|galois|

An Optim(L) Approach to Parsing Random Access Formats

Mark Tullsen, Sam Cowger, Mike Dodds (Galois, Inc.)
Peter Wyatt (PDF Association)

May 2024, LangSec

Thanks:

- SafeDocs (This work supported in part by DARPA awards HR001119C0073 and HR001119C0079).

The Backstory

- Writing correct & safe PDF parsers (SafeDocs)

The Backstory

- ~~Writing correct & safe PDF parsers (SafeDocs)~~

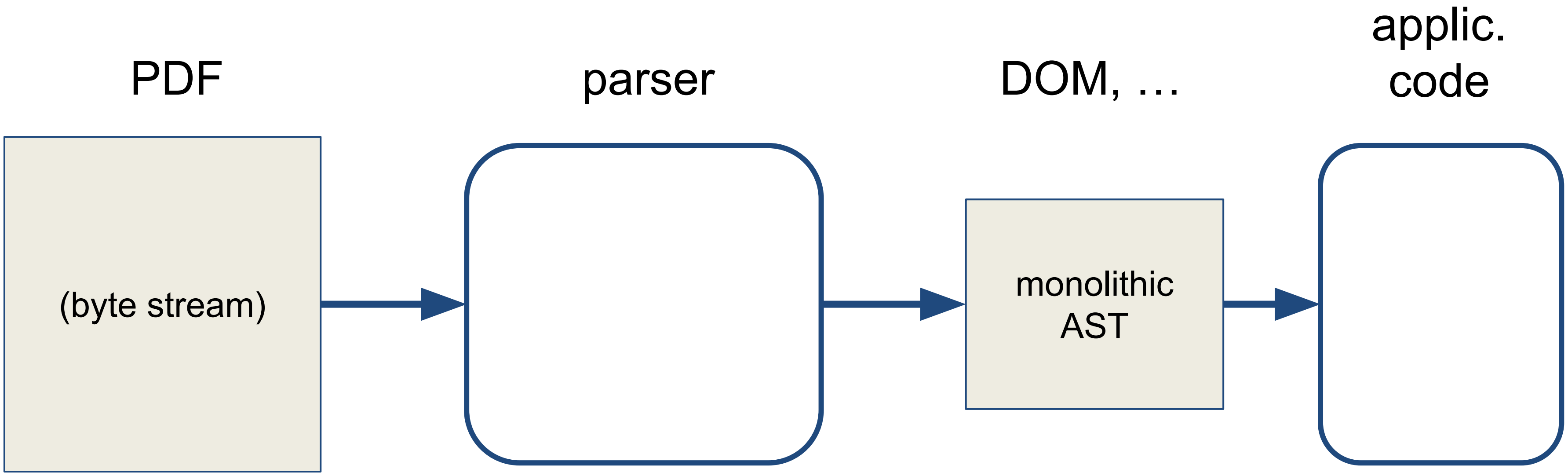
- Writing correct & safe & useful/efficient PDF tools (!)
  - A surprisingly different problem.
  - Get correct parser and efficient tool at the same time?
  - Needing not more / improved “parsing technology” but …

Definitions: Transformational vs. Reactive Systems

In *On the Development of Reactive Systems* (1985), Harel & Pneuli note:

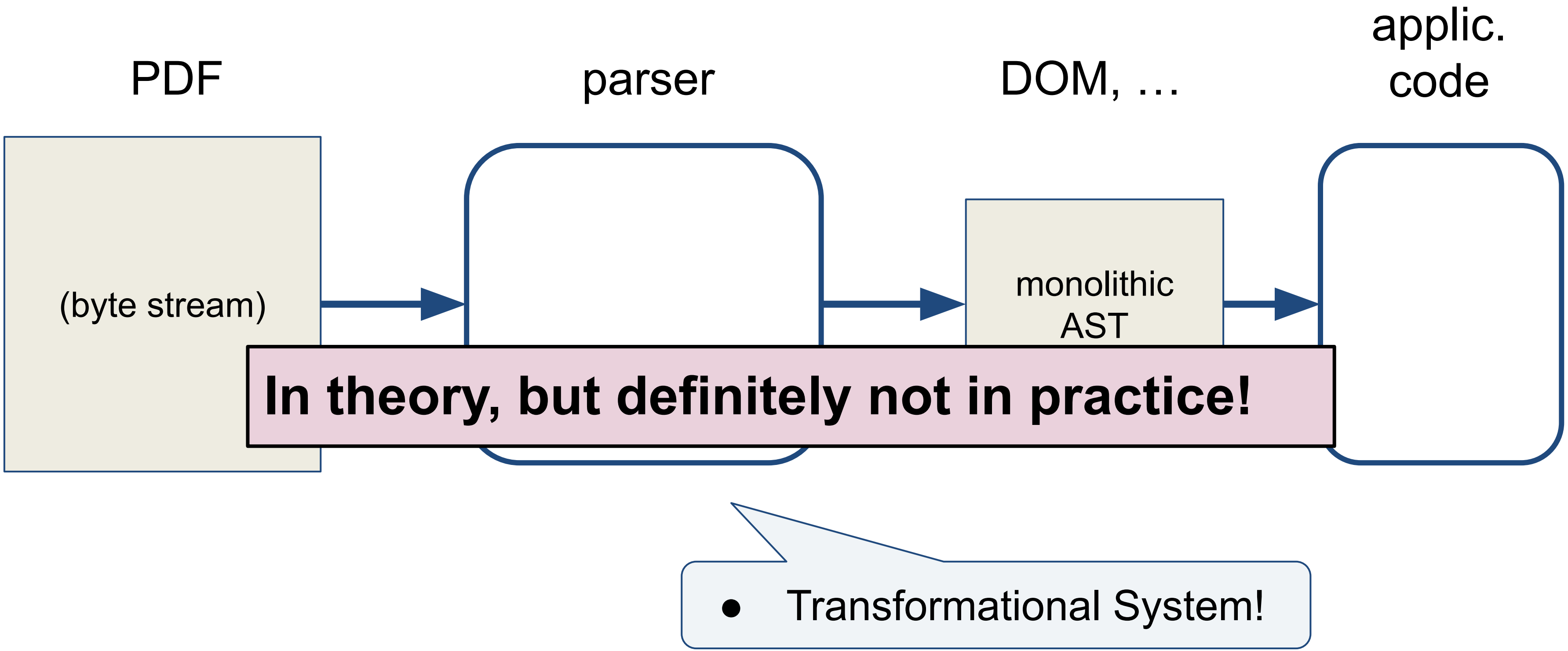
> "Our proposed distinction is between what we call <u>transformational</u> and <u>reactive</u> systems.
> …
> A <u>transformational</u> system accepts inputs, performs transformations on them and produces outputs.
> …
> <u>Reactive</u> systems, on the other hand, are repeatedly prompted by the outside world and their role is to continuously respond to external inputs."

The PDF Problem?

