Supplement to "An Empirical Analysis of Ad Hoc Parsers in Python"

Anonymous Authors

Definition 1. The ad hoc parser of a designated string-typed input variable is the set of program statements, contained within a single function, that read this input variable, or any string-typed variable transitively derived from it, whose hidden grammar comprises exactly those input values that do not cause any of these statements to fail.

1 Approach

We now present a general approach to locate all ad hoc parsers present in a program, based on collecting program slices conforming to definition 1. The program slices collected in this way capture the core of each ad hoc parser, beginning with the appearance of the parser's input string and ending at the point(s) where no more transformation of that string or its sub-strings occur. All operations within such a slice (i.e., all operations contained in an ad hoc parser by our definition), are immediately *grammar-relevant*: they determine the program's accepted values for the parser's input string. While any parsed (structured) data coming out of an ad hoc parser might be constrained further downstream (or by a caller up in the call-chain), that is outside the scope of our present operational definition. The delineation between ad hoc parsing and application logic is fluid—a defining characteristic of ad hoc parsing—but we want to firmly plant our feet on the rock-solid island of syntactic parsing amidst the sea of semantic constraints.

Algorithm 1 shows the entire approach: given a program $\mathcal{P}$ and a string oracle Ω, construct the program's control-flow graph G and add data dependence information to it, mark all string data flow using the oracle, and collect all program statements connected by string flows into a set of ad hoc parser slices $\mathcal{A}$.

Algorithm 1 Locating the set of ad hoc parsers $\mathcal{A}$ contained in a program $\mathcal{P}$ using a marking oracle Ω.

procedure LOCATE-PARSERS($\mathcal{P}, \Omega$)
  $G \leftarrow \text{CFG}(\mathcal{P})$
  ADD-DATA-DEPENDENCES(G)
  MARK-STRING-EDGES(G, Ω)
  $\mathcal{A} \leftarrow$ COLLECT-PARSER-SLICES(G)
  return $\mathcal{A}$

1.1 Control Flow

How to represent a program's control flow obviously varies strongly by programming language. To remain as general as possible, we make very few demands on the exact shape and nature of the control-flow graph.

Definition 2. A *control flow graph* (CFG) is a directed multi-graph G with nodes corresponding to program statements and *control flow edges* representing the possible transfer of control from one program statement to another. We assume that control flow edges are labelled, if necessary, to disambiguate multiple outgoing flows from a single node. A CFG may contain additional "synthetic" nodes not directly corresponding to any program statement. In particular, we assume every CFG contains one unique ENTRY node and one unique EXIT node and that each node in a CFG lies on some path from ENTRY to EXIT.

We use the following notation to refer to various properties of the CFG:

$x \xrightarrow{l}_{\text{cf}} y$	a control flow edge from node x to node y with label l
$\text{paths}_{\text{cf}}(x, y)$	all control flow paths from node x to node y
$\text{pred}_{\text{cf}}(x)$	the control flow predecessors of node x
$\text{nodes}(G)$	the set of all nodes in G

The CFG of a Python program can be constructed by a (mostly) top-down traversal of the program's abstract syntax tree (AST). Simple statements (such as assignments or standalone function calls) may be translated directly to CFG nodes. The nodes of consecutive statements in the program are connected by simple unlabeled control flow edges. Compound statements (such as `if`, `while`, `try`, `def`, etc.) need to be considered according to their semantics and might introduce multiple or no nodes of their own but will usually add multiple labelled control flow edges. For example, an `if` statement adds a node for its test expression, with two edges labelled `True` and `False`, representing the possible outcomes of the test. The `True` edge leads to the first statement in the body of the `if` statement, and the `False` edge leads to either the first statement in an `else` branch,[1] if one exists, or otherwise to the next statement after the whole `if` statement. Semi-structured control flow (induced by `break`, `continue`,

[1] In the Python AST, `elif` clauses are already simplified into nested `if-else-if` statements.

and exception handling) requires some additional bookkeeping during the AST traversal. For example, a `break` statement terminates the nearest enclosing loop, ceding control to the first statement after that loop, except if it passes control out of a `try` statement with a `finally` clause, in which case the statements in the `finally` clause are executed before the first statement after the loop. Exception handling is a particularly complicated aspect of Python's semantics and care must be taken to accurately represent the somewhat baroque control flow it occasionally introduces. For example, if there are multiple possible paths through the finally block (i.e., the continuation after finally differs based on how we entered it), then we add a pseudo exit node to the finally block that "dispatches" based on how the block was entered. This avoids having to duplicate the finally body.

