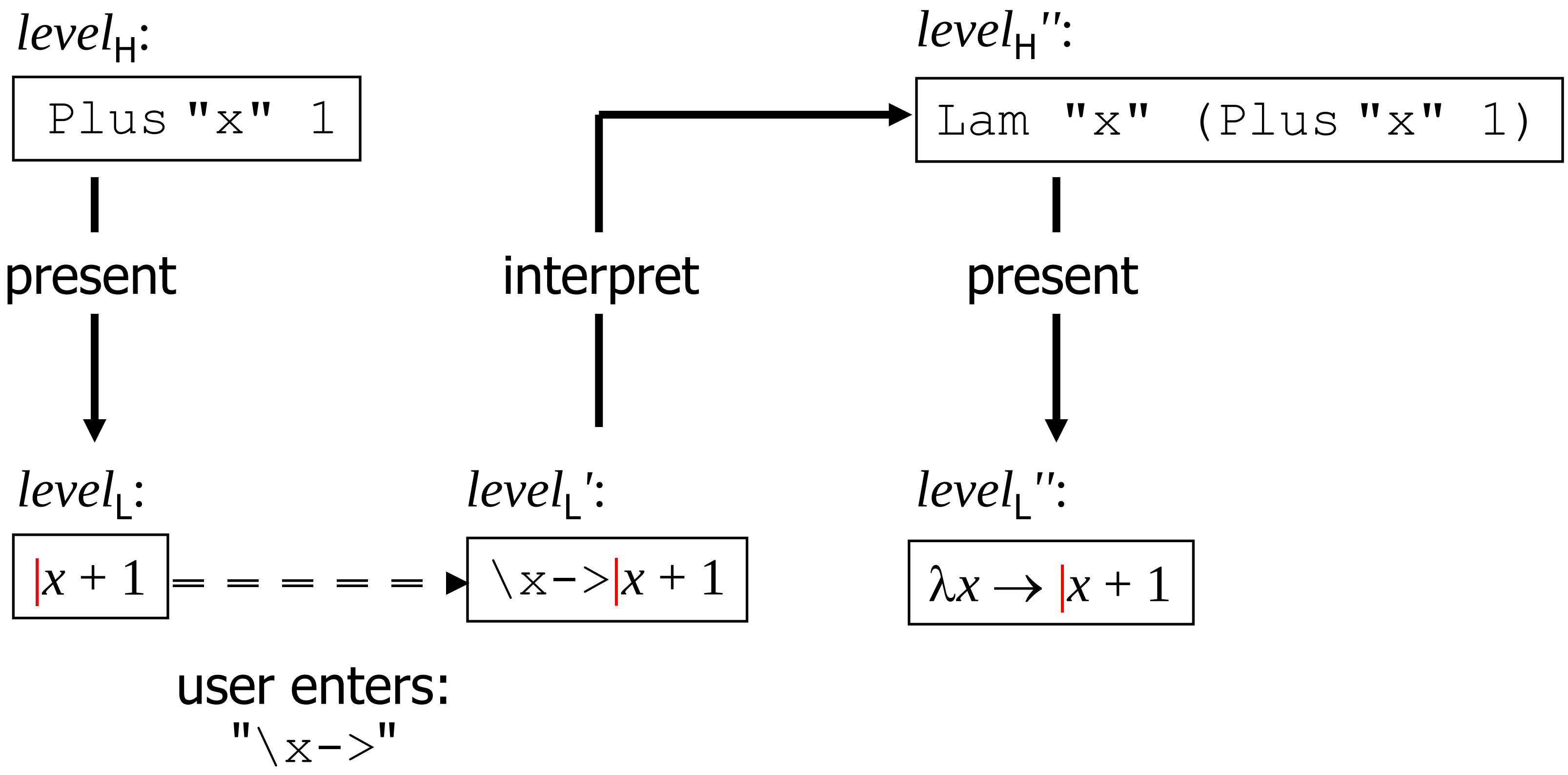
+ ~130 lines of sheet code (without move generator)

2. Specification

Step 1: Single layer

- Two levels:
  - $Level_H$ (Document) and $Level_L$ (Presentation)
- Given
  - present :: $Level_H \rightarrow Level_L$ (Total & injective)
- we specify
  - interpret :: $Level_L \rightarrow Level_H$
- PSD analogy:
  - (present, interpret) is (*get*, *put*)
  - ($Level_H$, $Level_L$) is (*Source*, *View*)
  - However, we give only a specification, no language
  - No assumptions on level types or formalism for functions

Data flow



$\mathit{level}_{\mathrm{H}}$:

`Plus "x" 1`

present

$\mathit{level}_{\mathrm{L}}$:

|$x + 1$

user enters:
`"\x->"`

interpret

$\mathit{level}_{\mathrm{L}}'$:

`\x->`|$x + 1$

$\mathit{level}_{\mathrm{H}}''$:

`Lam "x" (Plus "x" 1)`

present

$\mathit{level}_{\mathrm{L}}''$:

$\lambda x \rightarrow$ |$x + 1$

Requirements

- Precondition: $level_L$ = present $level_H$
- Edit operation: $level_L \in level'_L$

- Requirements:
- {true} $Comp$ {$level_L''$ = present $level_H''$} (Postcondition)
- {$level_L'$ = present h} $Comp$ {$level_L'' = level_L'$} (Pres-Inert)
- {true} $Comp$ {$level_l'$ “close to” $level_L''$} (Intended)

Computation

- Comp A $\mathit{level}_{\mathrm{H}}'' :=$ interpret $\mathit{level}_{\mathrm{L}}'$
  $\mathit{level}_{\mathrm{L}}'' :=$ present $\mathit{level}_{\mathrm{H}}''$

- $l =$ present $h \;\Rightarrow\; h =$ interpret l (InterPresent)
- Interpresent implies all requirements

Step 2: Extra state

- Extra state in both directions
  - Whitespace, expansion state, hidden data, etc.
- Relations instead of functions
- $Present :: Level_{\mathrm{L}} \sim Level_{\mathrm{H}}$
- $Interpret :: Level_{\mathrm{H}} \sim Level_{\mathrm{L}}$
- Equivalence relations L and H for extra state
  - $L :: Level_{\mathrm{L}} \sim Level_{\mathrm{L}}$ and $H :: Level_{\mathrm{H}} \sim Level_{\mathrm{H}}$
  - `("x + 1")` L `(" x +    1")`
  - `(Decl "f" 1)` H `(Decl "f" (Sum 1 2))`
- Wildcards: `data Decl = Decl String` $*_{Exp}$

Functions instead of relations

- $[x]_R \;=\; \{\, y \mid x\, R\, y \,\}$ Equivalence class
- $T/_R \;=\; \{\, [x]_R \mid x :: T \,\}$ Factor set: all eq. classes

- Functions between equivalence classes
- present :: $Level_{\mathrm{H}}/_{H} \rightarrow Level_{\mathrm{L}}/_{L}$
- interpret :: $Level_{\mathrm{L}}/_{L} \rightarrow Level_{\mathrm{H}}/_{H}$

- $l\ Present\ h \quad \equiv \quad [l]_L$ = present $[h]_H$ (present-Char)
- $l\ Interpret\ h \quad \equiv \quad [h]_L$ = interpret $[l]_L$ (interpret-Char)

Requirements

- $\{\mathrm{true}\}\ \mathit{Comp}\ \{[\mathit{level}_{\mathrm{L}}'']_{L} = \text{present } [\mathit{level}_{\mathrm{H}}'']_{H}\}$ (Postcondition)
- $\{[\mathit{level}_{\mathrm{L}}']_{L} = \text{present } [h]_{H}\}\ \mathit{Comp}\ \{\mathit{level}_{\mathrm{L}}'' = \mathit{level}_{\mathrm{L}}'\}$ (Pres-Inert)
- $\{\mathrm{true}\}\ \mathit{Comp}\ \{\mathit{level}_{\mathrm{l}}'\ \text{“close to”}\ \mathit{level}_{\mathrm{L}}''\}$ (Intended)
- $\{\mathrm{true}\}\ \mathit{Comp}\ \{\mathit{level}_{\mathrm{H}}\ \text{“close to”}\ \mathit{level}_{\mathrm{H}}''\}$ (Doc-Preserve)
- $\{[\mathit{level}_{\mathrm{L}}']_{L} = \text{present } [\mathit{level}_{\mathrm{H}}]_{H}\}\ \mathit{Comp}\ \{\mathit{level}_{\mathrm{H}}'' = \mathit{level}_{\mathrm{H}}\}$ (Doc-Inert)

Computation

- Select a value from eq. class, reusing extra state
- $> :: \mathit{Level}/_R \rightarrow \mathit{Level} \rightarrow \mathit{Level}$
  - E.g. `(Decl "g" *)` $>$ `(Decl "f" 1)` $=$ `(Decl "g" 1)`

- $[\,[x]_R > y\,]_R \;=\; [x]_R$ (>-Valid)
- $[x]_R = [y]_R \;\Rightarrow\; [x]_R > y \;=\; y$ (>-Idem)
- $[x]_R > y$ "close to" y (>-Close)