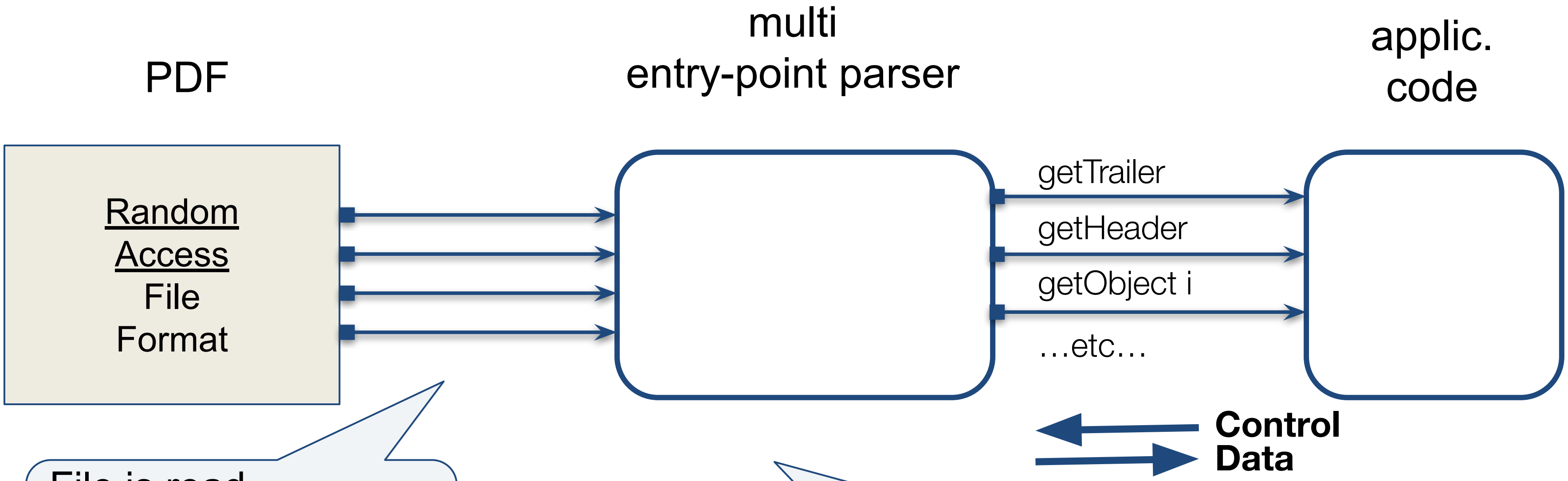


- Transformational System!

The PDF Problem?



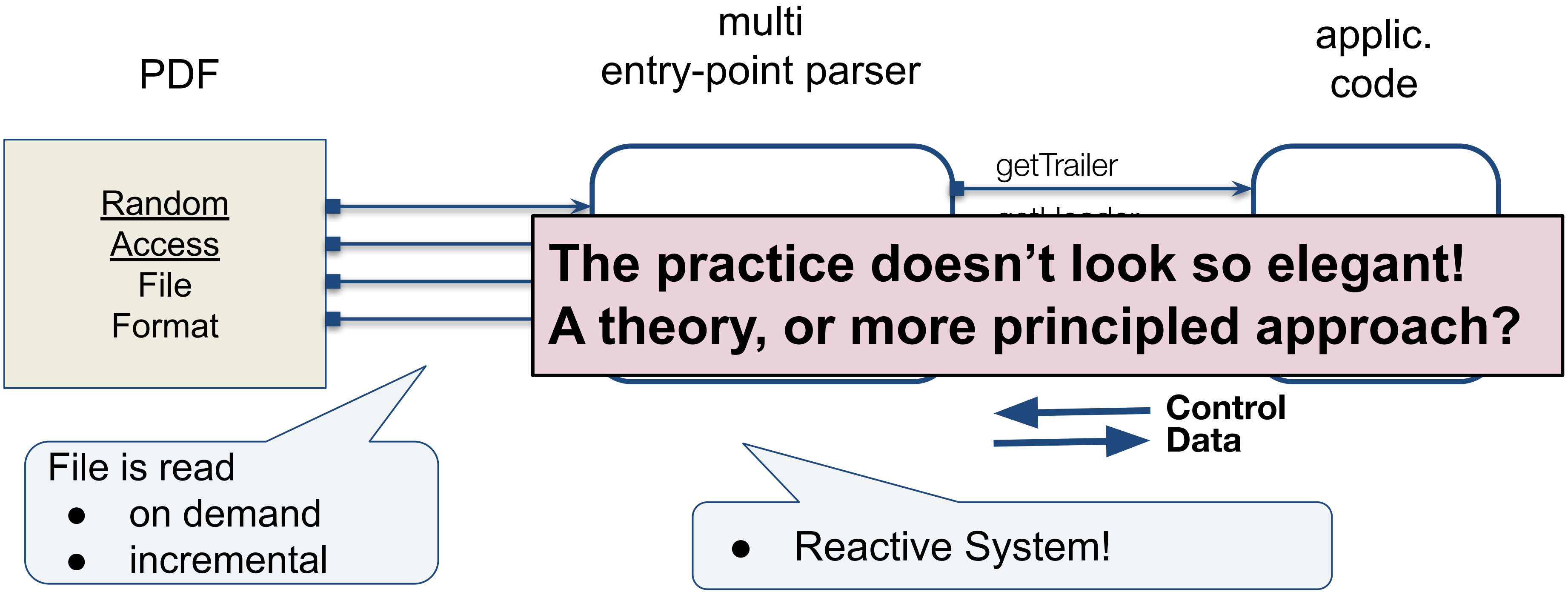
The PDF Problem is Actually …



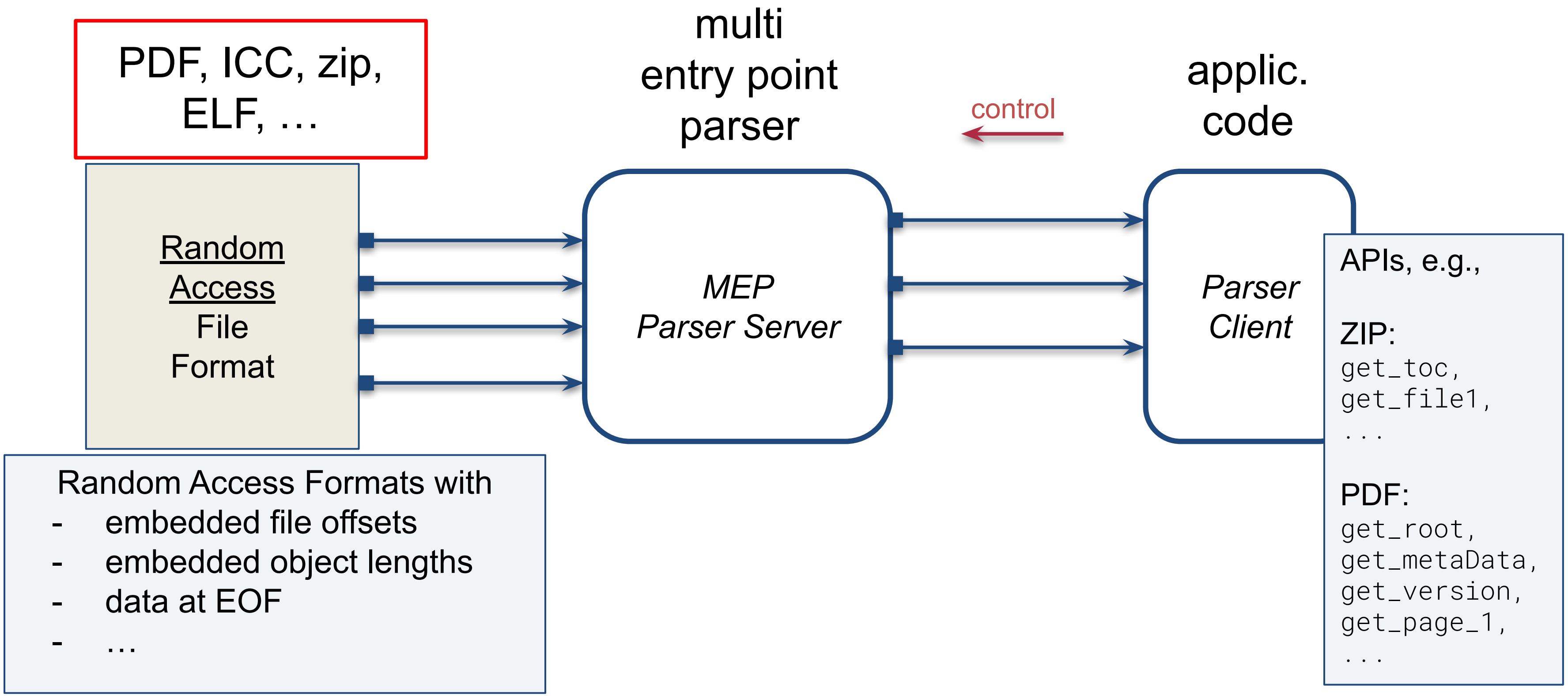
File is read
- on demand
- incremental

- Reactive System!