1.2 Data Dependence

The *data dependence graph* (DDG) models relationships between program statements in terms of variable usage. It assumes the existence of certain variable usage information, which can be collected during CFG creation:

$$\begin{aligned}\text{defs}(v) &= \{x \in \text{nodes}(G) \mid v \text{ is defined in } x\}\\ \text{uses}(v) &= \{x \in \text{nodes}(G) \mid v \text{ is used in } x\}\\ \text{kills}(v) &= \{x \in \text{nodes}(G) \mid v \text{ is killed in } x\}\end{aligned}$$

A variable is taken to be *defined* when it is first introduced or when a new value is stored in it; it is *used* whenever its value is read; and it is *killed* when it is re-defined or when some other mechanism removes it from scope (e.g., the Python `del` statement).

Definition 3. A node y is *data dependent* on a node x if there exists a variable v that is defined in x and used in v and there is at least one path from x to y in which v is not killed,

$$\begin{aligned}x \overset{v}{\dashrightarrow} y \iff{}& x \in \text{defs}(v) \wedge y \in \text{uses}(v) \wedge\\ & \exists p \in \text{paths}_{\text{cf}}(x, y)\colon p \setminus \{x, y\} \parallel \text{kills}(v).\end{aligned}$$

Algorithm 2 gives a simple procedure for adding data dependences to a CFG. Because it is based on computing reachable nodes, this algorithm is quadratic in the number of nodes, $\mathcal{O}(|V|^2)$. While faster implementations are possible,[2] the version presented here is entirely sufficient in practice, including for large-scale analysis.

[2] E.g., using bit vectors to compute all reaching definitions simultaneously in $\mathcal{O}(|V| \cdot |E|)$.

Algorithm 2 Add data dependence to a CFG.

```
procedure ADD-DATA-DEPENDENCES(G)
    for all v ∈ vars(G) do
        for all x ∈ defs(v) do
            W ≔ {x}
            V ≔ ∅
            R ≔ ∅
            while W ≠ ∅ do
                choose y ∈ W
                W ≔ W \ {y}
                for all y →cf z do
                    R ≔ R ∪ {z}
                    if z ∉ V then
                        V ≔ V ∪ {z}
                        if z ∉ kills(v) then
                            W ≔ W ∪ {z}
            for all y ∈ uses(v) ∩ R do
                add edge x -v-→ y
```

1.3 String Types

To find all of the program's string variables together with all relevant string data flow, while remaining suitably general and not dependent on a particular programming language's type inference capabilities, we introduce a global string type inference algorithm parameterized over a local string oracle. The oracle makes a determination of a particular variable's string-ness at a particular program point based on that variable's syntactic use within the relevant program node in the CFG; it has access to current knowledge about the string-ness of other variables within that same node, but it does not otherwise look beyond a single program node.

Definition 4. The *marking lattice* $\mathcal{M}$ is a three-element chain

$$\mathsf{N} \prec \mathsf{Y} \prec ?$$

of marks N (no), Y (yes), and ? (maybe), equipped with a binary *meet* operation $\sqcap = \min$. As a finite chain, $\mathcal{M}$ is a bounded distributive lattice with least element N and greatest element ?.[3]

Let G be the CFG under analysis, with

$E_{dd} = \{* \dashrightarrow *\}$	the set of data dependence edges,
$V = \text{nodes}(G)$	the set of nodes, and
$Var = \text{vars}(G)$	the set of variables declared in G.

[3] $\mathcal{M}$ is order-isomorphic to the conjunction reduct of Kleene's three-valued logic, except that its top element is the undetermined mark rather than a definite truth value.

We extend data dependence edges by markings drawn from $\mathcal{M}$, and write

$$x \overset{v}{\underset{m}{\dashrightarrow}} y$$

for a data dependence edge with mark $m \in \mathcal{M}$. The set E_{dd} is now the set of marked data dependence edges, ordered pointwise by the lattice order $\preceq$. Furthermore, we introduce the notion of a variable's *role* within a node, which can be either as a data source ($\uparrow$) or a data sink ($\downarrow$).

