Optim(L): Generating Lazy APIs from DAGs of Actions

Mark Tullsen & Sam Cowger
Galois, Inc.

May 2024, HCSS

|galois|

Optim(L): Generating Lazy APIs from DAGs of Actions

Mark Tullsen & Sam Cowger
Galois, Inc.

May 2024, HCSS

Thanks:

- SafeDocs (This work supported in part by DARPA awards HR001119C0073 and HR001119C0079).
- Sergey Bratus (our PM)
- Peter Wyatt, PDF Association

The Backstory

- Writing correct & safe PDF parsers

The Backstory

- ~~Writing correct & safe PDF parsers~~

- Writing correct & safe & useful PDF tools (!)
  - A surprisingly different problem
  - Needing not more / improved “parsing technology” but …

Outline

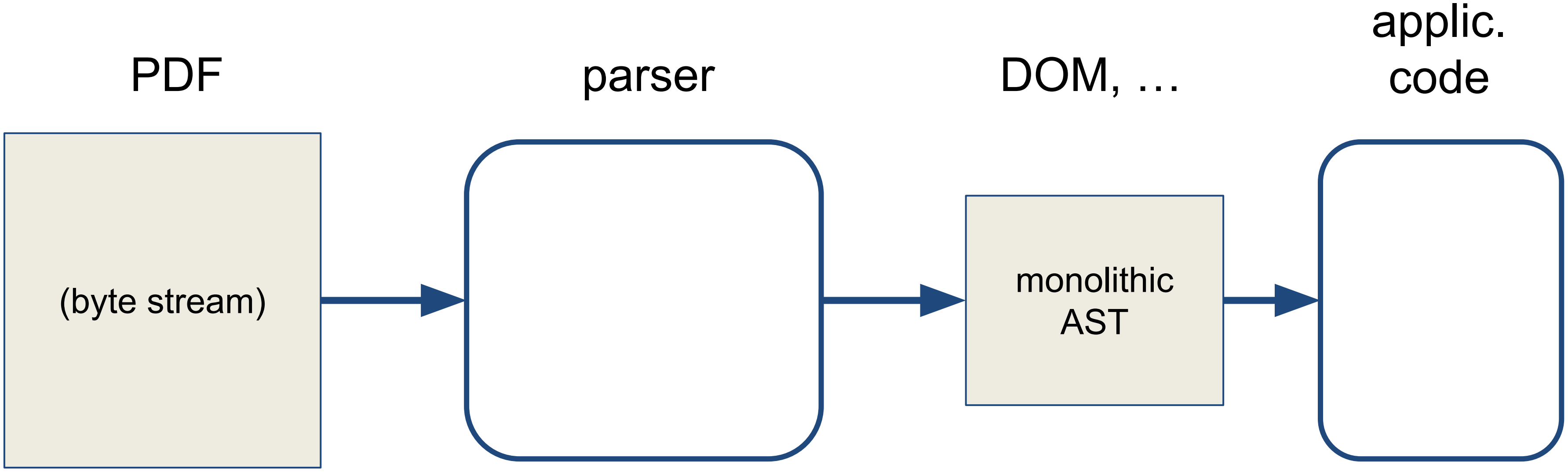
- A Need Discovered
    - What I needed, which wasn't a better parser.
    - (PDF, our use-case & running-example)
- Optim(L): Described & Applied
    - Evaluates "DAGs of L Actions" optimally
    - Can be instantiated to various "computation languages" L
    - We instantiate L to an eXplicit Region Parser (XRP) to achieve our needs
- Optim(L): Capabilities
- Optim(L): Assessments

Transformational vs. Reactive Systems

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

> “Our proposed distinction is between what we call transformational and reactive systems.
>
> …
>
> A transformational system accepts inputs, performs transformations on them and produces outputs.
>
> …
>
> Reactive 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?



PDF

(byte stream)

parser

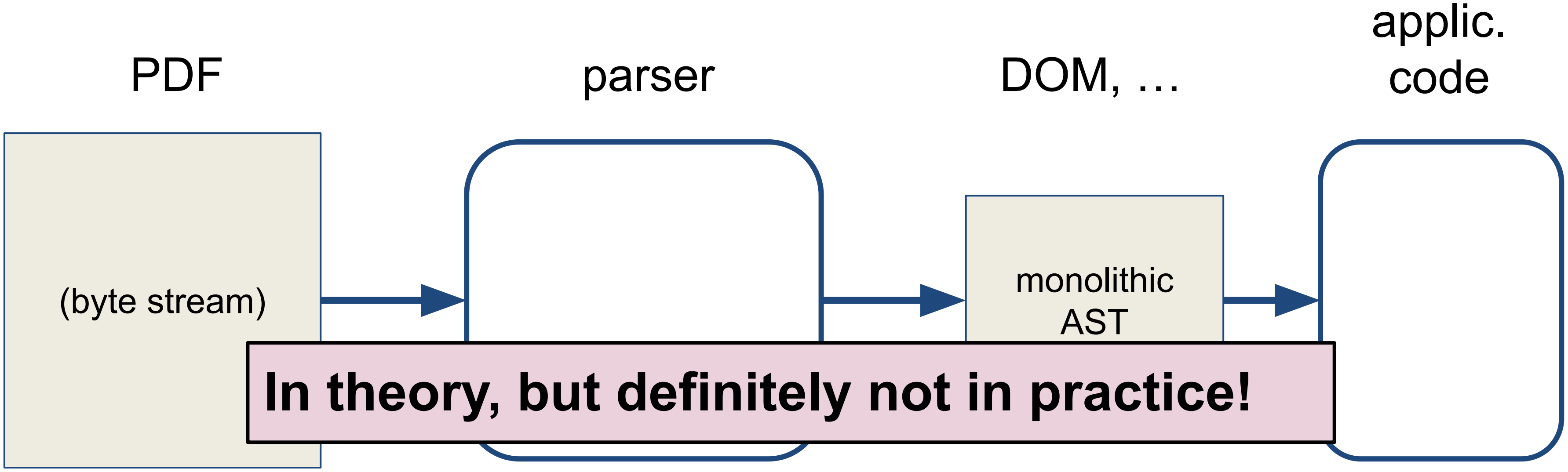
DOM, …

monolithic AST

applic. code

- Transformational System!