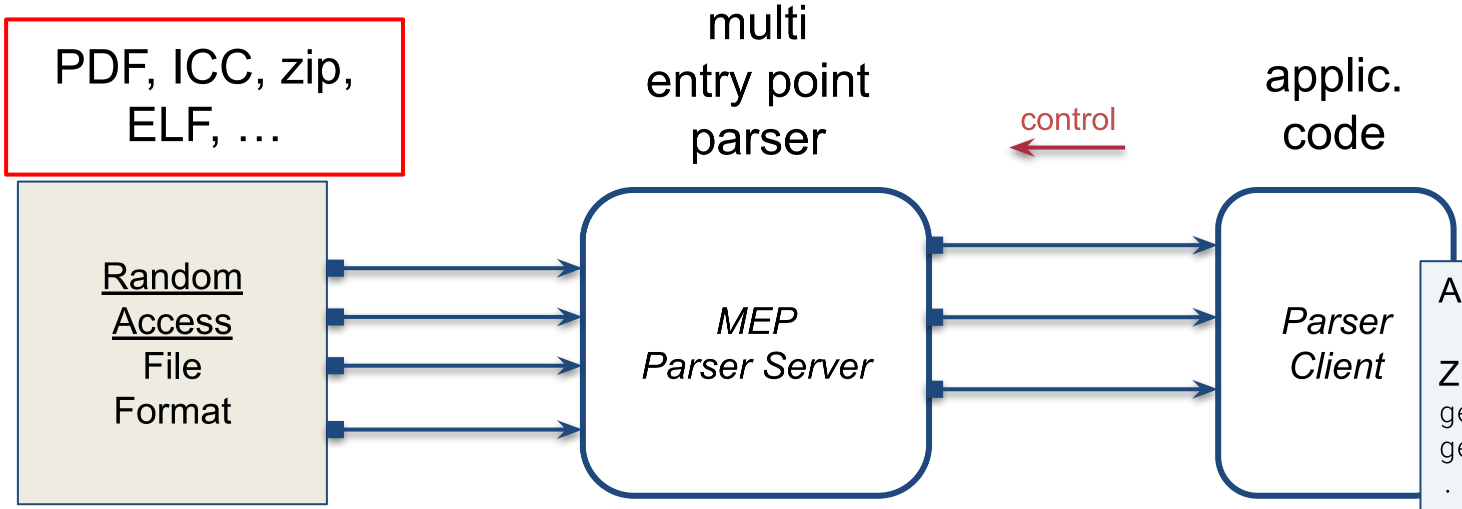
The PDF Problem is Actually …



Random-Access Formats and Multi-Entry-Point Parsers



Random-Access Formats and Multi-Entry-Point Parsers



APIs, e.g.,

ZIP:
`get_toc, get_file1, ...`

PDF:
`get_root, get_metaData, get_version, get_page_1, ...`

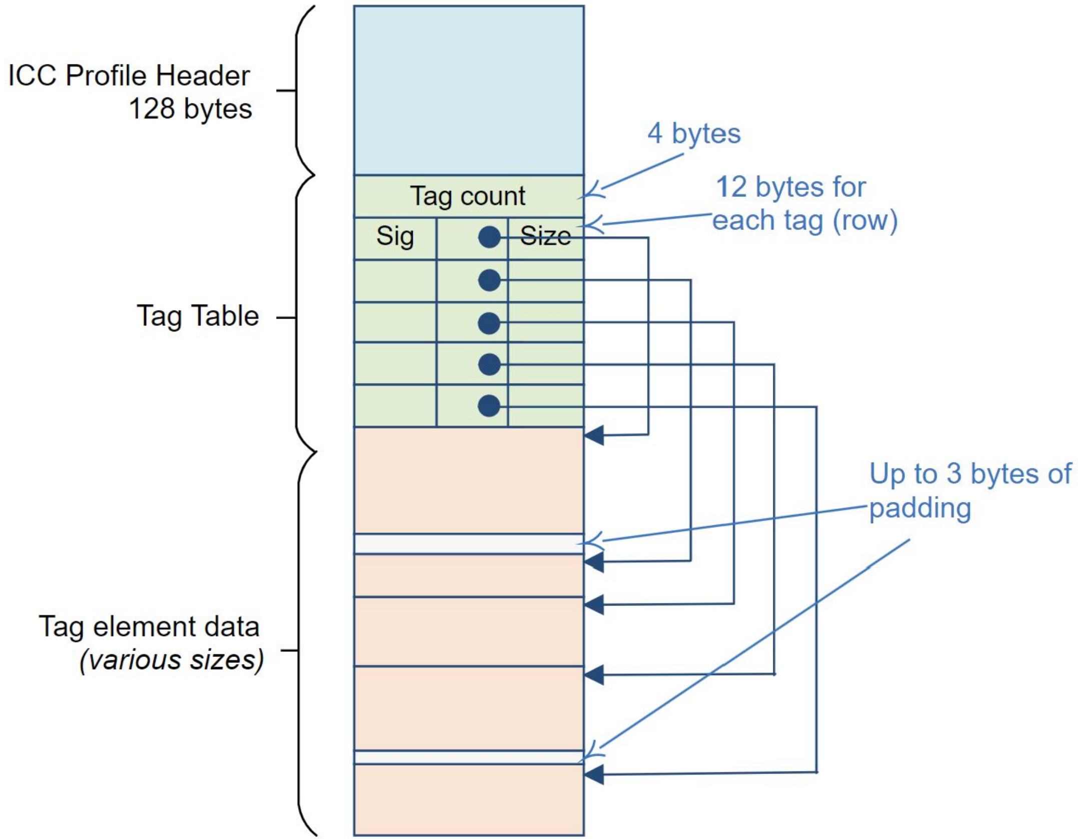
Random Access Formats with
- embedded file offsets
- embedded object lengths
- data at EOF
- …

- Safe? **All** sequences of calls?
- Easy to use?
- Ad hoc design for each format?
- The dreaded “shotgun parser”!

ICC: Our Running Example

- ICC - International Color Consortium
- ICCmax is a color management profile
- Used in PDF

Example Format: ICC (Tag Length Value ish)



ICC, A Traditional Approach

```
pICC : Parser [TED]
pICC = do
  cnt <- pInt4Bytes
  tbl <- pMany cnt pTblEntry  -- parse cnt Table Entries
  rsTeds <- except $ mapM getSubRegion tbl
  teds   <- mapM applyPTED rsTeds
  return teds

-- parse a Tagged Element Data (TED):
applyPTED :: Parser TED
applyPTED (sig,offset,size) =
  withParseRegion offset size (pTED sig)
```

1. Primitive Parsers

2. Monadic Combinators

ICC, A Traditional Approach

```
pICC : Parser [TED]
pICC = do
  cnt <- pInt4Bytes
  tbl <- pMany cnt pTblEntry  -- parse cnt Table Entries
  rsTeds <- except $ mapM getSubRegion tbl
  teds   <- mapM applyPTED rsTeds
  return teds

-- parse a Tagged Element Data (TED):
applyPTED :: Parser TED
applyPTED (sig,offset,size) =
  withParseRegion offset size (pTED sig)
```

1. Primitive Parsers

2. Monadic Combinators

New: Primitive to change the locus (region) of parse!

ICC, A Traditional Approach

```
pICC : Parser [TED]
pICC = do
   cnt <- pInt4Bytes
   tbl <- pMany cnt pTblEntry  -- parse cnt Table Entries
   rsTeds <- except $ mapM getSubRegion tbl
   teds   <- mapM applyPTED rsTeds
   return teds

-- parse a Tagged Element Data (TED):
applyPTED :: Parser TED
applyPTED (sig,offset,size) =
   withParseRegion offset size (pTED sig)
```

1. Primitive Parsers

2. Monadic Combinators

New: Primitive to change the locus (region) of parse!

Cons:
- Region primitive complicates!
- Evaluation order tied to format ordering (overspecifies order)
- Imperative, stateful monad
- Returns one monolithic value.

