Reuse Bench

Anonymous Author(s)
Affiliation
Address
email

Abstract

Ricci is awesome.

1 Introduction

2 Background

3 Datasets

4 Analysis

5 Conclusion

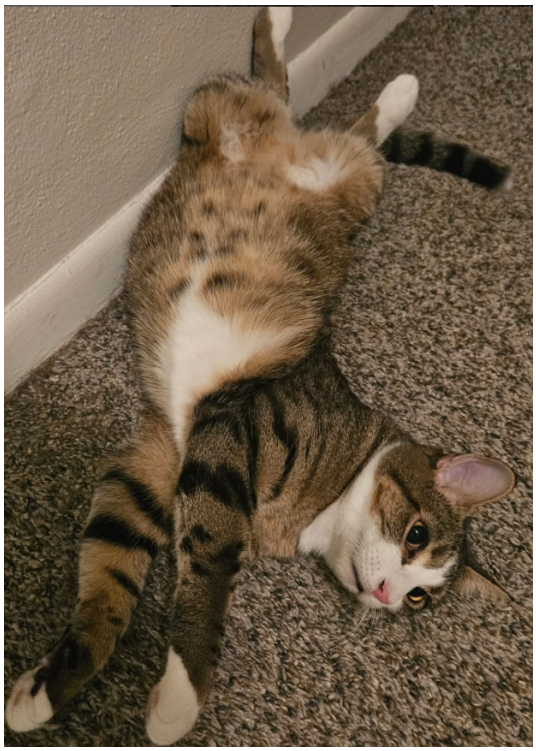

Figure 1: Ricci

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

References

A Dataset Summary

A.1 Identity

Problem Description.

Problem Example.

Solution Example.

Reuse component.

Difficulty control.

Evaluation.

Dataset Info. Dataset size

B CNF Conversion

Problem Description. CNF (Conjunctive Normal Form) conversion tasks typically ask us to start with a propositional logic formula (which may include connectives $\wedge, \vee, \rightarrow, \neg, \dots$) and systematically transform it into a *conjunction of disjunctions of literals*. The solver is required to *apply specific rewriting rules* (e.g., removing implications, moving $\neg$ inward, distributing $\vee$ over $\wedge$) in a **step-by-step** procedure, ensuring that each intermediate formula is logically equivalent to the previous one.

Problem Example. A typical assignment might read:

"Convert $(p \rightarrow q) \wedge \neg(r \vee p)$ into CNF. Show each transformation step: remove $\rightarrow$, push $\neg$, distribute $\vee$, etc."

So, the solver must:

- Rewrite $(p \rightarrow q)$ as $(\neg p \vee q)$,
- Apply De Morgan's laws on $\neg(r \vee p)$ to get $(\neg r \wedge \neg p)$,
- Possibly distribute any $\vee$ over $\wedge$ to finalize CNF form,
- Verify each step is an *equivalence* transformation.

Solution Example. Following the example $(p \rightarrow q) \wedge \neg(r \vee p)$:

1. **Remove implication:** $p \rightarrow q \equiv \neg p \vee q$. So the formula becomes:
$$(\neg p \vee q) \wedge \neg(r \vee p).$$
2. **Push negation inwards:** $\neg(r \vee p)$ by De Morgan's law is $(\neg r \wedge \neg p)$. So now:
$$(\neg p \vee q) \wedge (\neg r \wedge \neg p).$$
3. **Distribute** if needed:
$$(\neg p \vee q) \wedge \neg r \wedge \neg p$$
is already in CNF, since it is a conjunction of three clauses: $(\neg p \vee q)$, $(\neg r)$, and $(\neg p)$.
Hence the final CNF is:
$$(\neg p \vee q) \wedge (\neg r) \wedge (\neg p).$$

Reuse component. Once certain rewriting *identities* are established (like $\neg(p \vee q) \equiv \neg p \wedge \neg q$, or $p \rightarrow q \equiv \neg p \vee q$), one can *cite* them without re-deriving at each step. Similarly, if an earlier problem already removed implications from a sub-formula, that sub-formula can be *embedded* in a new formula—saving repeated transformations. This is akin to citing *known lemmas* in mathematics.

Difficulty control. Ways to scale **difficulty**:

- **Formula complexity**: more nested connectives, multiple implications, extended negations.
- **Extra equivalences needed**: dealing with $\leftrightarrow$, exclusive or, or special connectives (like $\oplus$).
- **Size of final distribution**: distributing $\vee$ over $\wedge$ in large expressions can be many steps.
- **Strict normal forms**: Some tasks might require not just CNF, but also further factorization (like a canonical ordering of literals).

Evaluation. To check correctness:

- **Equivalence check per step**: each rewrite must be a known logical equivalence (implication removal, De Morgan's, double negation, distribution).
- **Final form** is indeed CNF: a conjunction of disjunctions of literals (no leftover operators in the wrong place).
- **No skipped transformations**: if an entire distribution step is large, ensure the solver shows partial expansions or at least references the standard distribution law carefully.