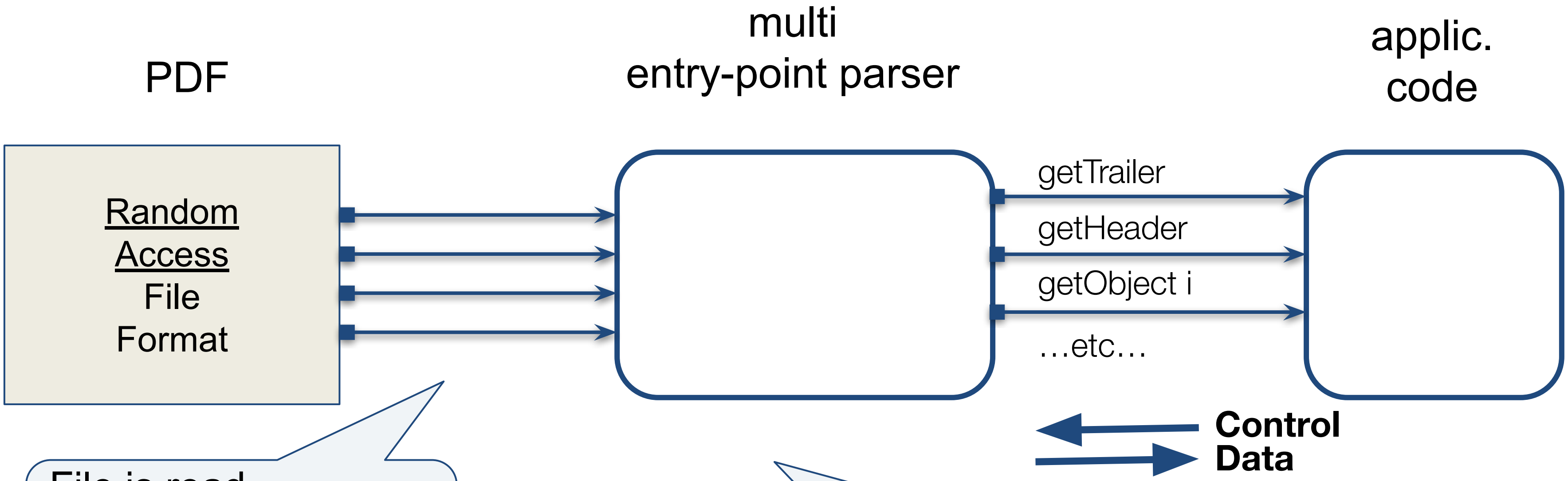
The PDF Problem?



In theory, but definitely not in practice!

- Transformational System!

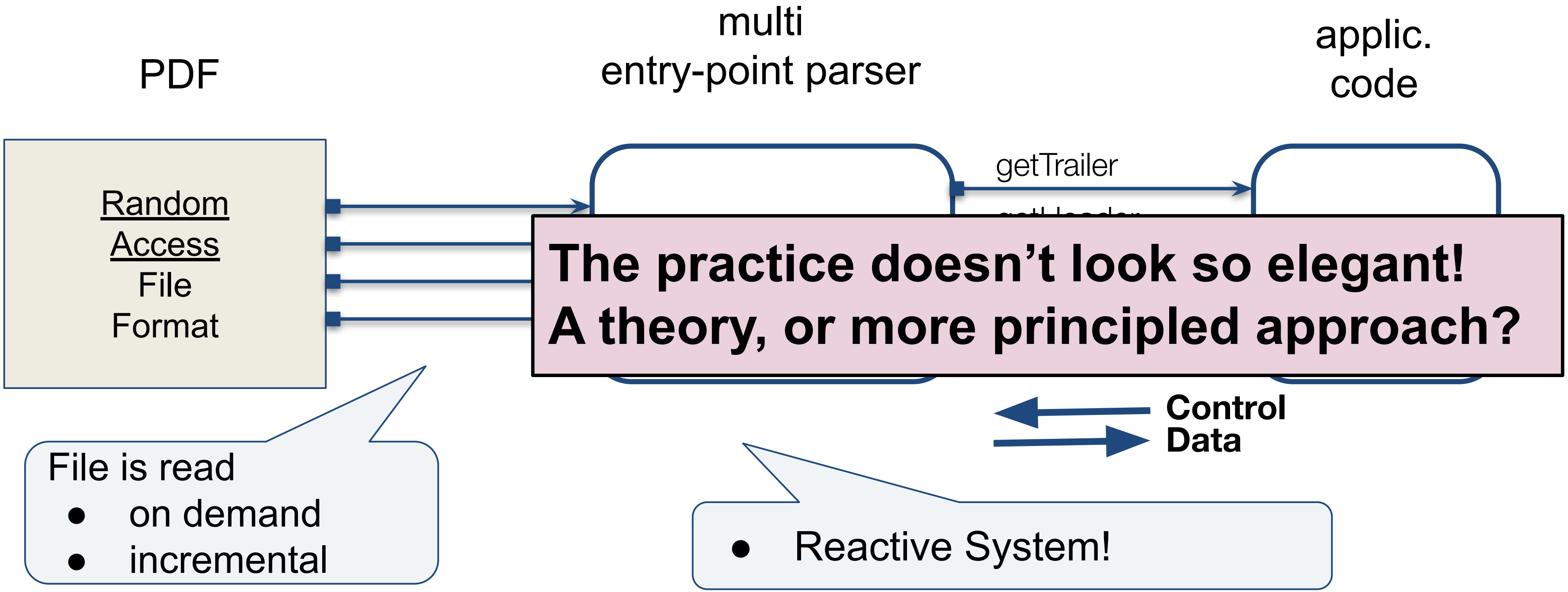
The PDF Problem is Actually …



File is read
- on demand
- incremental

- Reactive System!

