Remainders and Residuals

Parser Combinators, Sparse Pursuit, and What a Result Leaves Out

Flyxion
Independent Researcher

Abstract

A short functional-programming lecture on parser combinators, of the kind taught in introductory Haskell courses, turns out to contain a compact and technically precise anticipation of a much later formal vocabulary: the four operations POP, REFUSE, BIND, and COLLAPSE that organize the Spherepop calculus. The connection is not that parsing happens to need these operations as tools. It is that the parser type itself, once its remainder component is taken seriously, already distinguishes extraction from refusal, sequencing from insertion, and controlled collapse from mere success. This essay works through that correspondence in detail, using the standard list-of-pairs parser type as a case study, and argues that the recurring theoretical concern with what a computation preserves versus what it destroys was visible, in miniature, in one of the most ordinary corners of functional programming.

1 An ordinary lecture, an unusual amount of structure

Parser combinators are a standard early topic in functional programming courses [5]. A lecturer introduces a type, defines a handful of small functions over it, opens an interpreter, and shows that the definitions behave as expected on a few example strings. Nothing about the presentation announces itself as foundational. The running example is deliberately playful: a string of coffee-cup icons stands in for an unparsed sequence of characters, and the lecturer jokes about a clicker misbehaving whenever the word "monad" is mentioned.

Beneath that informality sits a type that repays close reading [6]:

$$\text{Parser}(A) \;=\; \text{String} \longrightarrow [(A, \text{String})].$$

A parser takes a string and returns a list of pairs. Each pair holds a value of type A, the thing recognized, together with the portion of the string that was not consumed in recognizing it. The lecture arrives at this type by successive refinement: first a bare function from strings to trees, then a function returning a single pair to account for leftover input, then a list of pairs to account for ambiguity or failure, and finally a type parameterized over an arbitrary result rather than a fixed tree.

Read quickly, this looks like ordinary type engineering, the kind of incremental generalization any functional-programming text performs on the way to a working parser library. Read carefully, it already contains a discipline that later, independently developed theoretical work returns to under different names: the discipline of never letting a result stand in for the whole

outcome of an operation. The pair (a, r) insists that a value a is not the entire story; the remainder r, what was left unconsumed, is part of the outcome too. What follows treats that insistence as the essay's central object, and traces its consequences through each of the small parsers the lecture builds.

2 Parsing as an act of distinction

Before any parser runs, an input string is an undifferentiated sequence of characters. It has positions and symbols, but it does not yet have operators, operands, or subexpressions; those are categories a grammar imposes, not properties the string announces on its own. Parsing the string

`"2*3+4"`

into the tree

$$\mathrm{Add}\big(\mathrm{Mul}(2,3),\ 4\big)$$

makes explicit a relation, that multiplication binds tighter than addition, that the flat inscription left implicit. The lecture's coffee-cup metaphor frames this as discovery: the string was "really" a sequence of individual cups all along, and parsing merely reveals what was there. But this framing understates what has happened. A parser does not uncover structure that already existed independently of any interpretive commitment; it applies a grammar, and different grammars can assign different structure to the identical inscription. Without a stated precedence rule, `"2*3+4"` is equally compatible with $\mathrm{Add}(\mathrm{Mul}(2,3),4) = 10$ and with $\mathrm{Mul}(2,\mathrm{Add}(3,4)) = 14$. The string does not choose between these; the grammar does.

This matters for what follows because it fixes the right way to think about every subsequent operation. Parsing is productive: it turns an unresolved sequence into an organized result. But it is productive under a rule, and the rule is external to the string. Distinction, in other words, is an act performed on the input, not a fact merely read off of it. Everything the parser combinators below do, extracting, refusing, sequencing, choosing, inherits this character: each operation commits to a particular way of dividing the world, and that commitment is always available for scrutiny, revision, or rejection under a different grammar.

3 `item`: extraction that keeps its receipt

The simplest parser in the lecture, called `item` and treated at greater length in standard textbook accounts of the same construction [7], does nothing more than take the first character off a string:

$$\mathrm{item}(\epsilon) = [\,], \qquad \mathrm{item}(x : xs) = [(x, xs)].$$

On the string `"abc"`, this returns $[(\texttt{'a'}, \texttt{"bc"})]$: the extracted character, paired with everything that remains.

The pairing is not incidental bookkeeping. A less careful implementation might simply shrink the input, treating `"abc"` as becoming `'a'` the way a stack becomes shorter after a pop, with the rest of the state implicit or reconstructed later from context. The parser instead returns an explicit decomposition: the selected value and the still-available continuation, both present in the same result. Nothing about where the parser stopped needs to be inferred afterward from some other source; it is written into the output.