A correct derivation ensures the final formula has the same truth conditions as the original, now in a strict CNF layout.

Dataset Info. Dataset size

C Context-free Grammar

Problem Description. Context-free grammar (CFG) checks often ask the solver to:

- Verify whether a given string can be derived by a certain CFG (i.e. membership checking).
- Prove or disprove that two grammars generate the same language.
- Convert a CFG to a certain normal form (e.g. Chomsky Normal Form), or show step-by-step derivations.

Such tasks require explicit **step-by-step** reasoning, for example enumerating a parse tree or using a parsing algorithm (like CYK, Earley, LL(1), or LR parsing) in a canonical order.

Problem Example. A typical instance might be:

"Given the grammar G with start symbol S and productions:

$$\begin{aligned} S &\to AB \quad | \quad b \\ A &\to aA \quad | \quad \epsilon \\ B &\to b \end{aligned}$$

check whether the string aab is in $L(G)$. Show the stepwise derivation or parse steps."

Alternatively, a question could ask to transform G into Chomsky Normal Form, detailing every elimination of ϵ-rules, unit-rules, etc.

Solution Example. Below is a short membership check using a leftmost derivation for aab:

1. Start with S.
2. Apply $S \to AB$:
$$S \implies AB.$$
3. For A, pick $A \to aA$:
$$AB \implies aA\,B.$$
4. Again for A, pick $A \to aA$:
$$aA\,B \implies aaA\,B.$$
5. Now $A \to \epsilon$:
$$aaA\,B \implies aa\,B.$$

6. For B, pick $B \to b$:

$$aaB \implies aab.$$

7. Hence $S \implies aab$. Thus $aab \in L(G)$.

Reuse component. Many CFG transformations or parsing steps are reusable:

- Once we have removed ϵ-productions or converted a subgrammar to CNF, we can *cite* that partial result in a larger grammar.
- A standard parsing table for a known sub-grammar can be reused in multiple parse checks.
- Known normal forms or well-known derivation patterns (like $S \to aSb \mid \epsilon$ for balanced parentheses) can be invoked repeatedly without re-derivation.

Difficulty control. Ways to scale **difficulty** in CFG tasks:

- **Grammar complexity**: more nonterminals, more productions, nested recursion.
- **String length**: from short strings (quick derivations) to longer ones requiring multiple expansions.
- **Normalization or transformations**: Chomsky Normal Form, Greibach Normal Form, or removing left recursion can be more involved for bigger grammars.
- **Parsing methods**: e.g. CYK vs. LL(1), each step must follow canonical rules for a unique path.

Evaluation. To judge correctness:

- **Verify each production use** in the derivation: every expansion must match a grammar rule exactly.
- **Check final derived string**: it must equal the target string with no extraneous symbols left.
- **Parsing correctness**: if using a parser-based approach (LL(1), LR(1)), confirm each step in the parsing table is consistent with the grammar and the next input symbol.
- **Grammar transformations**: each rewrite (eliminating ϵ, factoring out left recursion) must be verifiable by standard transformations that preserve language equivalence.

Any mismatch in expansions or incorrect unit-rule elimination reveals a flawed step. A correct chain of derivations or transformations ensures soundness of the final conclusion.

Dataset Info. Dataset size

D Lambda Calculus

Problem Description. Lambda calculus reduction tasks typically involve starting with a λ-expression (possibly containing multiple nested abstractions and applications) and *reducing* it via β-reduction rules (and sometimes α-conversion to avoid variable capture). The solver is asked to derive a normal form, or to perform a specific reduction strategy (e.g., *call-by-value*, *call-by-name*, or *leftmost-outermost*). Each intermediate step must be **verifiable** by the standard λ-calculus rules, and a *unique path* is enforced by fixing a particular strategy.

Problem Example. A typical problem might read:

"Reduce the expression $(\lambda x.\, x{+}1)\big((\lambda y.\, y{+}y)\; 3\big)$ using call-by-value. Show each step of the reduction."

Here, the solver must:

- Evaluate the *argument* $(\lambda y.\, y + y)\, 3$ first (call-by-value),
- Perform the actual β-reduction once that argument is evaluated,
- Arrive at a final numeric form (or a Church numeral, depending on how arithmetic is encoded).

Solution Example. Below is a short step-by-step reduction (in call-by-value) for

$$(\lambda x.\, x + 1)\big((\lambda y.\, y + y)\, 3\big).$$

1. *Evaluate the argument* $(\lambda y.\, y + y)\, 3$:

$$(\lambda y.\, y + y)\, 3 \;\longrightarrow\; 3 + 3 \;=\; 6$$

   (assuming an arithmetic interpretation or a further β-reduction if Church numerals are in play).