The PDF Problem is Actually …



PDF

multi
entry-point parser

applic.
code

Random
Access
File
Format

getTrailer

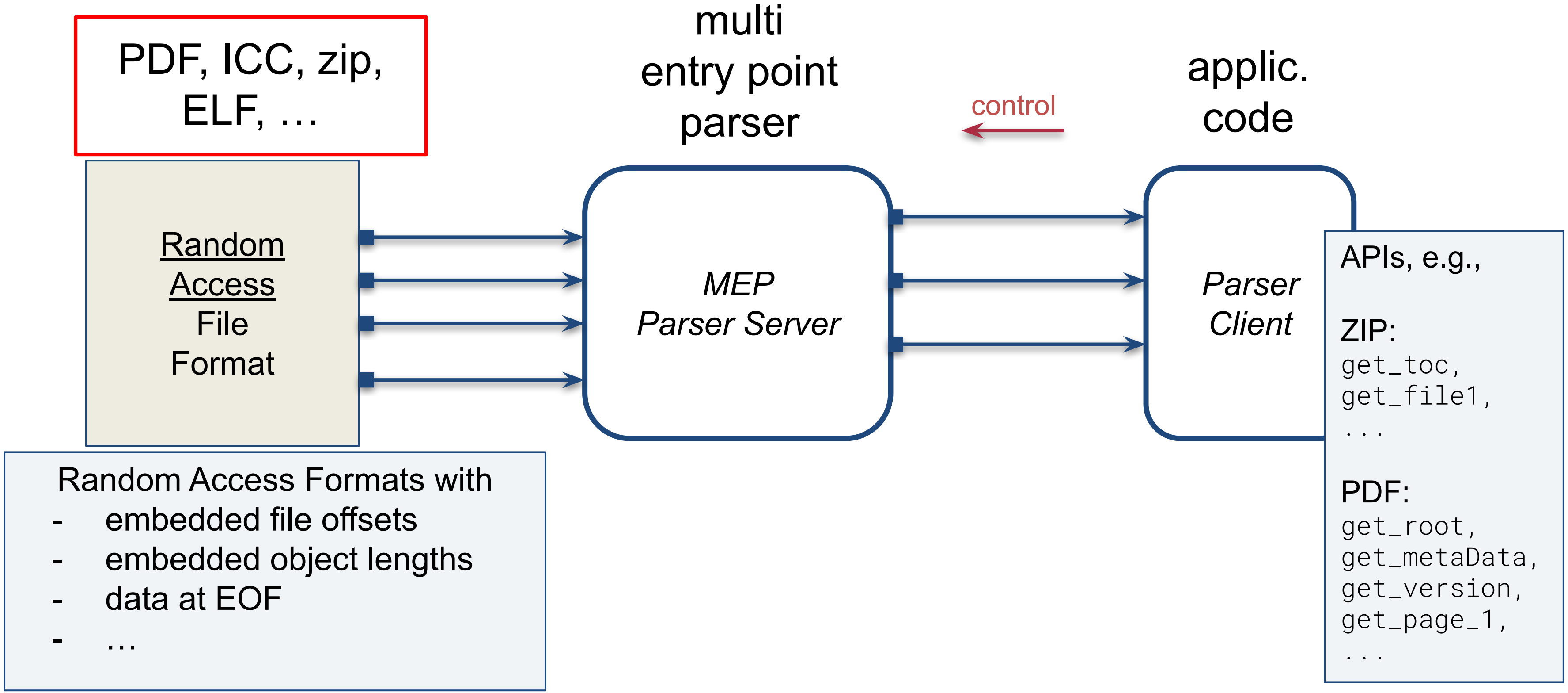
The practice doesn’t look so elegant!
A theory, or more principled approach?

Control
Data

File is read
- on demand
- incremental

- Reactive System!