This is close to the simplest possible instance of a POP operation as later formalized. The parser does not mutate or destroy the original string; `"abc"` is not altered by being handed to `item`. What POP names is not destructive removal but explicit separation: POP exposes a distinguished element together with the residual structure, without requiring the original structure to be mutated or forgotten. The vertical-bar notation sometimes used to write this, $\langle x \mid xs \rangle$, is not just a stylistic choice; the bar marks a boundary between what was selected and what remains selectable, a boundary that a bare function $x \mapsto x$ would erase by returning only x and leaving the separation itself unrecorded.

4 The remainder as continuation, not debris

It would be easy to describe `"bc"` as "leftover" input, in the sense of scrap material discarded after a value has been extracted from it. That description is available but misleading. Operationally, `"bc"` is not waste; it is exactly the state from which the next parsing step proceeds. A second parser applied to `"bc"` does not need to reconstruct where the first parser left off, search for a resumption point, or consult any record outside the returned pair. Here "continuation" means the residual state from which parsing can continue, not a first-class continuation in the narrower continuation-passing-style sense; the word is being borrowed for what it describes, not imported with its full technical apparatus. In that borrowed sense, the continuation has already been handed over, in full, as part of the result.

Chaining several such steps produces a trajectory rather than a single transformation:

$$s_0 \longmapsto (a_1, s_1) \longmapsto (a_2, s_2) \longmapsto \cdots .$$

Each s_i is a state from which further computation remains possible, not a residue left behind by computation already completed. This reframing has a sharp consequence. Suppose two parsers agree on their visible output from some input s:

$$p(s) = [(v, r_1)], \qquad q(s) = [(v, r_2)],$$

but $r_1 \neq r_2$. Both parsers report the same recognized value v. They are nonetheless not interchangeable, because they leave different futures open. One might have consumed more of the input than the other to arrive at the same v; one might permit a continuation the other forecloses. If only the value v is retained and the remainder is discarded, this difference becomes invisible. The two operations look identical because the comparison was only ever performed on the part of the outcome that survived.

This is a minimal, formally exact instance of a claim that recurs throughout later work on admissibility and continuation: output equality does not imply process equivalence,

$$v_p = v_q \;\not\Rightarrow\; p \equiv q,$$

and establishing equivalence, where it is wanted, requires comparing residual state and not only the headline result. The parser lecture never states this as a general principle. It does not need to; the type itself enforces attention to the remainder simply by including it in every returned pair.

4.1 Remainder and residual

A related but nonidentical notion of what remains appears in sparse-representation methods for signal processing, and it is worth a brief detour precisely because the difference is as instructive as the resemblance. In matching-pursuit algorithms, an approximate representation α_k built from k dictionary atoms leaves behind a residual,

$$r_k = y - D\alpha_k,$$

the part of the observed signal y that the current approximation does not yet account for. The next atom to be selected is chosen in relation to this residual, so computation advances through a sequence

$$r_0 \longmapsto (\gamma_1, r_1) \longmapsto (\gamma_2, r_2) \longmapsto \cdots,$$

where each γ_k is a newly selected contribution and each r_k records what the approximation so far fails to explain.

The analogy to a parser's remainder is structural, not literal, and the difference matters. A parser remainder is an unconsumed suffix of a discrete input, a piece of the string that has simply not yet been looked at. A pursuit residual is a numerical discrepancy, an error vector computed by subtracting a reconstruction from an observation; it is not unexamined material so much as a measured shortfall. Despite that difference, both quantities serve the same guarding function. Neither lets an intermediate result pass as a finished one. Omit the pursuit residual, and an approximation can be reported without disclosing how much of the signal it fails to explain; omit the parser remainder, and a prefix parse can be reported without disclosing how much of the input it leaves uninterpreted. In both cases, the quantity that would reveal incompleteness is exactly the quantity a careless report is most tempted to drop.

5 Failure without fabrication

The lecture's second simple parser always fails:

$$\text{failure}(s) = [\,].$$

Whatever the input, it returns the empty list. This is worth pausing on, because an empty list is a specific and disciplined kind of answer, not an absence of an answer. The parser does not guess a value under pressure to produce something; it does not silently substitute a default; it reports, in a form the rest of the system can inspect and act on, that no result meeting its rule was established from this input.

That is a description of refusal, not of incompetence. A parser built to recognize integers, given the string `"abc"`, correctly returns the empty list. Nothing has been disproved about `"abc"` in general; what has failed is one particular attempted relation, between this parser's grammar and this input. The same string might succeed under a parser for identifiers. Refusal, on this reading, is always typed and local:

$$\text{REFUSE}_{\text{integer}}(\texttt{"abc"}) \quad \text{does not entail} \quad \text{REFUSE}_{\text{identifier}}(\texttt{"abc"}).$$

Treating the empty list as a first-class, structured outcome, rather than as an exceptional condition to be caught and explained away elsewhere, is what keeps a single local failure from being read as a global verdict on the input. This is the same discipline that a REFUSE primitive is

later built to enforce explicitly: a failed attempt to establish a distinction should be recorded as a failed attempt under a specific rule, not silently converted into either a fabricated success or an unscoped condemnation of the material that was offered.