Definition 5. The *marking oracle* Ω is a function

$$\Omega^r : V \times \mathit{Var} \times \wp(E_{dd}) \to \mathcal{M}$$

parameterized over a variable role r. It takes as input a node, a variable, and the set of marked data dependence edges incident to the node, and it returns a mark indicating whether the variable, as it occurs within the node in the stated role, is string-typed (the oracle's *verdict*).

We assume an oracle satisfies four properties: purity, monotonicity, sound negatives, and plausible positives. These properties ensure the overall correctness of the global marking algorithm while allowing the oracle itself to stay local (i.e., analyzing one program statement at a time). To formulate these properties, and later propositions concerning the marking algorithm, we will use the notation

$$x;\, E \vdash v :^r \mathbb{S} \quad \text{and} \quad x;\, E \vdash v :^r \not\mathbb{S}$$

to mean that in node x under edge markings E, treating variable v in role r as string-typed is *locally consistent* or respectively *locally inconsistent*. In the former case, no use of v within x in role r contradicts the assumption that v carries a string value; in the latter, there exists some use of v within x in role r that is incompatible with v being a string. These are local syntactic judgments made by the oracle about a single program statement; the marking algorithm is responsible for assembling local judgments into a globally consistent assignment of string typings.

Property 1 (Purity)**.** Each oracle is a *pure function* of its arguments, in the sense that two calls to the oracle with the same variable in the same role at the same node, with the same incident edge markings, return the same mark.

Property 2 (Monotonicity)**.** Each oracle is *monotone*, such that

$$\forall r, x, v, E, E'. \quad E' \preceq E \implies \Omega^r(x, v, E') \preceq \Omega^r(x, v, E).$$

Lowering an input mark along the lattice may never raise a verdict. If the oracle has more information about surrounding variables, its answer can only get more precise (i.e., the verdict can only ever move downward in the lattice, from ? to Y to N).

Property 3 (Sound Negatives)**.** A N verdict is *sound*, meaning

$$\Omega^r(x, v, E) = \mathsf{N} \implies x;\, E \vdash v :^r \not\mathbb{S}.$$

If the oracle returns N, then treating v as a string in x in role r is locally inconsistent under edge markings E. The oracle may only return N if it can locally reason that the r-use of v in x is syntactically incompatible with v being a string. No such obligation is placed on Y or ?, which are allowed to be assigned optimistically—it is up to the marking algorithm to resolve any contradictions.

Property 4 (Plausible Positives)**.** A Y verdict is *plausible*, meaning

$$\Omega^r(x, v, E) = \mathsf{Y} \implies x;\, E \vdash v :^r \mathbb{S}.$$

If the oracle returns Y, then treating v as a string in role r is locally consistent under edge markings E. The oracle may only return Y if it can locally reason that the r-use of v in x is syntactically compatible with v being a string. In particular, if v could never be a string in the given context, then the oracle must not return a Y verdict. The oracle need not be omniscient: this is a purely local determination that might be contradicted globally by the marking algorithm.

Remark. Sound negatives ensure that the oracle never falsely excludes a genuine string variable, while plausible positives ensure that a Y verdict carries real (local) evidence of string-typing. However, the oracle is never obligated to make a positive verdict: if it determines that the variable could be a string but is unlikely to be so (for whatever reason), then it may safely return ?.

The marking algorithm is a global fixpoint computation over the marking lattice that propagates string-ness verdicts over the data dependence graph. It refines initially optimistic markings and descends monotonically toward a globally sound assignment of edge markings. The procedure initializes all data dependence edges with the mark ? and seeds the worklist W with all source nodes. It then repeatedly selects a node x and iterates through all variables v leaving x. Based on the source mark for v at x, it forms the consensus over all sibling edges, that is, the meet of the source mark with the combined meets of all sink marks for v at the data flow successor nodes of x. All sibling edges are then lowered to the consensus; if any mark actually changes, the nodes whose oracle verdicts could depend on it are then returned to the worklist.