Random-Access Formats and Multi-Entry-Point Parsers



PDF, ICC, zip, ELF, …

<u>Random</u>
<u>Access</u>
File
Format

multi
entry point
parser

MEP
Parser Server

control

applic.
code

Parser
Client

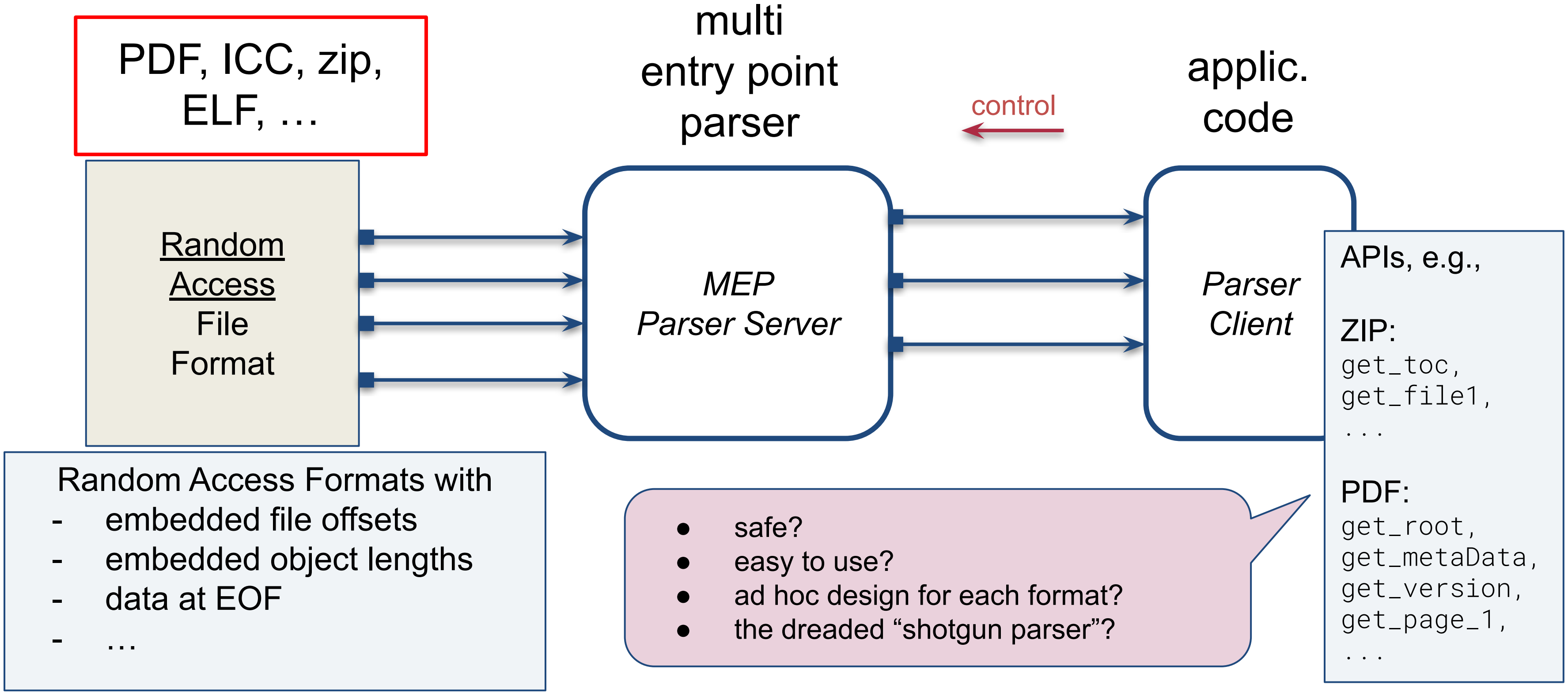
APIs, e.g.,

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
- …

Random-Access Formats and Multi-Entry-Point Parsers



PDF, ICC, zip, ELF, …

<u>Random</u>
<u>Access</u>
File
Format

multi
entry point
parser

MEP
Parser Server

control

applic.
code

Parser
Client

APIs, e.g.,

ZIP:
`get_toc,`
`get_file1,`
`...`

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

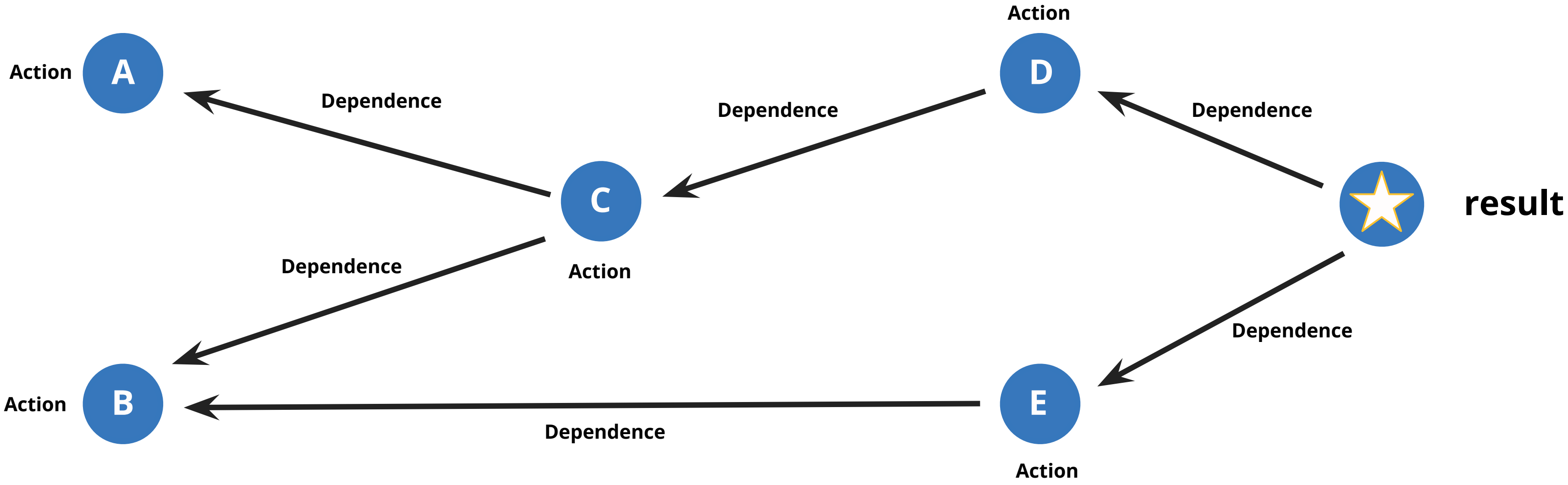
Random Access Formats with
- embedded file offsets
- embedded object lengths
- data at EOF
- …

- safe?
- easy to use?
- ad hoc design for each format?
- the dreaded “shotgun parser”?

Optim(L)