The discipline is nonetheless only partial, and it is worth being exact about where it stops. The empty list records that a parser failed, but it does not record why. Two parsers that fail for entirely different reasons, one because the input was empty, another because the first character violated a specific grammatical rule, produce the identical result:

$$\text{failure}_p(s) = [\,] = \text{failure}_q(s),$$

even though p and q refused for unrelated reasons. The empty list preserves exactly one distinction, success versus failure, while collapsing every distinction among the ways failure can occur. A richer refusal type would keep a diagnostic alongside the outcome,

$$\text{Result}(A) = \text{Success}(A, r) \mid \text{Refusal}(\rho, r),$$

where ρ names the rule or condition that was violated. The always-failing parser in the lecture is an excellent model of explicit nonproduction. It is not yet a fully informative refusal in the sense the Spherepop primitive eventually demands; it anticipates REFUSE only partially, in the same way that `item` anticipates POP only partially; both are minimal analogues, not finished instances, of the calculus built around them.

6 `return`: introducing a value without any claim to have found it

The third simple parser, conventionally called `return`, takes a value and produces a parser that always succeeds with it, consuming nothing:

$$\text{return}(v)(s) = [(v, s)].$$

Applied to `"abc"`, return(`'d'`) yields $[(\texttt{'d'}, \texttt{"abc"})]$: success, with the input left entirely untouched.

The contrast with `item` is instructive precisely because the two can produce outputs that look alike once the remainder is dropped. Compare

$$\text{item}(\texttt{"abc"}) = [(\texttt{'a'}, \texttt{"bc"})]$$

with

$$\text{return}(\texttt{'a'})(\texttt{"abc"}) = [(\texttt{'a'}, \texttt{"abc"})].$$

Both report the value `'a'`. But the first is a claim about the input: an `'a'` was recognized there, and the recognition consumed it. The second makes no such claim; it supplies an `'a'` from elsewhere, and the input is exactly as it was before. To see the two results collide, define a value-forgetting map on singleton-success lists,

$$\text{value}\big([(a, r)]\big) = a,$$

which discards the remainder and keeps only the recognized value. Applying it to both results erases the distinction between them:

$$\text{value}\big(\text{item}(\texttt{"abc"})\big) = \text{value}\big(\text{return}(\texttt{'a'})(\texttt{"abc"})\big) = \texttt{'a'}.$$

Only the remainder, which value has just thrown away, tells the two apart. This is a compact demonstration of why provenance is not, in general, recoverable from a finished value alone. Once the state component of a result is discarded, an act of extraction and an act of insertion can become observationally indistinguishable, even though they mean entirely different things about where the value came from and what it obligates the next step to do.

7 `bind`: dependency carried forward, not merely juxtaposed

The lecture stops short of naming the full sequencing operation, but the parser type leads to it directly, in the same spirit as the general monadic treatment of functional programming [9, 10]. Given $p : \text{Parser}(A)$ and a function $f : A \to \text{Parser}(B)$, their combination is

$$(p \gg\!= f)(s) \;=\; \text{concat}\big[\, f(a)(r) \mid (a, r) \leftarrow p(s) \,\big].$$

This is deliberately written as a list comprehension followed by concatenation rather than as a set-theoretic union. The result of $p(s)$ is a list, with a definite order and definite multiplicity, not a set; using $\bigcup$ instead would quietly convert list into set, silently discarding duplicate derivations that happen to produce identical (a, r) pairs and erasing the order in which alternatives were found. Both of these are exactly the kind of information this essay is arguing a careful account of parsing should not discard without comment, so the definition of `bind` itself has to preserve them. The first parser establishes a value a together with a remainder r; the function f is then applied to a to choose a second parser, and that second parser runs on r, not on the original s. Schematically,

$$s \xrightarrow{\;p\;} (a, r) \xrightarrow{\;f(a)\;} (b, r').$$

This is a literal binding: the value a is not simply passed along as an argument to some independently chosen next step, it determines which next step is chosen, and the state that step operates on is exactly the state the first step left behind. A digit-recognizing parser feeding an operator-choosing function, for instance, makes the second parsing decision genuinely conditional on what the first one found, rather than merely sequential with it.

The distinction matters because ordinary function composition can look superficially similar while lacking this property. Two operations run one after another are not automatically bound in this sense unless the second is chosen by, and operates on the residue of, the first. `bind` is the operation that converts a pair of otherwise independent recognizers into an organized dependency, and it does so using nothing more than the remainder that was already present in the parser type.

8 Ordered choice as a controlled collapse

The lecture defines one further combinator, an ordered alternative between two parsers:

$$(p \,\square\, q)(s) = \begin{cases} q(s), & p(s) = [\,], \\ p(s), & p(s) \neq [\,]. \end{cases}$$

q is tried only if p fails outright. This is a reasonable and widely useful rule. It is also, on inspection, a genuine act of collapse, and one that deserves more scrutiny than its convenience usually receives.