Proposition 1 (Termination)**.** *Mark-String-Edges terminates for arbitrary oracles, irrespective of the oracle properties.*

Proof. The algorithm loops over program nodes in the worklist W by removing one node on each iteration and possibly adding multiple others; the algorithm terminates when W is empty. A node enters W only on initialization or immediately after some edge mark has been updated. An edge mark m is only ever updated by $m' = c \sqcap m$ with $m' \neq m$. By definition of the lattice meet $m' \preceq m$, which the inequality forces to $m' \prec m$. Thus, at most $2|E_{dd}|$ updates can occur. Because each update adds a bounded number of nodes to W there are finitely many insertions into W in total. As the loop consumes one node from W per iteration, it can only run finitely many times. □

Algorithm 3 Mark all data flow edges transporting strings. Note that we implicitly assume an oracle call to always receive the relevant set of edges with the current state of markings.

procedure MARK-STRING-EDGES(G, Ω)
  for all $x \overset{v}{\dashrightarrow} y$ **do**
    set edge mark $x \underset{?}{\overset{v}{\dashrightarrow}} y$
  $W \coloneqq \{x \mid x \overset{v}{\dashrightarrow} y\}$
  while $W \neq \emptyset$ **do**
    choose $x \in W$
    $W \coloneqq W \setminus \{x\}$
    for all $v \in \mathrm{vars}(x)$ **do**
      if $m = \mathsf{N}$ **for all** $x \underset{m}{\overset{v}{\dashrightarrow}} y$ **then**
        continue
      $m_x \coloneqq \Omega^{\uparrow}(x, v)$
      $c \coloneqq m_x \sqcap \left(\bigsqcap_{x \underset{m}{\overset{v}{\dashrightarrow}} y} \left(m \sqcap \Omega^{\downarrow}(y, v) \right) \right)$
      for all $x \underset{m}{\overset{v}{\dashrightarrow}} y$ **do**
        $m' \coloneqq c \sqcap m$
        if $m' \neq m$ **then**
          update edge mark $x \underset{m'}{\overset{v}{\dashrightarrow}} y$
          $W \coloneqq W \cup \{y\} \cup \mathrm{pred}_{\mathrm{dd}}(x) \cup \mathrm{pred}_{\mathrm{dd}}(y)$
          if $m_x \neq \mathsf{N}$ **then**
            $W \coloneqq W \cup \{x\}$

Proposition 2 (Fixpoint)**.** *The final set of marked edges E^* produced on termination is a fixpoint: re-examining any node against E^* (i.e., running an oracle with E^* as input) yields no change.*

Proof. A node is removed from W only by being processed, and is re-inserted whenever an incident edge changes. On termination, $W = \emptyset$, so no incident edge has changed for any node since it was last processed and that last processing resulted in no changed marks. By the property of oracle purity, verdicts depend only on the set of marked edges, which is now the unchanging E^*, so re-examination of any node cannot result in any changed marks. □

Proposition 3 (Confluence)**.** *The final set of marked edges E^* is independent of the order that nodes are selected from the worklist W and equals the greatest marking $E \preceq \top$ closed under the oracle.*

Proof. Each oracle is assumed to be a fixed monotone function of the marking, so a full pass of Mark-String-Edges is a monotone reductive operator on the finite lattice of marked data dependence edges E_{dd}. Through the worklist, only and exactly those nodes whose incident edges have changed are re-examined, making each pass a fair chaotic iteration. Iterating a monotone reductive operator downward from $\top$ (the initial state where all edge markings are ?) yields a decreasing chain whose limit is the greatest fixpoint. Because it is fair, the chaotic iteration reaches the same limit regardless of selection order. □

Proposition 4 (Soundness of N)**.** *Every edge $x \xrightarrow[\mathsf{N}]{v} y$ in the final set of marked edges E^* guarantees that variable v is not a string source in node x nor a string sink in node y,*

$$x \xrightarrow[\mathsf{N}]{v} y \in E^* \implies x;\, E^* \vdash v :^{\uparrow} \not\$ \;\wedge\; y;\, E^* \vdash v :^{\downarrow} \not\$.$$

Proof. An edge $x \xrightarrow{v} y$ is marked N because either (a) the source oracle for v at x returned N, or (b) one of the sink oracles for a sibling edge returned N. By sound negatives, this means that treating v as a string is locally inconsistent in the respective position. In case (a), if the variable cannot be defined as a string at its source, then it is obviously not a string at any of its sinks. In case (b), if the variable cannot be used as a string at some sink y', then it also cannot be defined as a string at the source x, lest there be a type inconsistency. And since all sibling edges share the same source, the variable thus cannot be a string at any of its other sinks. □