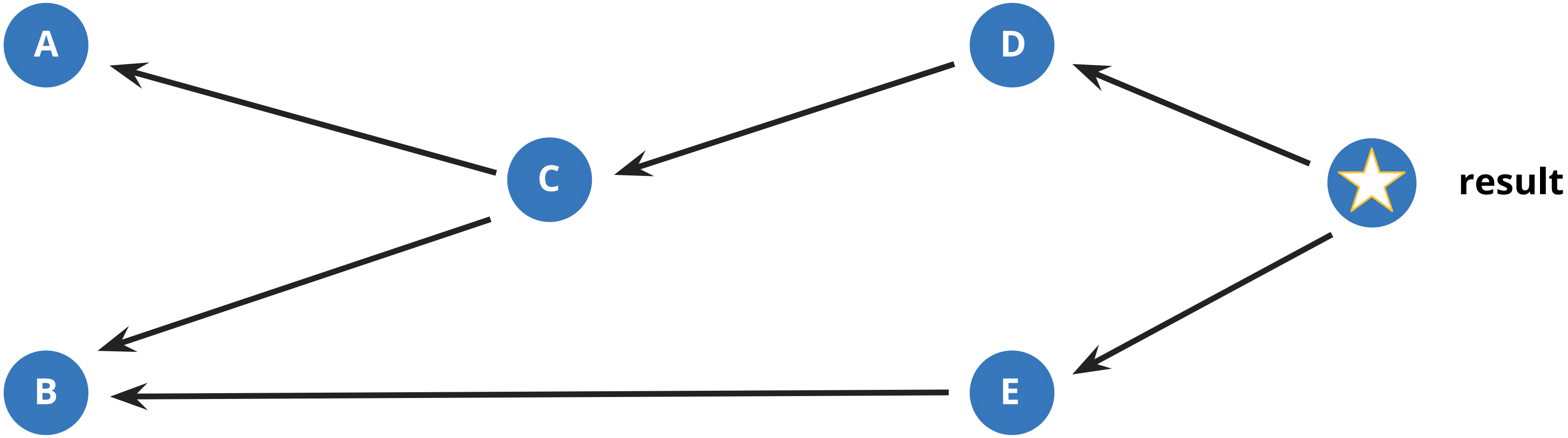
The Directed Acyclic Graph (DAG) view

Traditional, Monolithic Program



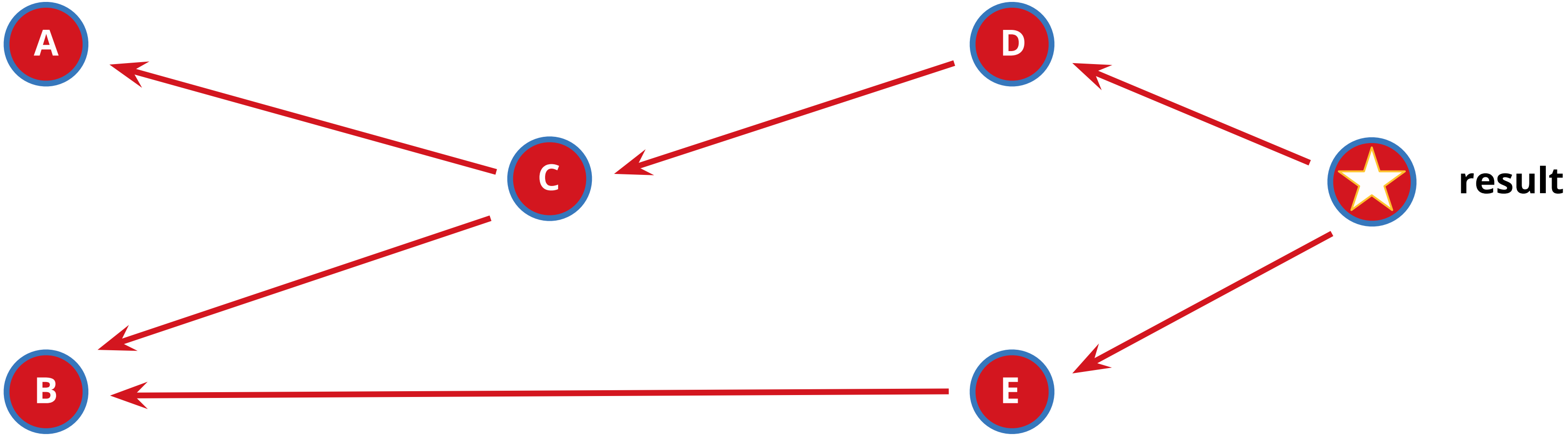
Traditional, Monolithic Program

Initial State: Actions not yet invoked



Traditional, Monolithic Program

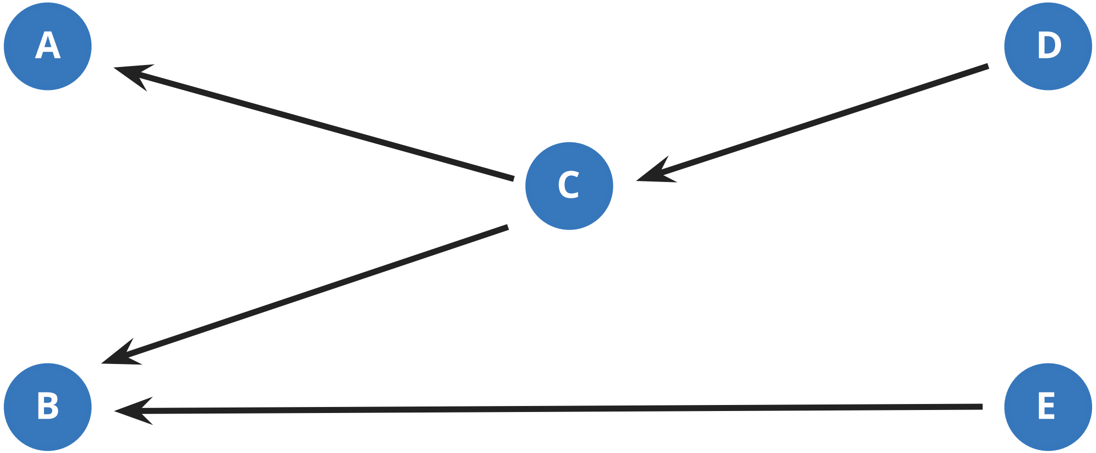
Final state: All Actions Are Invoked



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

Optim(L): Multiple Entry Points

Initial State: Actions not yet invoked

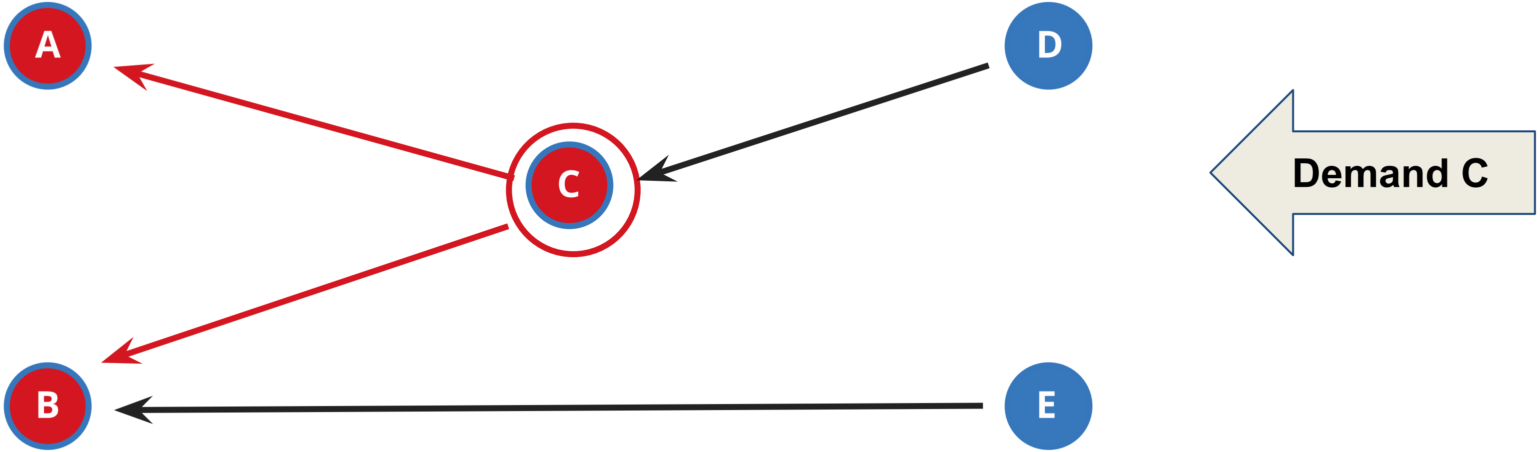


NO single
result/main/...