ICC, Using Optim(L), L=XRP

```
[optimal|
icc : Region -> ICC
icc rFile =
  { (cnt,rRest) = <| pInt4Bytes                           @!  rFile    |>
  , tbl         = <| pManySRPs (v cnt) pTblEntry    @!- rRest    |>
  , rsTeds      = <| except $ mapM (getSubRegion rFile) (v tbl) |>
  , teds        = <| mapM applyPTED rsTeds                         |>
  }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```

ICC, Using Optim(L), L=XRP

```
[optimal|
icc : Region -> ICC
icc rFile =
  { (cnt,rRest) = <| pInt4Bytes                      @!  rFile     |>
  , tbl         = <| pManySRPs (v cnt) pTblEntry     @!- rRest     |>
  , rsTeds      = <| except $ mapM (getSubRegion rFile) (v tbl)    |>
  , teds        = <| mapM applyPTED rsTeds                         |>
  }
|]
```

- Optim(L) syntax (in quasiquote)
- Compiled away using Template Haskell
- The `icc` module is an unordered set of bindings

```
applyPTED r = pTED (region_width r) `appSRP` r
```

Explicit Region Parsing (XRP) DSL

- shallow embedding in Haskell
- regions explicit and abstract
- no need for `seek` primitive.

ICC, Using Optim(L), L=XRP

For each binding: an entry point (or lazy API call, or demand)

```
[optimal|
icc : Region -> ICC
icc rFile =
 { (cnt,rRest) = <| pInt4Bytes                      @!  rFile    |>
 , tbl         = <| pManySRPs (v cnt) pTblEntry     @!- rRest    |>
 , rsTeds      = <| except $ mapM (getSubRegion rFile) (v tbl)  |>
 , teds        = <| mapM applyPTED rsTeds                        |>
 }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```

← get_tbl

ICC, Using Optim(L), L=XRP

```
[optimal|
icc : Region -> ICC
icc rFile =
  { (cnt,rRest) = (5,[4..EOF])
  , tbl         = [te1,te2,te3,te4,te5]
  , rsTeds      = <| except $ mapM (getSubRegion rFile) (v tbl) |>
  , teds        = <| mapM applyPTED rsTeds                      |>
  }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```

← get_tbl
→ [...]
← get_cnt
→ 5
…

Assessments

Nice

- Multi-entry points
  - generalizes single entry point (returning monolithic AST)
- The format is described declaratively (no over sequentialization)
- Has an imperative realization
  - Computationally efficient
- Gives us a novel semantics: “lazy actions”
  - (not the the same as lazy evaluation)
  - Very useful: un-needed actions do not cause failure.

However,

- XRP must be explicit about regions
  - This allows for Optim(L) to know dependencies.
  - … and non-dependencies, giving us parallelism.

Assessments

Nice

- Multi-entry points
  - generalizes single entry point (returning monolithic AST)
- The format is described declaratively (no over sequentialization)
- Has an imperative realization
  - Computationally efficient
- Gives us a novel semantics: “lazy actions”
  - (not the the same as lazy evaluation)
  - Very useful: un-needed actions do not cause failure.

However,

- XRP must be explicit about regions
  - This allows for Optim(L) to know dependencies.
  - … and non-dependencies, giving us parallelism.

Mitigation: Easy to abstract over
- single entry access to
- sequential parsers

(giving us “pay for what we use”)

Assessments

Nice

- Multi-entry points
  - generalizes single entry point (returning monolithic AST)
- The format is described declaratively (no over sequentialization)
- Has an imperative realization
  - Computationally efficient
- Gives us a novel semantics: “lazy actions”
  - (not the the same as lazy evaluation)
  - Very useful: un-needed actions do not cause failure.

<u>However,</u> less useful than desired:

Could we access **one** element of \`teds\` without parsing all \`teds\`? (the whole ICC file!)

However,

- XRP must be explicit about regions
  - This allows for Optim(L) to know dependencies.
  - … and non-dependencies, giving us parallelism.

<u>Mitigation</u>: Easy to abstract over

- single entry access to
- sequential parsers

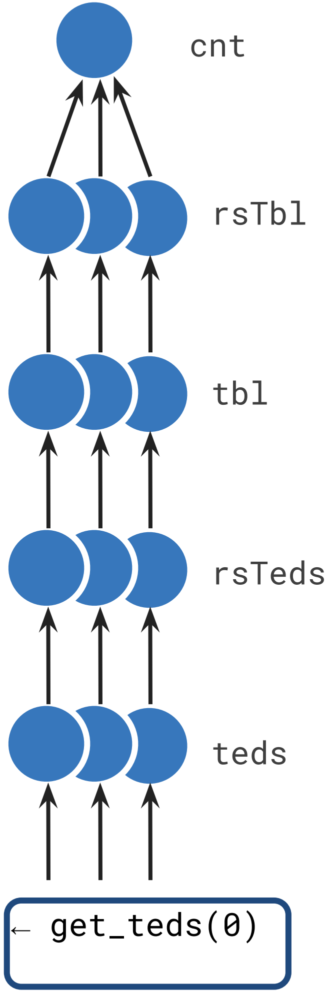
(giving us “pay for what we use”)

Optim(L) with Lazy Vectors

```
[optimal|
icc_lazyVectors : Region -> ICC
icc_lazyVectors rFile =
 { (cnt,rRest) = <| pInt4Bytes   @! rFile |>
 , rsTbl       = generate (v cnt)
                    <| \i-> regionIntoNRegions
                          (v cnt) rRest (width pTblEntry) i     |>
 , tbl         = map rsTbl <| \r-> pTblEntry @$$ r               |>
 , rsTeds      = map tbl   <| \r-> except $ getSubRegion rFile r |>
 , teds        = map rsTeds  <| applyPTED                        |>
 }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```

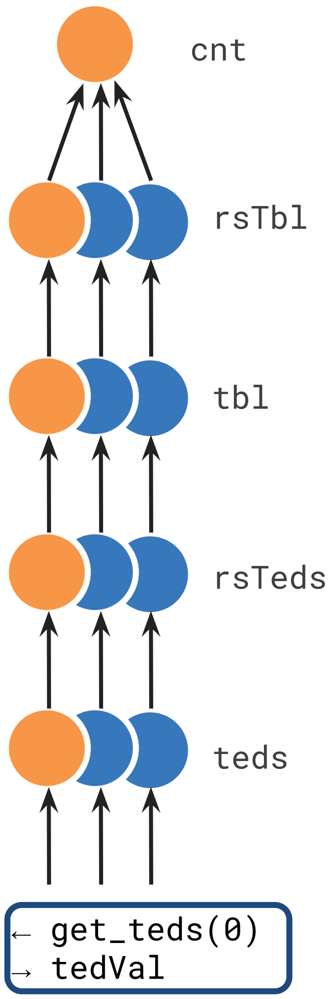
Optim(L) with Lazy Vectors

```
[optimal|
icc_lazyVectors : Region -> ICC
icc_lazyVectors rFile =
 { (cnt,rRest) = <| pInt4Bytes   @! rFile |>
 , rsTbl      = generate (v cnt)
                  <| \i-> regionIntoNRegions
                         (v cnt) rRest (width pTblEntry) i     |>
 , tbl        = map rsTbl <| \r-> pTblEntry @$$ r               |>
 , rsTeds     = map tbl    <| \r-> except $ getSubRegion rFile r |>
 , teds       = map rsTeds  <| applyPTED                         |>
 }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```



Optim(L) with Lazy Vectors

```
[optimal|
icc_lazyVectors : Region -> ICC
icc_lazyVectors rFile =
 { (cnt,rRest) = <| pInt4Bytes   @! rFile |>
 , rsTbl      = generate (v cnt)
                  <| \i-> regionIntoNRegions
                           (v cnt) rRest (width pTblEntry) i     |>
 , tbl        = map rsTbl <| \r-> pTblEntry @$$ r               |>
 , rsTeds     = map tbl   <| \r-> except $ getSubRegion rFile r |>
 , teds       = map rsTeds  <| applyPTED                        |>
 }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```



Optim(L)

Some Pleasant Implications

Sanity Checking of Regions

```
[optimal|
icc : Region -> ICC
icc rFile =
 { (cnt,rRest) = <| pInt4Bytes                      @!  rFile    |>
 , tbl         = <| pManySRPs (v cnt) pTblEntry     @!- rRest    |>
 , rsTeds      = <| except $ mapM (getSubRegion rFile) (v tbl) |>
 , teds        = <| mapM applyPTED rsTeds                        |>

 , crsFile      = <| XRP.makeCanonicalRegions (r cnt : r tbl : rsTeds) |>
 , isCavityFree = <| hasNoCavities $ XRP.complementCRs rFile crsFile   |>
 , teds_safe    = <| if isCavityFree
                     then return teds
                     else throwE ["teds not safe"] |>
 }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```

Fail if any regions overlap!

Fail if any cavities (unused regions) in file.

One Source: Parser Tools AND Validators