- Comp A $\mathit{level}_{\mathrm{H}}'' :=$ interpret $[\mathit{level}_{\mathrm{L}}']_L >_H \mathit{level}_{\mathrm{H}}$
  $;\;\; \mathit{level}_{\mathrm{L}}'' :=$ present $[\mathit{level}_{\mathrm{H}}'']_H >_L \mathit{level}_{\mathrm{L}}'$

- $[l]_L =$ present $[h]_H \;\Rightarrow\; [h]_H =$ interpret $[l]_L$ (InterPresent)
- InterPresent implies all requirements

Step 3: Composite layer

- Not discussed in detail because of complexity
- Problem when composing:
  - Composition of mappings between equivalence classes is not necessarily a mapping between equivalence classes again
  - Components $[l]_{LL} = \text{present}_{\text{L}}\ [m]_{LH}$ and $[\mathrm{m}]_{HL} = \text{present}_{\text{H}}\ [\mathrm{h}]_{HH}$
  - Composition is not always $[l]_{CL} = \text{present}_{\text{C}}\ [h]_{CH}$
- For present, we may assume a valid decomposition, but interpret may fail
- Other problems occur as well
- Solution: strong restrictions on composition, explained in thesis

Step 4: Duplicates

- Example: present $x = (x, x, x)$
- Hard to define without assumptions on types and mappings
- When a duplicate is modified, the modified value is used for the document update
  - Updated title in contents leads to update on chapter title

- Problem with Intended requirement
  - Intended: $\{\text{true}\}\ \mathit{Comp}\ \{\mathit{level}_{\mathrm{l}}'\ \text{“close to”}\ \mathit{level}_{\mathrm{L}}''\}$
  - Result of $(1,1,1) \in (1,2,1)$ should be $(2,2,2)$
  - However, $(1,2,1)$ is closer to $(1,1,1)$ than $(2,2,2)$

- Solution: block out interfering duplicates with operator Δ

Requirements: Intended

- $\Delta :: \ Level \rightarrow Level \rightarrow Level^*$
  - Δ depends on present, usage: $level_l \ \Delta \ level_l'$
  - Duplicates of edited parts become wildcards, which do not affect “close to” and equality.
  - For present $x = (x, x, x)$ $\Rightarrow$ $(1,1,1) \ \Delta \ (1,2,1) = (*,2,*)$
  - For present $(x,y) = (x, y, x+y)$ $\Rightarrow$ $(2,5,7) \ \Delta \ (2,6,7) = (2,6,*)$

    $(2,5,7) \ \Delta \ (2,5,8) = (*,*,8)$
  - Not possible to give formal definition

- New requirement:
  - $\{\text{true}\} \ Comp \ \{level_l \ \Delta \ level_l' \ \text{“close to”} \ level_L''\}$ (Intended)
- Because presentation extra state is now no longer reused for duplicates, we need the (weaker) requirement:
  - $\{\text{true}\} \ Comp \ \{level_L' \ \text{“close to”} \ level_L''\}$ (Pres-Preserve)

Requirements: Pres-Inert

- Also a problem with Pres-Inert
- Old Pres-Inert: $\{[level_{\mathrm{L}}']_L = \text{present } [h]_H\}\ Comp\ \{level_{\mathrm{L}}'' = level_{\mathrm{L}}'\}$

- Simple source editor with document [f = a + 2, a = 1]
  and presentation with type check "f = ...; a = 1 OK"
- User update "1" $\in$ "True" should yield "f = ...; a = True ERROR"
- But since "f = ...; a = True OK" is the presentation of some h,
  Pres-Inert specifies update on hidden body of f (e.g. a+2 $\in$ undefined)

- Only require updated parts to be inert and not their duplicates
  + fix precondition to handle interfering duplicates in presentation
  - $\{[level_{\mathrm{l}}\ \Delta\ level_{\mathrm{l}}']_L =^* \text{present } [h]_H\}\ Comp\ \{level_{\mathrm{l}}\ \Delta\ level_{\mathrm{l}}' =^* level_{\mathrm{L}}''\}$

    (Pres-Inert)
- Other requirements remain unchanged

Conclusions / future work

- Given present, specification of interpret plus computation
- When simple requirements are met, more complex requirements can be proven
- Specification helps to clarify concept of extra state

- Specification
  - Find workable restrictions for composite layer
  - Maybe formalize closeness
- Simple fixes on prototype
  - Incrementality (10x faster), focus, edit model
- Add evaluation layer
- Automatically construct interpret that meets the specification
  - Typed Inv + presentation extra state + AG
  - Inversion is fragile, must be easy to specify non-automatic parts

Questions?

For thesis & more information:

`http://www.cs.uu.nl/research/projects/proxima/`