Remark. In a dynamically typed language such as Python, a single variable v may carry different values along different data flow paths—indeed, the same variable may carry a string value in one branch of the program while never admitting strings in another. While such a construction ostensibly generates some string data flow and would seem to contradict proposition 4, the actual type of v must correctly be considered a union of all possible types it can assume along all possible execution paths and is thus no longer a simple string type;

the algorithm may soundly mark it N. Note that this does not contradict an oracle optimistically assuming values with union types that include the string type to be strings: that is a local oracle determination consistent with the oracle properties and any global contradictions are later resolved by the marking algorithm.

Corollary 1 (Completeness)**.** *Let E^*_m denote non-overlapping subsets of E^* limited to edges with mark m. From proposition 4, it follows that the joint set $E^*_{\mathsf{Y}} \cup E^*_{?}$ is complete with respect to string data flow: every edge along which we may treat v as a string at both source and sink is contained therein,*

$$x;\, E^* \vdash v :^{\uparrow} \mathbb{S} \;\wedge\; y;\, E^* \vdash v :^{\downarrow} \mathbb{S} \implies x \overset{v}{\dashrightarrow} y \in E^*_{\mathsf{Y}} \cup E^*_{?}.$$

In particular, no edge that could carry a string variable is marked N.

Proposition 5 (Justification of Y)**.** *Every edge $x \underset{\mathsf{Y}}{\overset{v}{\dashrightarrow}} y$ in the final set of marked edges E^* is* justified *by the oracle, in the sense that,*

(a) treating v as a string has not been found locally inconsistent,

$$x \underset{\mathsf{Y}}{\overset{v}{\dashrightarrow}} y \in E^* \implies \Omega^{\uparrow}(x, v, E^*) \neq \mathsf{N} \;\wedge\; \Omega^{\downarrow}(y, v, E^*) \neq \mathsf{N},$$

(b) treating v as a string has some global plausibility,

$$x \underset{\mathsf{Y}}{\overset{v}{\dashrightarrow}} y \in E^* \implies x;\, E^* \vdash v :^{\uparrow} \mathbb{S} \;\vee\; \exists (x \overset{v}{\dashrightarrow} y') \in E_{dd}.\; y';\, E^* \vdash v :^{\downarrow} \mathbb{S}.$$

Proof. An edge $x \overset{v}{\dashrightarrow} y$ is marked with the meet of the oracle verdict at its source node x and the consensus verdict (i.e., iterated meet) over all sibling sinks $\{y' \mid x \overset{v}{\dashrightarrow} y'\}$, which include y. If this final edge mark is Y, then by definition of the marking lattice, none of the contributing verdicts must be N; in particular, the source verdict at x and the sink verdict at y cannot be N. Also by definition of the marking lattice, a final mark of Y requires that at least one of the contributing verdicts must be Y. Under plausible positives, this means that treating v as a string must be locally consistent at the source node x and/or one or more of the sink nodes y'. □

Remark. The positive witness for the string typing of some variable v may lie at a sibling sink $y' \neq y$, rather than at one or both of the specific endpoints of $x \underset{\mathsf{Y}}{\overset{v}{\dashrightarrow}} y$. This does not imply any type inconsistency at those endpoints, as no oracle has returned N at either x or y and by sound negatives such a verdict would require evidence of syntactic string incompatibility. At the same type, it does not fully rule out the possibility of type inconsistency. However, a Y edge carries strictly more information than a ? edge: both are free of oracle-certified contradiction, but only the former has positive evidence for string typing at some use site of v.