```
[optimal|
pdf : Region -> ICC
pdf rFile =
  { header   = <| ... |>
  , trailer  = <| ... |>
  , metadata = <| ... |>
  , ...
  , dom_lax  = map ... <| ...|>

  , dom_validated =
     <| if isValid dom_lax
        then return dom_lax
        else throwE ["invalid PDF"] |>
  }
|]
```

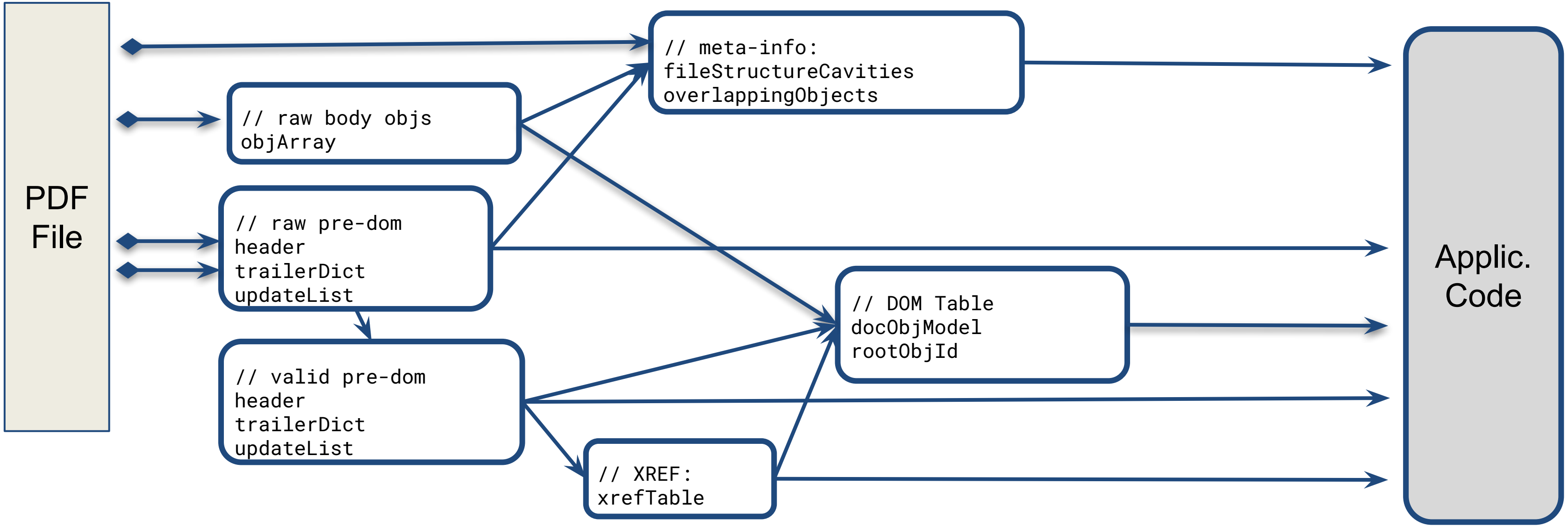
A pragmatic parser

Extending with full validation (every jot and tittle)

- Modularity: core, minimal parser is decoupled from validation code.
- `dom_lax` & `dom_validated` are equivalent (modulo …)
- lazy actions essential to making this work.

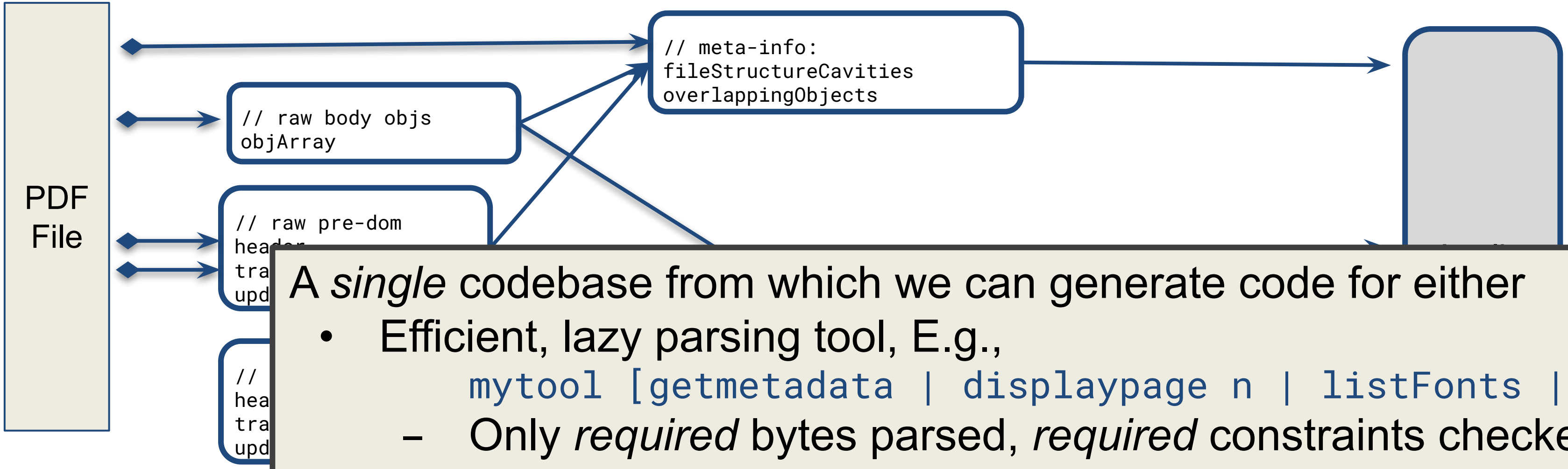
Vision: PDF Library as (DAG of) MEP Components

- Reading, parsing, constraint checking, value computation is demand driven
- Each MEP can **add** parsers, value constraints, or computation



Vision: PDF Library as (DAG of) MEP Components

- Reading, parsing, constraint checking, value computation is demand driven
- Each MEP can **add** parsers, value constraints, or computation



A *single* codebase from which we can generate code for either

- Efficient, lazy parsing tool, E.g.,
  `mytool [getmetadata | displaypage n | listFonts | … ]`
  - Only *required* bytes parsed, *required* constraints checked
- PDF validator
  - *All* bytes parsed, all constraints checked

Optim(L)

Regarding Semantics …

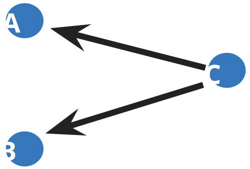
Optim(L): The Theory

Key design decision in Optim(L)!

Optim(L)

- Parameterized over the language '*L*' of computations.
- The language *L* of computations must be a <u>commutative monad</u>: i.e., the order of independent actions does not matter:

```
      do {a <- A; b <- B; c <- C[a,b]}
   == do {b <- B; a <- A; c <- C[a,b]}
```



Examples of commutative monads

- Identity: (i.e., pure code)
- Maybe: exceptions
- Reader: read-only globals

<u>Not</u> commutative monads:

- StateM: mutable globals
- IO (Input Output monad)

Possibly:

- IO as reader, …

Optim(L): Multiple Interpretations

Generally, Optim(L) has a “lazy” interpretation, but other interpretations are useful

Where L is a commutative monadic language,
and *m* is a Optim(L) module that binds L computations..

$[[\ \mathtt{Optim}_{\mathtt{Lazy}}(L)(\mathtt{m})\ ]]$ – no action is ever repeated, results cached
$[[\ \mathtt{Optim}_{\mathtt{NoCaching}}(L)(\mathtt{m})\ ]]$ – no thunks used, this generates pure code.
$[[\ \mathtt{Optim}_{\mathtt{Tracing}}(L)(\mathtt{m})\ ]]$ – lazy, logs all demands
$[[\ \mathtt{Optim}_{\mathtt{Profiling}}(L)(\mathtt{m})\ ]]$ – lazy, counts all demands

You can look at these interpretations as “programmable” variable lookups.

Optim(L): Observationally Equivalent

Observational Equivalence

- Defined in terms of API calls
- Not in terms of optimality, side-effects, or etc.

Client code cannot distinguish the lazy APIs generated from these:

$$[\![\ \mathtt{Optim}_{\mathtt{Lazy}}(L)(\mathtt{m})\ ]\!]$$
$$[\![\ \mathtt{Optim}_{\mathtt{NoCaching}}(\mathtt{L})(\mathtt{m})\ ]\!]$$
$$[\![\ \mathtt{Optim}_{\mathtt{Tracing}}(L)(\mathtt{m})\ ]\!]$$

A little more regarding XRP

For Parsing Random Access Formats, **L**=**XRP**

- (eXplicit Region Parser language)
- Three things
    - ReaderException monad.
    - Regions
        - I.e., `[startbyte..endbyte]`, but …
        - **Abstract**: can't see inside, we manipulate using "region algebra"