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

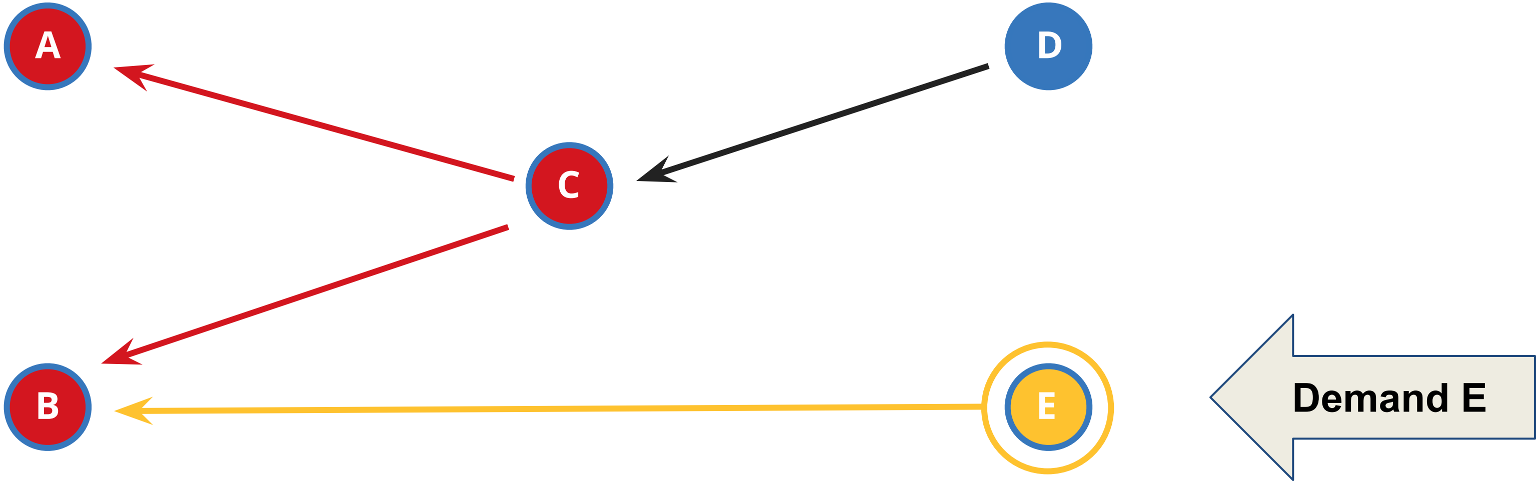
Optim(L): Demands invoke actions & update state

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

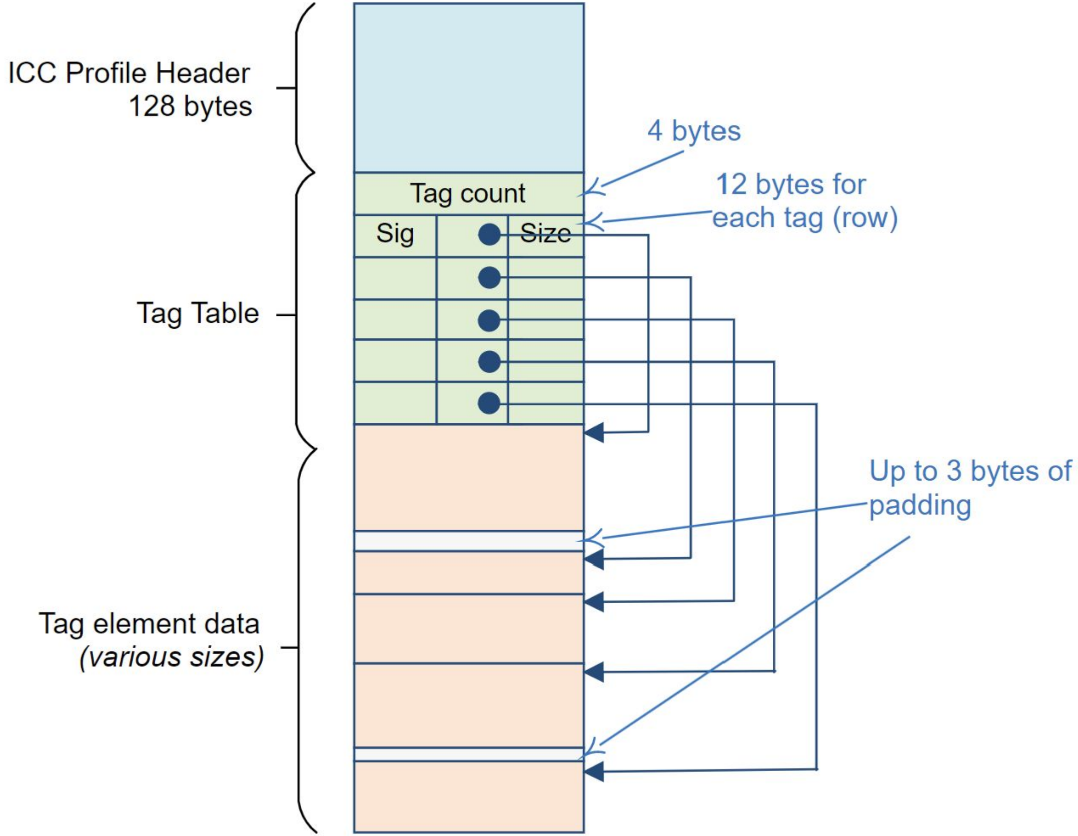
Not the same as “lazy evaluation”:

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

Optim(L)

Let’s see the code.

Example Format: ICC



ICC - International Color Consortium; ICCmax is a color management profile; used in PDF.

ICC, The 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)
```

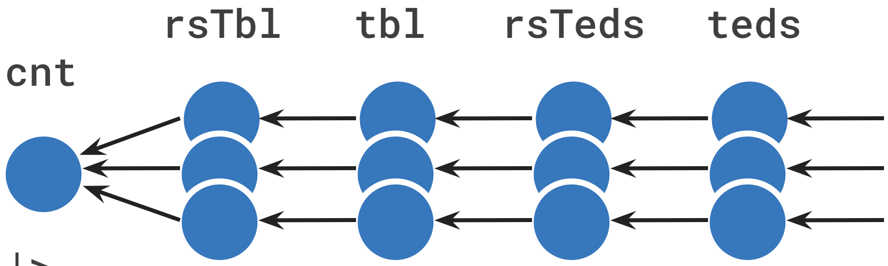
Optim(L), L=XRP