Suppose both parsers could, in principle, interpret the same input:

$$p(s) = [(a, r_p)], \qquad q(s) = [(b, r_q)].$$

It is worth being precise about what $(p \,\square\, q)(s)$ actually does here, because two different operations could be described by the loose phrase "reduces two possibilities to one," and this combinator performs only one of them. One operation, call it enumerative collapse, would first assemble both results into a single ledger and then select from it:

$$[(a, r_p), (b, r_q)] \;\longmapsto\; [(a, r_p)].$$

That is not what ordered choice does. Evaluating $(p \,\square\, q)(s)$ evaluates $p(s)$ first, and once $p(s)$ succeeds, $q(s)$ is simply never evaluated at all. Call this procedural foreclosure:

$$p(s) \neq [\,] \implies q(s) \text{ is not examined.}$$

The alternative (b, r_q) is not disproved, not compared, not weighed against (a, r_p) and found wanting; the question of whether q would have succeeded is simply never asked. It remains true, extensionally, that $q(s)$ would evaluate to $[(b, r_q)]$ if someone bothered to check, but nothing internal to the combinator's execution ever represents that fact. This is, if anything, a sharper form of the concern than enumerative collapse would be: at least a ledger that is built and then pruned leaves a record of what was pruned. Procedural foreclosure converts priority directly into control flow before any common ledger of alternatives is ever assembled, so there is no ledger for a later step to consult, and no record that a choice was even made.

None of this makes left bias itself suspect. In real parser combinators, trying p before q is normally deliberate and often encodes a real grammatical decision, for instance that a keyword should be recognized before a more general identifier rule is allowed to consume it. Left bias is not itself an inadmissible collapse. It becomes unjustified only when implementation order is allowed to stand in for a grammatical priority that was never independently specified, that is, when the fact that p happened to be written first, rather than any stated rule about which interpretation should win, is doing the work of deciding the outcome. The mere fact that p succeeded is not, by itself, evidence that q was inadmissible; it is evidence only that p was tried first and that trying it was sufficient to stop the search.

The lecture makes this collapse easier to overlook by an additional, quietly consequential decision: it restricts attention throughout to parsers whose result list has length zero or one. The general type,

$$p(s) = [(a_1, r_1), \ldots, (a_n, r_n)],$$

can represent several simultaneously live interpretations, a genuine possibility space. Choosing to work only with the empty-or-singleton fragment of that type does not remove ambiguity from parsing in general; it removes ambiguity from the representation, before any particular input has been examined. Whatever plurality a real grammar might need to express has to be handled elsewhere, or not handled at all. This is exactly the kind of question that a later, more general theory of admissible collapse is built to keep visible: not whether collapsing multiple candidates to one is ever legitimate, it very often is, but what warrants that collapse in a given case, and whether the warrant was ever actually supplied or merely assumed by the shape of the type being used.

It is worth being explicit about what, exactly, is being criticized here, since the two things are easy to run together. Within the lecture's own worked examples, no parser ever actually returns

more than one result; there is, in the running examples themselves, no second candidate sitting unexamined for ordered choice to foreclose. The critique above is therefore a critique of the type's designed capacity, not a report of something observably lost in any particular example the lecture works through. It says that a mechanism built to select among several live possibilities, the general list-valued type, is being operated in a regime, empty-or-singleton results only, where it can never actually be called on to select among anything, and that this makes the collapse easy to overlook precisely because it costs nothing in every case the lecture happens to show. Whether that is a defect depends on the task: for the lecture's stated purpose, teaching parser combinators, it costs nothing, because a grammar of arithmetic expressions with clear precedence genuinely does not need to represent live ambiguity. The point is not that the lecture's parsers are secretly ambiguous and the lecture hides it. It is that the type is capable of representing ambiguity, the capability is real and stated in the type signature, and a reader could reasonably come away from the lecture without ever noticing that capability exists, because nothing in the examples ever exercises it.

9 Lists as a ledger, not merely a convenience

The lecturer offers a stated reason for using lists rather than the more restrictive `Maybe` type, echoing a much older argument for representing failure and multiple successes as a list [8]: lists compose easily with existing list operations, and the resulting code is, in the lecturer's judgment, more elegant. That is a fair implementation-level argument. But the list type carries a further significance beyond convenience. An empty list records that no continuation was established. A singleton records that exactly one was. A list of length greater than one would record that several remain live, each retaining its own value and its own remainder:

$$[(a_1, r_1), \ldots, (a_n, r_n)] \quad — \quad \text{several admissible continuations, side by side.}$$

`Maybe` can express only the first two of these cases; it distinguishes failure from a single success but has no room to hold open plurality once it exists. The list type, in its full generality, is a small ledger of admissible parses rather than a container that happens to be either empty or full. The lecture's decision to restrict itself to the empty-or-singleton fragment of this type, discussed above as a form of collapse, can now be seen from the other side: it keeps the syntax of a possibility ledger while committing, by construction, to never actually recording more than one entry in it. The richer structure is present in the type and latent in the code, but it is never exercised.