        - **Explicit**: Each parser must be applied to a region (no defaults)
    - Top level MEP parser is passed a top (file) level region

With Optim(L), achieves

- MEP parsers
- with optimal (caching)
- for random-access formats
- described declaratively
- implemented statefully

Optim(L)

In Conclusion …

Assessments

- Pleasant
  - Write unordered “action bindings”; then
    - Choose the interpretation
    - Let the compiler
      - do the dependency analysis
      - generate efficient, imperative code (thunks)
  - Order bindings semantically, not per data-dependencies.
- Commutative monad restriction
  - Limits scope of Optim(...)
  - But this pushed us towards a better design for XRP.
  - … and more generic Optim(L)

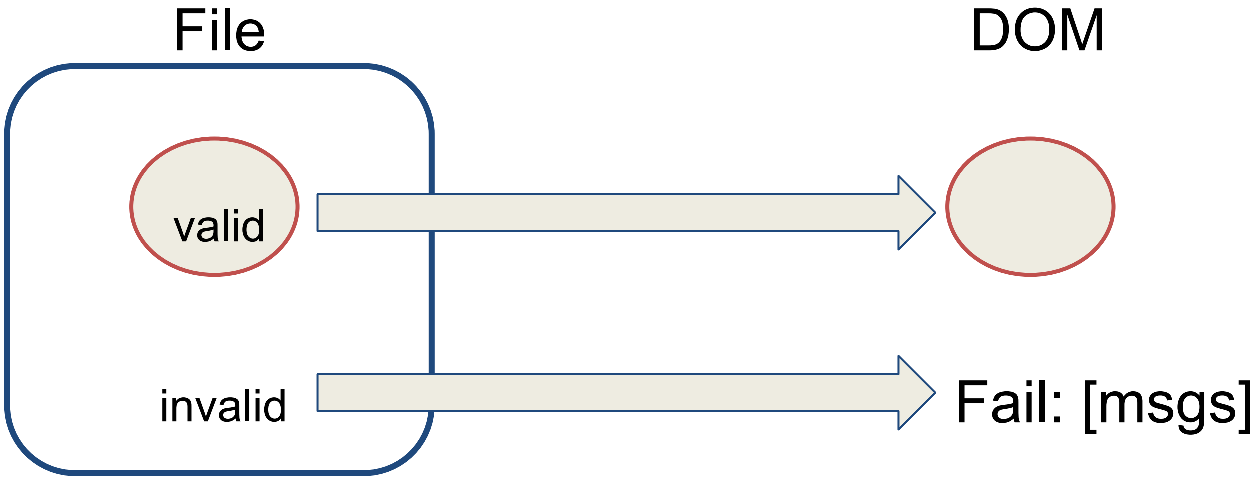
Future Developments

- Implement as standalone language & compiler, this would allow
  - More optimizations
  - Ability to create multiple tools from one Optim(L) program (e.g. validator and parser, each one “sliced, specialized, & optimized”)
  - Targeting other languages
- Optim(L)/XRP: apply to more random-access formats
- Research “bidirectional capabilities”
  - When L is bidirectional, then make Optim(L) bidirectional.

Questions?

<Backup Slides>

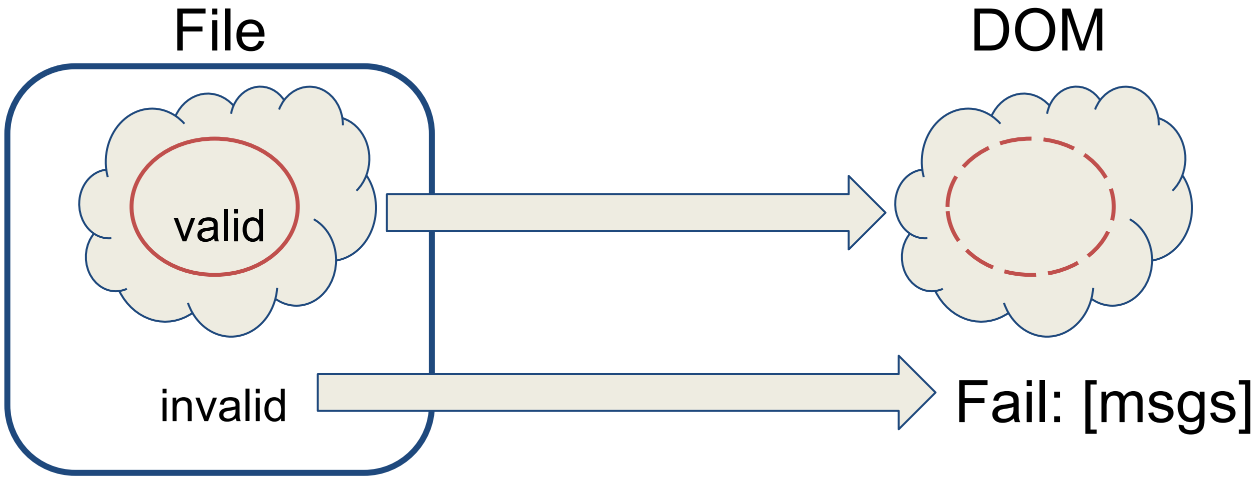
Parser ≠ Validator



Validator:

only valid PDFs can produce DOM (**must** Fail otherwise)

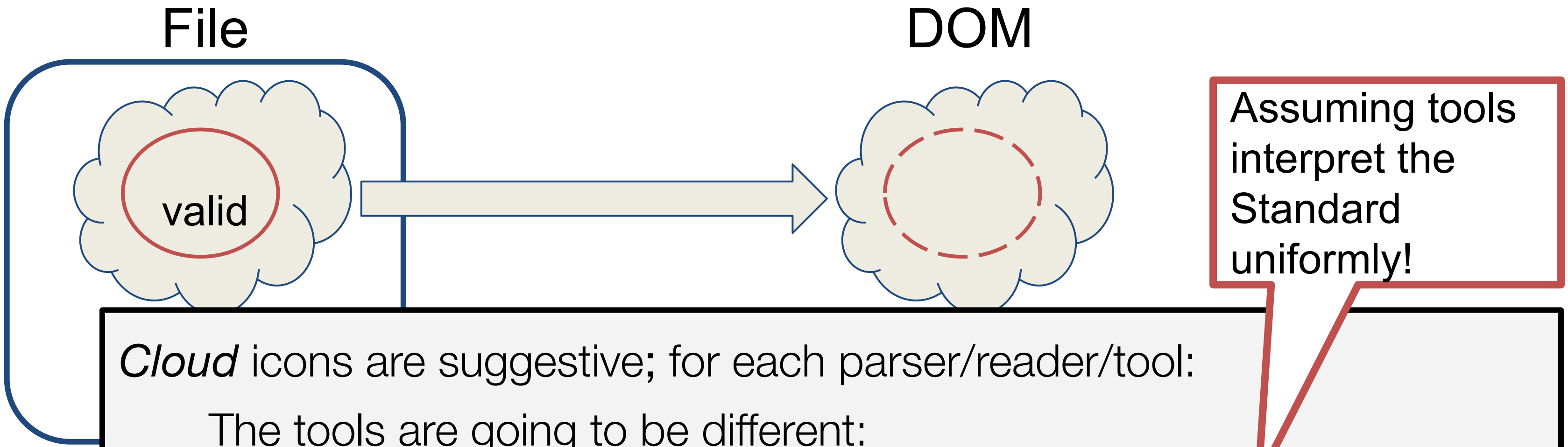
Parser ≠ Validator



Parser:

efficiently, construct the correct DOM when a valid PDF

Parser ≠ Validator



Cloud icons are suggestive; for each parser/reader/tool:

The tools are going to be different:

- redundancies in format allow for different choices
- tools in practice allow “minor” errors