```
[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
```

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)

Regarding Semantics …

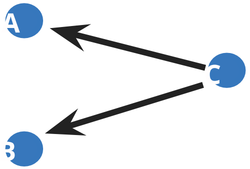
Optim(L): The Theory

Key design decision in Optim(L)!

Optim(L)

- Parameterized over the language '*L*' of computations.
- The language *L* of computations must be a commutative monad: 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

Not commutative monads:

- StateM: mutable globals
- IO

Possibly:

- IO as reader, …

Optim(L): Multiple Interpretations

Generally Optim(L) has a “lazy” interpretation, but others 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, can generate 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

Observationally Equivalence

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

So, a client cannot distinguish these lazy APIs (i.e., the semantics):

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

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

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

Optim(L)

Applied …

Optim(XRP) For Random Access Formats

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

- (eXplicit Region Parser language)
- Three things
  - ReaderException monad.
  - Add explicit, abstract regions
    - I.e., `[startbyte..endbyte]` , but abstract
    - A combinator library for manipulating regions safely
    - Non “sequential parsers” must be applied to a region
  - Top level MEP parser is passed top level abstract region

Achieves

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

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

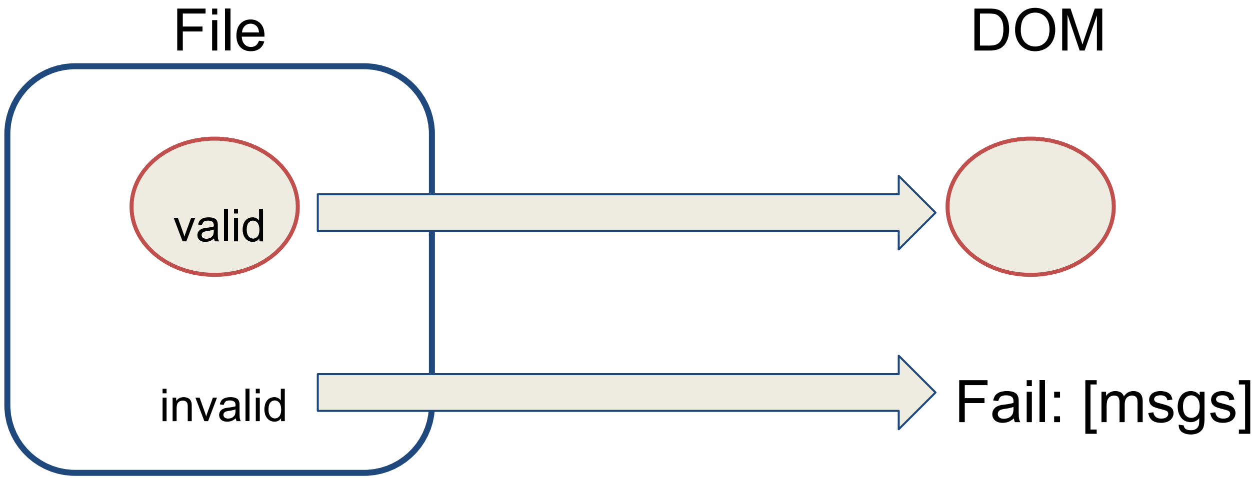
	L	monad	Binding values	We get
1	pure bash	Maybe	FileStream	In program make capability (no persistence)
2	Haskell/_	Identity	a	Lazy API to get/compute globals
3	Haskell/_	Reader	a	Lazy API for accessing global config. data
4	Haskell/_	ReaderMaybe	a	[as above] but allow for failures
5	ML, …	Identity	a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

Optim(L)

Capabilities