10 Sparse pursuit and the warrant for collapse

A more rigorous model of what a warranted collapse requires comes from a different field entirely: Michael Elad's work on sparse and redundant representations for signal and image processing [3]. The comparison is offered here as a mathematical parallel, not as part of the historical lineage traced elsewhere in this essay; nothing in Elad's work descends from, or feeds into, the parser-combinator lecture. What it supplies is a setting in which the question ordered choice raises only informally, what actually warrants selecting one candidate over another, has been worked out with full mathematical precision.

Suppose an observed signal $y \in \mathbb{R}^n$ is represented using a dictionary $D \in \mathbb{R}^{n \times m}$, so that

$$y = D\alpha$$

for some coefficient vector α. When $m > n$, the dictionary is overcomplete, typically built or learned to contain more atoms than are strictly needed to span the signal space [1]. Under an overcomplete dictionary, more than one coefficient vector can synthesize the identical observed signal:

$$D\alpha_1 = D\alpha_2 \;\not\Rightarrow\; \alpha_1 = \alpha_2.$$

Agreement at the level of the reconstructed object, in other words, does not establish agreement at the level of the representation that produced it. This is the signal-processing counterpart of two parsers returning the same semantic value while differing in derivation or in the continuation each one leaves open: the finished result alone underdetermines the process.

Sparse-representation methods resolve this underdetermination not by discovering a hidden fact about which α is correct, but by adding an explicit selection principle. One seeks the coefficient vector with the fewest nonzero entries that still reconstructs the observation within an acceptable tolerance,

$$\widehat{\alpha} \in \arg\min_{\alpha} \|\alpha\|_0 \quad \text{subject to} \quad \|y - D\alpha\|_2 \leq \varepsilon,$$

and practical pursuit algorithms approximate this generally intractable optimization either by selecting atoms greedily or by relaxing the nonconvex ℓ_0 objective to a tractable surrogate [2]. In the vocabulary developed above, a pursuit algorithm is not a single extraction; it is a repeated cycle of POP, BIND, and COLLAPSE. An atom is selected from the dictionary, its coefficient is bound into the developing representation, and the field of representations still consistent with the observation is narrowed according to the stated sparsity objective.

What this comparison sharpens is the distinction, only gestured at in the discussion of ordered choice, between a collapse that merely succeeds and a collapse that is warranted. Any α satisfying $y = D\alpha$ succeeds in the weak sense of reproducing the observation. That alone establishes nothing about whether the representation is unique, sparse, stable under small perturbations of y, or recoverable from noisy measurements. Sparse-representation theory studies exactly these further conditions, properties of the dictionary's structure and bounds on the sparsity of α, under which a selected representation can be shown to be the unique or reliably recoverable one. The warrant for collapse, in other words, is never supplied by successful reconstruction on its own; it is supplied by a further relation among the representation, the dictionary, the selection rule, and the recovery conditions the theory establishes.

Ordered choice has the same logical shape, stated far less precisely. That the first parser succeeds shows only that it reaches an admissible local result; it does not show that no other parser could have reached an equally admissible one. A grammar's stated priority can select the first parse in exactly the way a sparsity objective selects one coefficient vector, but, as with sparsity, the priority rule has to be treated as part of the model, argued for on its own terms. It cannot be read off after the fact from the single result that happened to survive, once every alternative it would have needed to be compared against has already disappeared.

11 Analysis, synthesis, and negative structure

Elad and collaborators draw a further distinction that bears directly on this essay's account of REFUSE: between sparse synthesis models and cosparse analysis models, two descriptions of

structure that were once treated as close to interchangeable but are not [4]. In the synthesis model already introduced above, a signal is characterized by the existence of a small set of active generators,

$$x = D\alpha, \qquad \|\alpha\|_0 \text{ small.}$$

The representation records which small collection of atoms suffices to build x. In the analysis model, an operator Ω is applied directly to the signal itself,

$$z = \Omega x, \qquad \|z\|_0 \text{ small,}$$

and the signal is characterized instead by the many rows of Ω under which it produces no response. Synthesis records a small set of contributors that are present; analysis records a large set of relations under which the signal is, in effect, silent.