Corollary 2 (Exactness)**.** *Let an oracle be* complete *if it never returns* ?*. Under oracle completeness,* E^*_{Y} *exactly characterizes all string data flow,*

$$x \xrightarrow[\mathsf{Y}]{v} y \in E^* \iff x;\, E^* \vdash v :^{\uparrow} \mathbb{S} \;\wedge\; y;\, E^* \vdash v :^{\downarrow} \mathbb{S}.$$

Proof. Since no oracle verdict is ? under oracle completeness, every consensus verdict is Y or N, hence $E^*_? = \emptyset$. By proposition 5(a), a Y edge implies that neither endpoint oracle returns N and thus both must return Y, which by plausible positives yields string consistency at both endpoints. In the other direction, corollary 1 (Completeness) under $E^*_? = \emptyset$ means that string consistency implies a Y edge. □

The strength of the guarantees provided by the Mark-Edges algorithm depends directly on the strength of the oracle. The table below summarizes the relationship between subsets of the fixpoint set of marked edges E^* and the true sets of string-typed data flow edges $E_{\mathbb{S}}$ and non-string-typed data flow edges $E_{\not\mathbb{S}}$ under varying oracle assumptions.[4]

Oracle Assumptions	E^*_{N}	$E^*_{\mathsf{Y}} \cup E^*_?$	E^*_{Y}	$E^*_?$
Sound Negatives	$\subseteq E_{\not\mathbb{S}}$	$\supseteq E_{\mathbb{S}}$	–	–
+ Plausible Positives	$\subseteq E_{\not\mathbb{S}}$	$\supseteq E_{\mathbb{S}}$	justified	–
+ Oracle Completeness	$= E_{\not\mathbb{S}}$	$= E_{\mathbb{S}}$	$= E_{\mathbb{S}}$	$= \emptyset$

It suffices that oracles emit sound negatives to enable the core guarantees of the analysis. Proposition 4 (Soundness of N) ensures that only genuinely non-string data flow is marked N and by corollary 1 (Completeness) we know that the joint set of all edges not marked N is a safe over-approximation of possible string data flow, regardless of oracle quality. Requiring plausible positives from the oracle does not strengthen these guarantees in the limit, but highly increases the confidence in the results of the algorithm, as per proposition 5 (Justification of Y): Y edges now carry real positive evidence of string typing within the variable's use sites, along with certified absence of evidence for non-string data flow. This allows for practical oracle implementations that handle common syntactic patterns definitively, are allowed to employ sensible heuristics, and may return ? when encountering genuine ambiguity. Under this regime, Y edges can be processed with high confidence of reflecting actual string data flow. Ideally, oracles would be complete and answer every typing question exactly, leaving no ambiguity in the final set of marked edges. The Y edges then constitute an exact characterization of all real string data flow in the program (corollary 2, Exactness). Such an oracle is most easily implemented for statically typed languages, where type inference can fully determine the exact type of any variable at any use site. Alas, Python is a different beast.

[4] Oracle purity and monotonicity are prerequisites for algorithm confluence and the existence of a fixpoint and are implicitly assumed.

1.4 Parser Slices

Having determined string data flow in the program graph, we can now collect program slices that conform to our definition of ad hoc parsing. Each slice begins with the program statement that defines or introduces the string source variable and extends through the string-typed data dependence subgraph, including all program nodes directly connected by string data flow.

Algorithm 4 gives the procedure to collect all parser slices in the program. Each unique slice starts with a source node, defined by having at least one outgoing string data flow and no incoming string flows. The absence of incoming string flow distinguishes origin source nodes, where a fresh string value enters into a parser, from interior parsing nodes that continue another parser's string flow. A source node can have multiple distinct variables with outgoing string flows; each node/variable pair establishes a separate parser slice.

The extraction of a single slice S starts with a source node/variable pair and the slice is grown by forward reachability through data dependence edges. The first step from the slice source node is restricted to string data flow involving the original source variable. All subsequent steps include any string data flow, regardless of variable. This conforms to definition 1 and the notion that any transitively derived string-typed variable continues to be part of the same parsing process. Note that control flow edges are irrelevant at this point; the slice is solely a data dependence subgraph, comprising all and only those program nodes that are immediately grammar-relevant.

Collect-Parser-Slices scans the program graph once in $\mathcal{O}(|V| + |E|)$ to locate all slice source nodes and enumerate outgoing string variable edges. For each slice source node/variable pair, all nodes and edges in the slice's reachable subgraph are visited at most once; thus, each slice is collected linearly in the size of its subgraph, in at most $\mathcal{O}(|E|)$. Note that parser slices may overlap, and each is extracted independently. Since each program node can only define a bounded number of string variables, the number of unique source node/variable pairs is $\mathcal{O}(|V|)$, giving an overall runtime of $\mathcal{O}(|V| \cdot |E|)$.