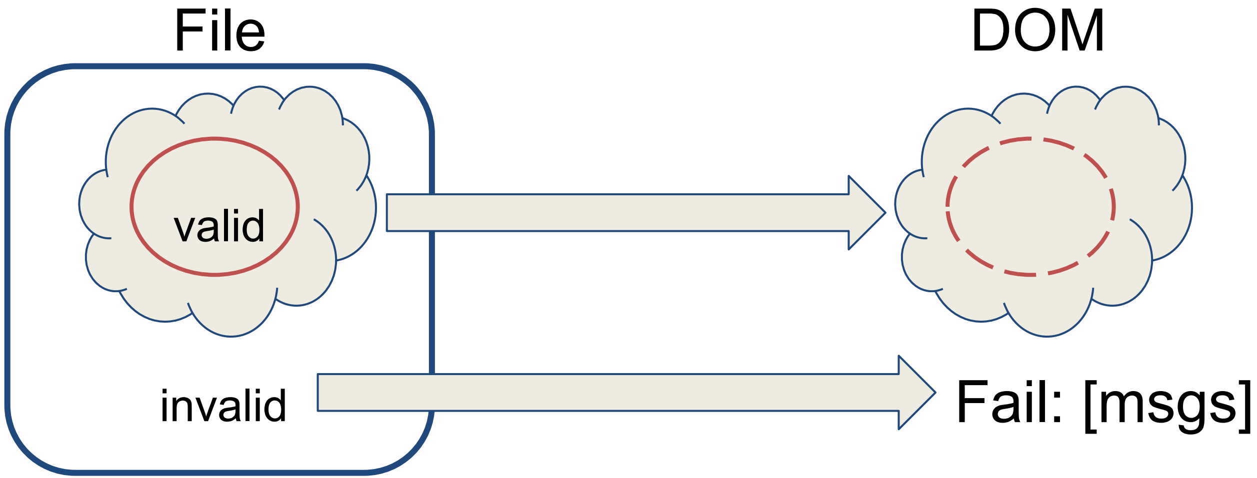
Parser ≠ Validator



Validator:

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

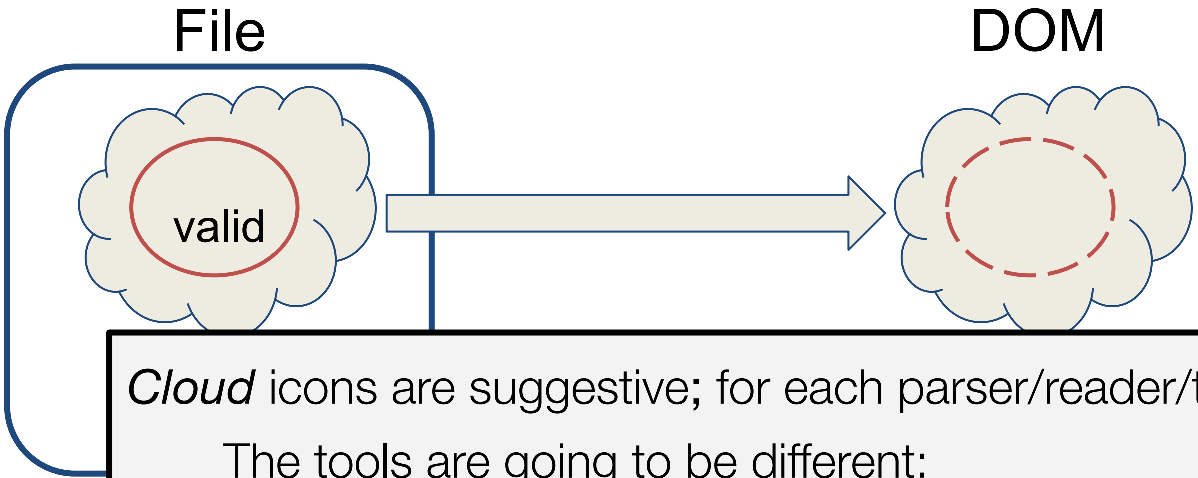
Parser ≠ Validator



Parser:

efficiently, construct the correct DOM when a valid PDF

Parser ≠ Validator



Assuming tools interpret the Standard uniformly!

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

The tools are going to be different:

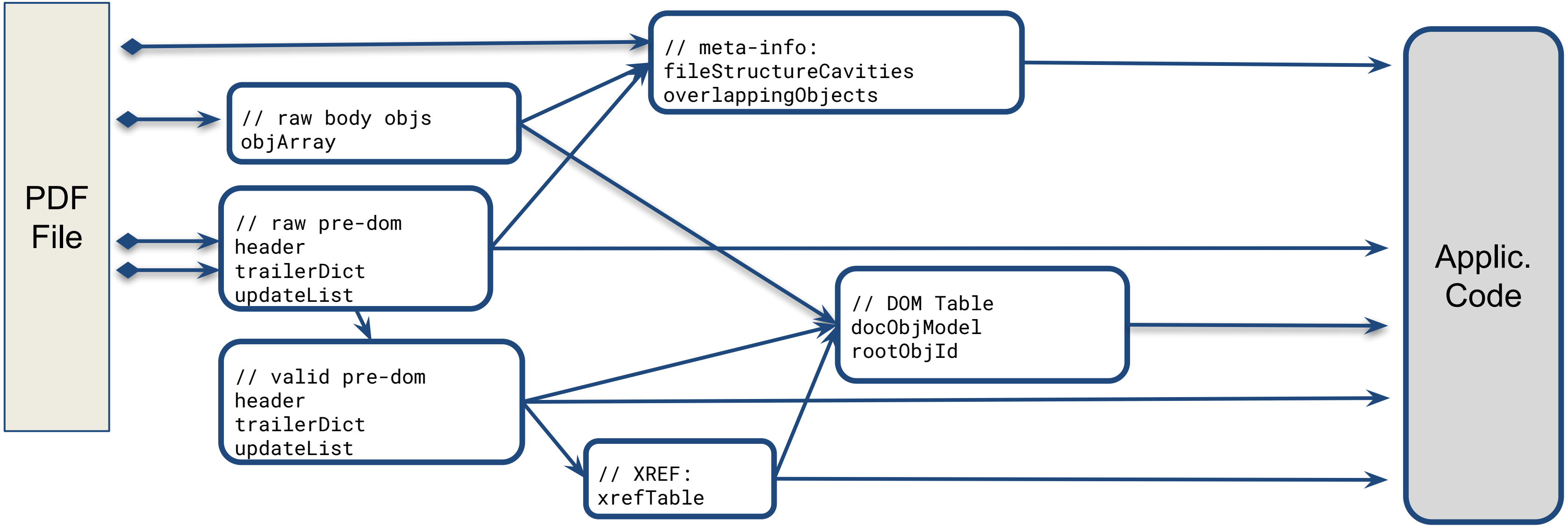
- redundancies in format allow for different choices
- tools in practice allow “minor” errors
- tool may traverse & evaluate implicit data structures differently.

Goal for our “parser specification”:

- Encompass any reasonable & correct cloud

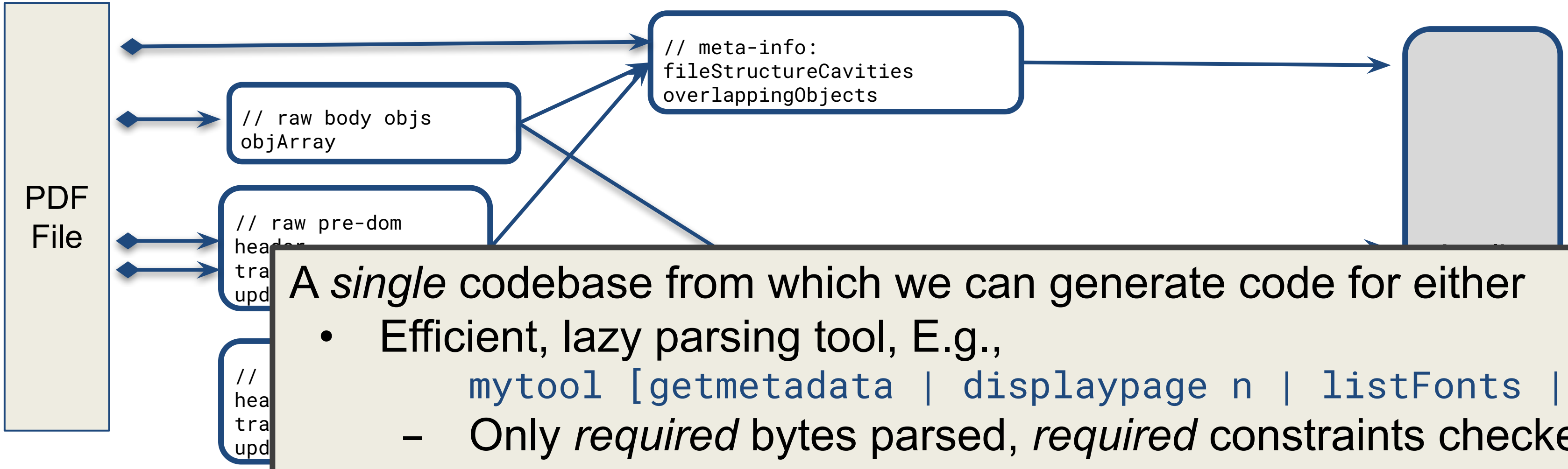
Vision: PDF Library as (DAG of) MEP Components

- 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)

In Conclusion …

Assessments

- We think this is sweet
  - Writing unordered non-IO monadic “bindings”
    - Choosing the interpretation
    - Getting efficient, general, imperative code out
    - Letting our compiler do the dependency analysis
  - Being able to order the bindings semantically, not per data-dependencies.
- Commutative monad restriction
  - Limits scope
  - But this pushed us towards a better design for XRP.

Assessments

- Implementation in Template Haskell
  - Straightforward implementation
  - Types in Optim(L) match types in L
  - Lose some generality, “stuck” with L in Haskell
- Using Haskell, we get different L’s trivially: just use a different monad (user ensures commutative)
- Lazy vectors
  - Must be done in Optim(L), not in L
  - Not too onerous
  - Vector-element laziness very useful.
    - Feels the right “bang for the buck”

Future Developments

- Implement as standalone language and compiler, this allows
    - More optimizations
    - Ability to create multiple tools from one spec. (e.g. validator and parser)
- Optim(XRP): apply to more formats
- Research “bidirectional capabilities”
    - When L is bidirectional, then Optim(L) might be.

Questions?