This distinction gives the essay's account of REFUSE a sharper edge than the parser lecture alone can supply. An empty or vanishing response need not indicate an absence of structure. Under an analysis model, the specific pattern of operators to which an object returns zero is itself part of what characterizes the object; the zeros are not gaps in a description but constitutive of it. This is exactly the caution the earlier discussion of REFUSE was reaching for when it insisted that refusal be typed rather than treated as a single undifferentiated failure. A bare empty list, failure$(s) = [\,]$, says only that something returned nothing. A ledger recording which operator, applied under which rule, returned zero, in the manner Ω records which of its rows a signal fails to activate, turns that nothing into a structured, and in principle informative, pattern of refusal.

12 Completion as a further, separable condition

A parser can succeed while leaving material unconsumed:

$$p(s) = [(a, r)], \qquad r \neq \epsilon.$$

Whether this counts as an adequate result depends entirely on what the parser was for. If the task is to recognize a prefix, it is a success outright. If the task is to interpret an entire document, it is not yet one; something remains uninterpreted. A parser aimed at full-document interpretation therefore needs an additional condition layered on top of local success.

For the deterministic, singleton-or-empty parsers the lecture works with, that condition can be stated simply:

$$\text{complete}(p, s) \iff \exists a.\ p(s) = [(a, \epsilon)].$$

For the fully general, list-valued parser type, however, this single predicate quietly conflates two different questions, and it is worth separating them once the possibility of more than one result is back on the table. The first is whether a complete parse exists at all among the candidates returned:

$$\text{hasComplete}(p, s) \iff \exists a.\ (a, \epsilon) \in p(s).$$

The second is whether that complete parse is the only candidate returned:

$$\text{uniquelyComplete}(p, s) \iff p(s) = [(a, \epsilon)] \text{ for exactly one } a.$$

A parser that returns one complete parse alongside an incomplete alternative satisfies hasComplete but not uniquelyComplete; the single-predicate definition above would simply call it incomplete, which may or may not be the intended judgment depending on whether the incomplete

alternative should count against the result. Finding one way to account for the whole input does not, by itself, establish that it is the only way, and a definition that only ever checks for existence would silently discard that further fact. This separates two questions that are easy to run together even in the deterministic case: whether a candidate structure was formed at all, and whether that structure accounts for everything the task required it to account for. A parser that quietly discards the remainder, reporting only a, effectively converts every local success into an apparent complete one, whether or not any material was actually left over. The information that something went uninterpreted is not merely omitted from the report; it becomes structurally impossible to recover from the report, because the report no longer contains it. This is a small but exact instance of a more general failure mode: a result can look finished simply because the part of the outcome that would have shown it was unfinished was never carried forward.

13 What a tree does, and does not, preserve

The parse tree is an improvement over the flat string in one important respect and a loss in another. It makes explicit relations, precedence, grouping, that the original inscription only implied. But it does not, by itself, preserve the history of how it was obtained. Two different parsing procedures, one perhaps trying and discarding several alternatives before settling on a derivation, another following a single unambiguous path, can arrive at the identical tree:

$$p(s) = [(t, \epsilon)], \qquad q(s) = [(t, \epsilon)].$$

If only t is kept, these two histories become indistinguishable after the fact. For evaluating an arithmetic expression, this loss is usually harmless; the tree is all that is needed to compute a value. For auditing a decision, tracing a scientific inference, or establishing that a derivation is unique rather than merely one of several that happened to converge, it is not harmless at all.

A parser willing to retain more could return a triple instead of a pair,

$$[(t, r, \pi)],$$

where π records the derivation actually taken. Equality of the resulting tree would then no longer imply equality of the route to it:

$$t_p = t_q \;\not\Rightarrow\; \pi_p = \pi_q.$$

This is the same gap, restated at the level of trees rather than characters, between possessing an answer and possessing a warranted route to that answer. The ordinary parser type does not track π at all; it was never designed to. Noticing the absence is not a criticism of the lecture's design choices for its own limited purpose. It is a way of seeing exactly where a design choice made for one purpose, evaluating expressions correctly, quietly forecloses a different purpose, auditing how a particular structure came to be accepted, that the same underlying data could in principle have supported.

14 The asymmetry of preserved information

The preceding sections have treated the parser type's retention of the remainder as a virtue, and largely by contrast with what a cruder implementation might discard. It is worth stating plainly,

before moving on, that this virtue is narrow and specific rather than general. The parser pair (a, r) preserves exactly one boundary: recognized value versus unconsumed suffix. It does not, as ordinarily used, preserve several other boundaries that the discussion above has already had occasion to want.

It does not separately preserve the consumed lexeme, the literal span of the input that was read in producing a, apart from a itself; for a parser that transforms as well as recognizes, the source text and the semantic value it was mapped to are not the same thing, yet only the latter survives. It does not preserve alternative derivations that were tried and discarded, nor, as the discussion of ordered choice showed, alternatives that were never tried at all because an earlier branch foreclosed them. It does not preserve the reason for a failure, only the fact of one. And it does not preserve anything about the cost of reaching the result, how much search, backtracking, or computation was required to produce (a, r) as opposed to some other candidate.