- tool may traverse & evaluate implicit data structures differently.

Goal for our “parser specification”:

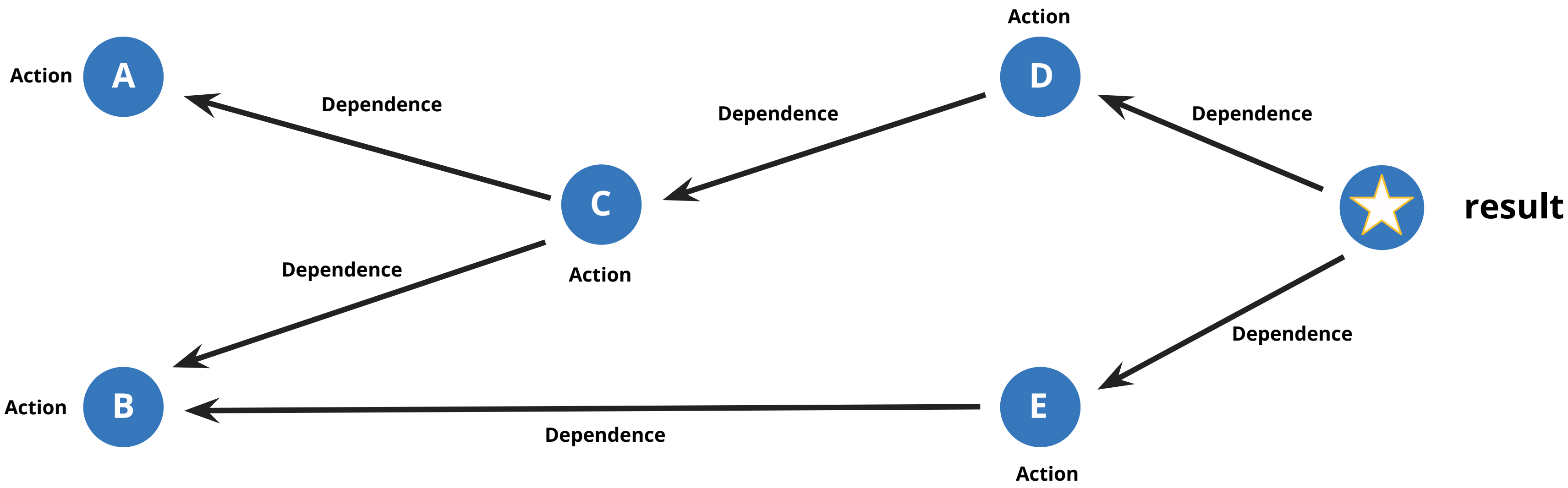
- Encompass any reasonable & correct cloud

Optim(...): Some Useful Instantiations (?)

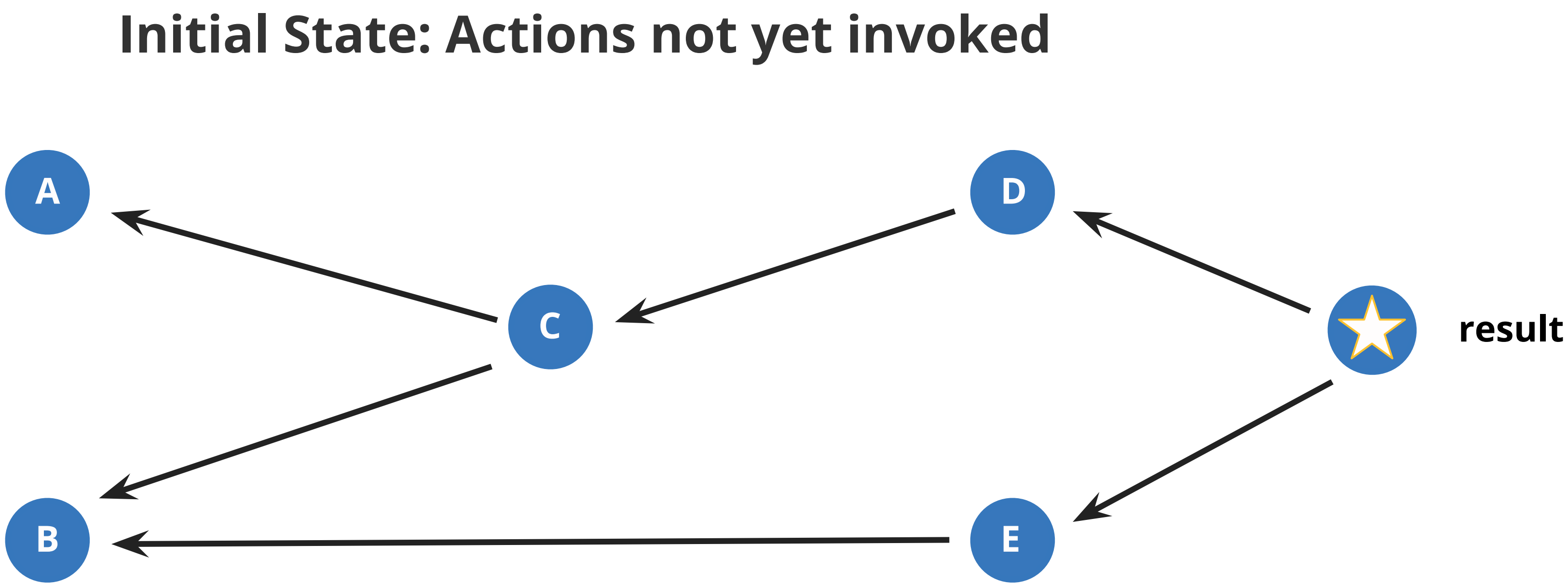
	L	monad	Binding values	We get
1	pure bash	Maybe	FileStream	In program make capability (no persistence)
2	Haskell/_	Identity	a	Lazy API to get/compute globals
3	Haskell/_	Reader	a	Lazy API for accessing global config. data
4	Haskell/_	ReaderMaybe	a	[as above] but allow for failures
5	ML, …	Identity	a	Add laziness to non-lazy language
6	Haskell/_	Reader	a	Thread down name supplies, RNG seeds, …

We're so used to the "imperative virus" and/or the monad transformer approach, we're not seeing declarative alternatives.

Traditional, Monolithic Program

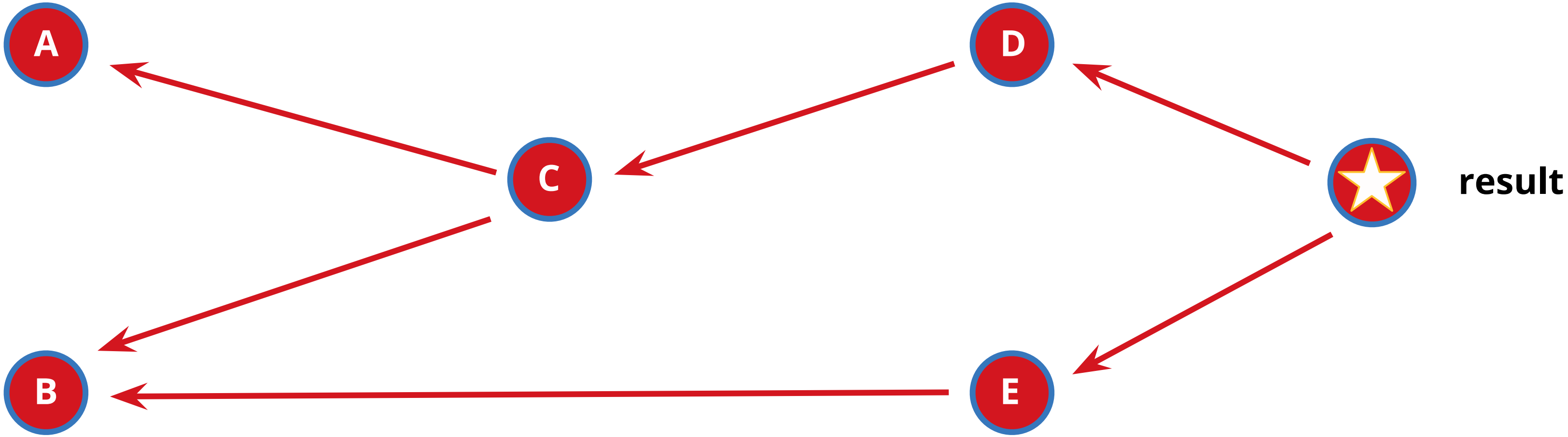


Traditional, Monolithic Program



Traditional, Monolithic Program

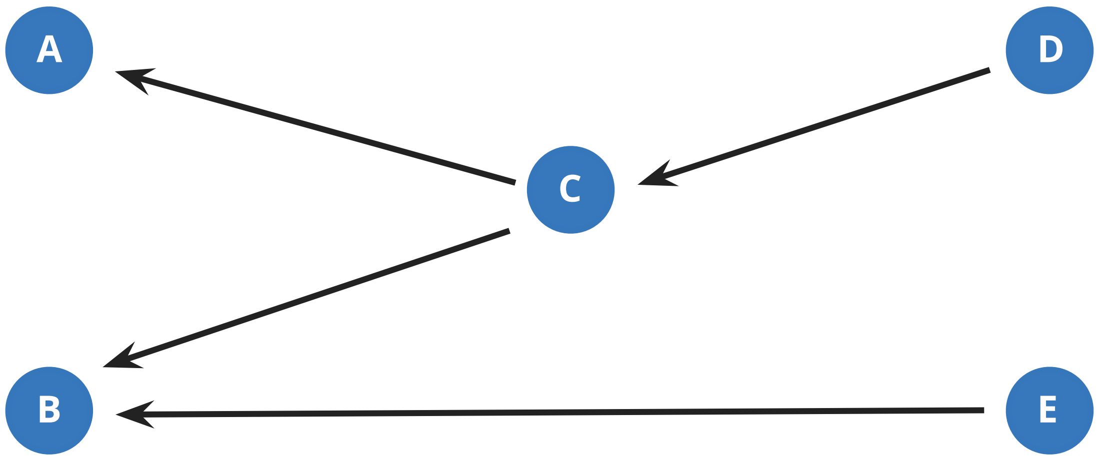
Final state: All Actions Are Invoked



What if … all we wanted was `(fst result).50` …?

Optim(L): Multiple Entry Points

Initial State: Actions not yet invoked

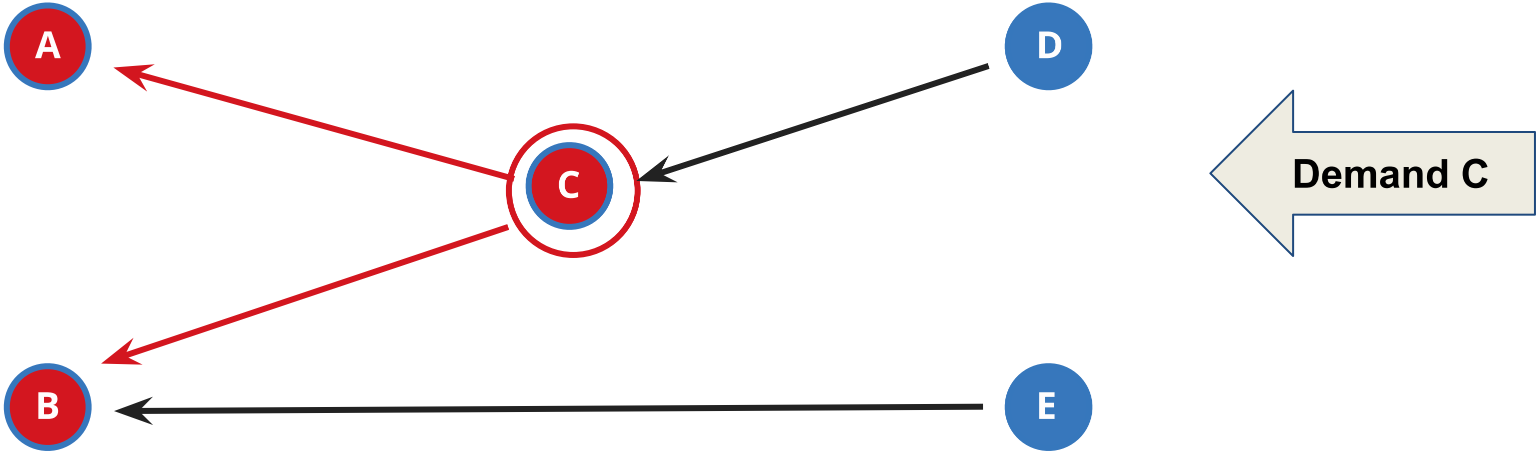


NO single
result/main/...

- Entry points {C,D,E}.
- Or {A,B,C,D,E}?
  - User decides.

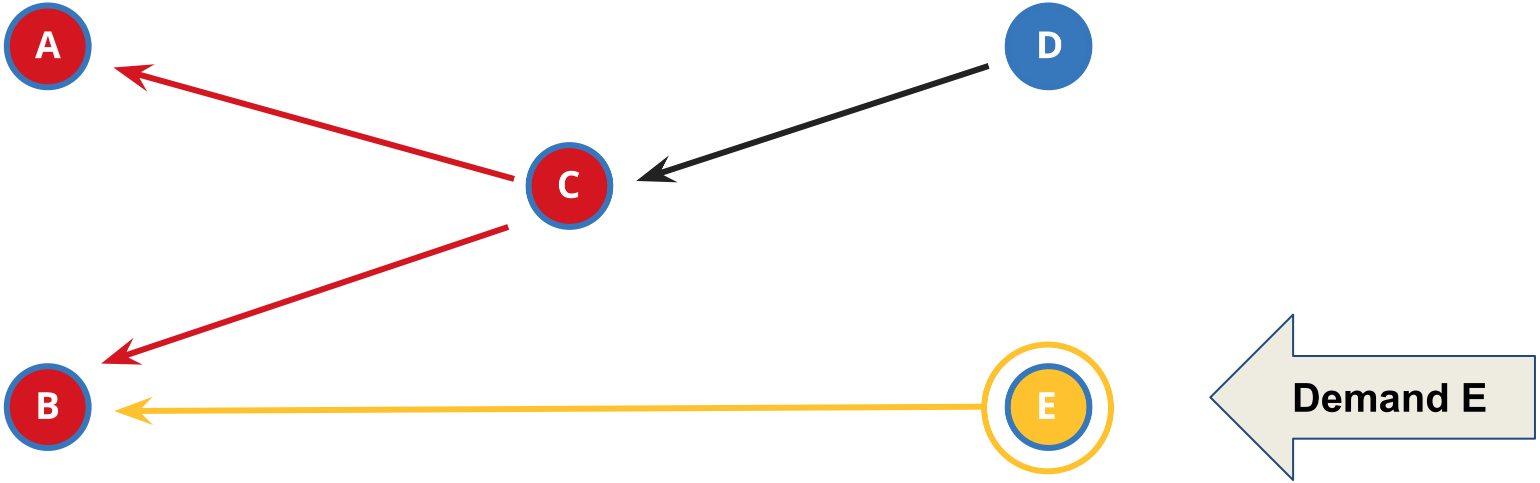
Optim(L): Demands invoke actions & update state

Intermediate state 1:
Actions A, B, and C are invoked (results cached)



Optim(L): Demands invoke actions & update state

Intermediate State 2:
B is already computed, so only E is invoked (results cached)



Optim(L): Important

Not the same as “lazy evaluation”:

- Multi-entry points
- “Actions” on the nodes are not computations but monadic actions.