1.5 Trivial Parsers

All string program slices collected using the techniques described in this chapter are, by our own definition, ad hoc parsers. Any program slice extracted using algorithm 4 fits definition 1 by construction. However, even though they conform to our operational definition of what constitutes an ad hoc parser, some of these program slices would actually not be considered "real" parsers by most people.

There is a distinct class of string programs that common sense dictates are not real parsers. These programs merely output strings, or just concatenate strings, or only perform total functions over strings, not restricting the shape or contents of their inputs in any way. These *trivial* parser programs all have one precise technical thing in common: their input language is universal—they accept any string. Trivial parsers never fail, regardless of the contents of their input strings. Conflating such trivial parsers with genuine (i.e., not universally

Algorithm 4 Collect all parser slices in a program graph.

```
procedure COLLECT-PARSER-SLICES(G)
    𝒜 ← ∅
    for all x ∈ V where ∃y. x -->_Y y ∧ ∄z. z -->_Y x do
        for all v ∈ {v | ∃y. x -v->_Y y} do
            S ← {x}
            W ← {y | x -v->_Y y}
            while W ≠ ∅ do
                choose y ∈ W
                S ← S ∪ {y}
                W ← W \ {y}
                for all y -->_Y z do
                    if z ∉ S then
                        W ← W ∪ {z}
            𝒜 ← 𝒜 ∪ {S}
    return 𝒜
```

accepting) parsers is generally undesirable: an empirical study of ad hoc parsing that included trivial parsers risks being diluted, and expending serious program analysis efforts on such programs seems a pointless waste of resources. Ideally, a procedure for locating ad hoc parsers would filter out any trivial parsers in advance. Unfortunately, parsers can be linguistically trivial while being computationally interesting.

This second class of trivial parsers exposes the unfortunate reality that *in general* a determination of triviality by input language can only be made by actually inferring that input language—a decidedly non-trivial problem! As such, recognizing trivial parsers is actually part of the outcome of our entire endeavor, and not some possible a priori filtering step.

Nevertheless, there are parsers that truly are trivial both linguistically and computationally, and we can give a simple conservative procedure to recognize some (but not all) of these obviously trivial parsers without resorting to full-scale grammar inference. Determining triviality with respect to how a particular string variable is used at a particular program node is necessarily highly programming language dependent and is thus left to a Boolean triviality oracle $\Omega^\top$ whose only requirement is that it be *conservative*: it must never produce false positives; however, it may easily produce false negatives, which is what allows for a simple practical implementation. We say that a parser slice S is *obviously trivial* if all of its nodes are trivial with respect to the string variable read at that node, as determined by the triviality oracle,

$$\forall (x \xrightarrow[\mathsf{Y}]{v} y) \in S.\, \Omega^\top(y, v) \implies S \text{ is trivial.}$$

Table 1 sketches an implementation of $\Omega^\top$ for Python programs. This is an incomplete sketch, but assuming it returns $\bot$ for any cases not shown here, it

Table 1: Sketch of a Python triviality oracle $\Omega^{\top}(x, v)$.

x	$\Omega^{\top}$
`return` e	$\Omega^{\top}(e, v)$
e_1 `=` e_2	$\Omega^{\top}(e_2, v)$
e	$\top$ if $v \notin e$
v	$\top$ if v is only read
`f"...{`e_1`}...{`e_n`}..."`	$\forall e_i.\ \Omega^{\top}(e_i, v)$
e_1`[`e_2`:`e_3`]`	$\Omega^{\top}(e_1, v)$
e_1`[`e_2`]`	$\bot$
`print(`e_1`,...,`e_n`)`	$\forall e_i.\ \Omega^{\top}(e_i, v)$
e_1`.`m`(`e_2`,...,`e_n`)`	$\forall e_i.\ \Omega^{\top}(e_i, v) \wedge m \in \{$`strip`, `upper`, $\dots\}$

is sound (if overly conservative). Of note is that string slicing `s[i:j]` is always a trivial operation in Python, regardless of the indices, which implicitly clamp to the length of the slicee; and attempting to slice an empty string will simply produce another empty string. The same is not true for simple indexing `s[i]`, which can fail with an `IndexError`.