Writing this out schematically makes the asymmetry explicit:

$$\underbrace{(a, r)}_{\text{preserved}} \;\Big/\; \underbrace{(\ell, \pi, \rho, \phi, c)}_{\text{ordinarily discarded}} \;,$$

where ℓ is the consumed source span, π the derivation, ρ the refusals actually examined, ϕ the alternatives foreclosed without examination, and c the computational cost. The parser type is not, in other words, simply preservationist, a design that keeps everything and discards nothing. It makes a specific, and in context quite reasonable, decision about which single boundary is worth carrying forward and which several others are not. That decision constitutes a preservation budget, chosen once, at the level of the type, before any particular input or grammar is considered.

This reframing changes what the essay's central analogy is claiming. The point is not that the ordinary parser type happens to anticipate Spherepop's primitives while some other, more careless type would not have. The point is that every representation of a computational outcome, the parser type very much included, makes exactly this kind of decision, and the decision is itself an act of admissibility: a judgment, made in advance, about which distinctions are worth the cost of carrying and which can be safely, or not so safely, allowed to collapse. The parser type earns its place as a case study not because it is unusually generous with what it preserves, but because the one boundary it does insist on, the remainder, is enough to make the general shape of that trade-off visible without any additional apparatus. A type that preserved nothing would teach the lesson too weakly to notice; a type that preserved everything would teach it too diffusely to isolate. The parser type, preserving exactly one thing and discarding the rest by default, sits at the point where the trade-off can be seen clearly.

15 From parsing a string to parsing an interpretive system

An ordinary parser asks a single, well-scoped question: what structure does this string have, under this grammar? A different, and considerably more demanding, family of questions can be asked about the parsing process itself: which grammar was actually applied, which distinctions did it recognize, which portion of the input did it consume, which alternatives did it reject or never examine in the first place, what continuation remains available afterward, and can the accepted result be traced back to a specific derivation rather than merely asserted?

Nothing in the lecture answers these questions, and it would be a mistake to read the lecture as gesturing toward them. But the ordinary parser type already supplies the raw material a more

demanding version would need. Where an ordinary parser returns

$$(a, r),$$

a parser built to answer the further questions above might return something closer to

$$(a, r, \pi, \rho, \alpha),$$

where π records the derivation chosen, ρ records alternatives that were considered and rejected, and α records ambiguity that was noticed but not yet resolved. The ordered-choice combinator examined above shows that ρ alone is not enough. An alternative can fail to appear in the final result for at least three distinct reasons: it was examined and accepted, it was examined and refused, or it was never examined because an earlier branch already succeeded and foreclosed it. The third status is not a case of ρ; nothing was considered and rejected, because nothing was considered at all. A more adequate epistemic state therefore needs its own field for foreclosure, separate from refusal:

$$E = \langle\, a,\ r,\ \pi,\ \rho_{\text{refused}},\ \phi_{\text{foreclosed}},\ \alpha_{\text{unresolved}} \,\rangle,$$

a structured account of what was decided, what was ruled out, what was never reached, and what is still open, rather than a bare answer. This distinction matters for exactly the reason the previous section raised: an alternative disproved by evidence is not equivalent to an alternative bypassed by control flow before it was ever evaluated, and a result type without a $\phi_{\text{foreclosed}}$ field has no way to tell the two apart. Ordinary parser combinators are the minimal case of this larger family, the case in which every field except the first two, the accepted value and the remainder, has been silently dropped.

16 Stating the correspondence directly

The four operations that organize the Spherepop calculus can now be matched, term for term, against the machinery examined above.[1]

POP extracts a distinguished element from a larger structure while preserving what remains:

$$\text{POP}(x :: xs) = \langle x, xs \rangle.$$

`item` provides the minimal parser analogue of POP. It shares POP's essential separation of a selected element from a preserved residue, though nothing in the lecture guarantees that it models every feature a full Spherepop POP might carry, such as identity, addressability, or substrate-specific ledger effects; the correspondence is structural, not a claim that the two definitions coincide in every respect.

REFUSE records that no admissible extraction was established under a given rule, without fabricating a result or condemning the input beyond the scope of that rule:

$$\text{REFUSE}(p, s) \iff p(s) = [\,].$$

[1]The full specification of Spherepop's four primitives, including their treatment of admissibility, confluence, divergence, and replay, is given elsewhere as a separate, self-contained calculus; POP, REFUSE, BIND, and COLLAPSE are its sole primitive operations, and properties such as admissibility or equivalence are analyses of histories built from them rather than additional primitives. This essay draws on only as much of that specification as the comparison below requires, and does not attempt to restate it in full.

The always-failing parser, and any parser that correctly returns the empty list rather than a guess, models the outer shape of REFUSE: a disciplined, structured non-answer rather than a fabricated one. As the earlier discussion of diagnostic collapse noted, it models only the outer shape; the empty list cannot yet distinguish one reason for refusal from another the way a fuller REFUSE with an attached ρ would.

BIND makes a subsequent operation depend on a value already established, while forwarding the residual state that the first operation left behind:

$$\text{BIND}(p, f)(s) = \text{concat}\big[\, f(a)(r) \mid (a, r) \leftarrow p(s) \,\big].$$

This is the sequencing operation the parser type supports, whether or not the lecture gives it that name, provided list order and multiplicity are preserved rather than collapsed into set union.

COLLAPSE selects, or commits to, one result out of a larger field of candidates:

$$\text{COLLAPSE}\{(a_i, r_i)\}_{i=1}^{n} = (a_k, r_k).$$

Ordered choice models one particular route to a COLLAPSE-shaped outcome, the procedural-foreclosure route, whenever more than one parser could, in principle, have succeeded on the same input; it reaches a single committed result without ever assembling the field $\{(a_i, r_i)\}$ that the COLLAPSE notation above presents as already given. It is, either way, an operation whose legitimacy depends on a warrant that the parser type itself does not supply.

What Spherepop adds, on this reading, is not a new operation absent from ordinary computation, but a demand that these four already-occurring operations be named and inspected rather than left buried inside the ordinary control flow of a program. A typical parser implementation extracts, refuses, sequences, and collapses constantly, in the course of doing entirely conventional work. It almost never says so. The remainder, once dropped, takes the evidence of all four operations with it.

17 Why an introductory lecture is the right place to find this

It would be convenient, but inaccurate, to present parser combinators as a deliberate anticipation of a later formal theory. They are not that. They are a small, self-contained piece of functional-programming pedagogy, built to teach a specific technique, with no stated interest in refusal, provenance, or collapse as general concepts. The correspondence traced here is not a hidden intention recovered from the lecture; it is a structure that was present in the type all along, available to be noticed by anyone who took the remainder as seriously as the lecture's own type signature insists on doing.

That is, in fact, the more interesting observation. A later theoretical vocabulary organized around distinction, refusal, binding, and collapse did not need to be imported from some more exotic source to find its first concrete instance. It was already sitting inside one of the most standard exercises in an introductory course, in a type most students learn to write correctly within the first hour without ever being asked what its second component is protecting them from losing. The lesson generalizes past parsing. Any operation that turns an unresolved input into an organized result, an extraction, a computation, a decision, a summary, can be examined by the same four questions a parser combinator answers whether or not it is asked to: what did it recognize, what did it refuse, what did it commit forward, and what did it collapse away in producing the answer it gave.

References

[1] Michal Aharon, Michael Elad, and Alfred Bruckstein, "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation," *IEEE Transactions on Signal Processing*, vol. 54, no. 11, pp. 4311–4322, 2006. doi:10.1109/TSP.2006.881199.

[2] Alfred M. Bruckstein, David L. Donoho, and Michael Elad, "From Sparse Solutions of Systems of Equations to Sparse Modeling of Signals and Images," *SIAM Review*, vol. 51, no. 1, pp. 34–81, 2009. doi:10.1137/060657704.

[3] Michael Elad, *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, New York, 2010. doi:10.1007/978-1-4419-7011-4.

[4] Michael Elad, Peyman Milanfar, and Ron Rubinstein, "Analysis Versus Synthesis in Signal Priors," *Inverse Problems*, vol. 23, no. 3, pp. 947–968, 2007. doi:10.1088/0266-5611/23/3/007.

[5] Erik Meijer, "Functional Parsers," in *Functional Programming Fundamentals*, Chapter 8 of 13, C9 Lectures, Microsoft, 2009. Available at: `https://learn.microsoft.com/en-us/shows/c9-lectures-erik-meijer-functional-programming-fundamentals/c9-lectures-dr-erik-meijer-functional-programming-fundamentals-chapter-8-of-13`.

[6] Graham Hutton and Erik Meijer, "Monadic Parsing in Haskell," *Journal of Functional Programming*, vol. 8, no. 4, pp. 437–444, 1998. doi:10.1017/S0956796898003050.

[7] Graham Hutton, *Programming in Haskell*, 2nd ed. Cambridge University Press, Cambridge, 2016.

[8] Philip Wadler, "How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages," in Jean-Pierre Jouannaud, ed., *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, vol. 201, Springer, Berlin and Heidelberg, pp. 113–128, 1985.

[9] Philip Wadler, "The Essence of Functional Programming," in *Proceedings of the Nineteenth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, ACM, New York, pp. 1–14, 1992. doi:10.1145/143165.143169.

[10] Philip Wadler, "Monads for Functional Programming," in Johan Jeuring and Erik Meijer, eds., *Advanced Functional Programming*, Lecture Notes in Computer Science, vol. 925, Springer, Berlin and Heidelberg, pp. 24–52, 1995. doi:10.1007/3-540-59451-5_2.