2. Now substitute the result into $\lambda x.\, x + 1$:

$$(\lambda x.\, x + 1)\, 6 \;\longrightarrow\; 6 + 1 \;=\; 7.$$

3. Final result: 7.

Each $\rightarrow$ step references a valid β-reduction (or arithmetic evaluation if representing numbers as Church numerals), consistent with call-by-value ordering.

Reuse component. Common *mini-reductions* or known sub-expressions are often reused:

- Once you have shown $\lambda y.\, (y + y)$ acting on a Church numeral n reduces to $2n$, you can *cite* that directly in larger expressions.
- Standard combinators (like $\mathbf{I} = \lambda x.\, x$ or $\mathbf{K} = \lambda x.\, \lambda y.\, x$) can be used and referenced as lemmas.

Hence, partial λ-calculus derivations can be integrated into bigger tasks much like citing theorems or subroutines.

Difficulty control. Ways to scale **difficulty**:

- **Expression size:** More nested applications and abstractions, e.g. multiple λ layers.
- **Reduction strategy:** (Call-by-name vs. call-by-value vs. normal-order) influences the path of reduction.
- **Explicit α-conversion**: If variable capture arises, ensuring correct renaming steps can add complexity.
- **Encoding of data structures:** If the expression uses Church numerals, booleans, or pairs, the solver must recall these encodings and systematically reduce them.

Evaluation. To verify a solution:

- **Check each β-reduction**: Ensure correct substitution (no variable capture, no missed occurrences of the replaced variable).
- **Check the sequence of redexes** matches the required strategy (e.g. leftmost-outermost).
- **Confirm final normal form**: No further β-reductions are possible (if a normal form is required).
- If the solution is partial, check that all intermediate steps align with the official reference derivation or a standard λ-calculus engine for correctness.

Dataset Info. Dataset size

E Merge Sort

Problem Description. Merge Sort is a *divide-and-conquer* sorting algorithm that splits an array (or list) into two halves, recursively sorts each half, then *merges* the two sorted halves into one fully sorted array. A typical Merge Sort reasoning task requires demonstrating each recursive call and intermediate merge step in **step-by-step** detail. This ensures the solver clearly shows how subarrays get sorted and combined.

Problem Example. A standard prompt might say:

"Apply Merge Sort to the list $[\,4,\ 1,\ 3,\ 9,\ 7\,]$. Show each major recursive call and how you merge sorted sublists."

The solver is expected to partition the array repeatedly (splitting in half each time), then do step-by-step merges of smaller sorted subarrays until the entire list is sorted.

Solution Example. A brief illustration using the list $A = [4,\ 1,\ 3,\ 9,\ 7]$:

1. **Split** A into two parts:
$$A_{\text{left}} = [4,\ 1,\ 3], \quad A_{\text{right}} = [9,\ 7].$$
2. **Recursively sort** A_{left}:
   - Split $[4, 1, 3]$ into $[4, 1]$ and $[3]$.
   - Sort $[4, 1]$ by splitting into $[4]$ and $[1]$, then merging to get $[1, 4]$.
   - Merge $[1, 4]$ with $[3]$ to get $[1, 3, 4]$.
3. **Recursively sort** $A_{\text{right}} = [9, 7]$:
   - Split into $[9]$ and $[7]$, then merge to get $[7, 9]$.
4. **Final merge**: merge $[1,\ 3,\ 4]$ with $[7,\ 9]$ step by step:
   - Compare 1 and 7: pick 1 first, etc.
   - Eventually obtain the sorted array $[1,\ 3,\ 4,\ 7,\ 9]$.

Reuse component. Once you have demonstrated the subroutine for *merging two sorted lists* in one problem, you can *cite* that process in subsequent tasks. For example, if we already established how to merge $[1, 4]$ and $[3]$, we do not need to re-derive every comparison in a new question about merging $[2, 5]$ and $[1, 9]$. Thus, the merging algorithm acts like a reusable "lemma" or subroutine in future Merge Sort tasks.

Difficulty control. To tune the **difficulty**:

- **List size**: A short list of 5 elements vs. a large list of 20+ elements (which yields more recursive calls).
- **Varied structure**: Repeated elements, negative numbers, or special patterns (e.g., partially sorted lists).
- **Intermediate details required**: Some tasks only require the final sorted array, others demand each *partial* merge step in detail.
- **Tie-breaking strategy**: If you want a unique step-by-step path, specify how to handle ties or even how to split uneven arrays (e.g., if 7 elements, which half gets 3 vs. 4).

Evaluation. To verify correctness:

- Check each *recursive partition* is done as specified (splitting the list correctly).
- Confirm each *merge step* is done in ascending order, comparing the smallest elements first. One can re-check the subarrays are indeed sorted before merging.
- The final output should be the list in sorted order.
- If the task demands full tracing, ensure each sub-step matches a known reference solution (every partial merge or recursion is in the correct sequence).

Dataset Info. Dataset size

F Quantified Boolean Formulas (QBF)

Problem Description. A *Quantified Boolean Formula* (QBF) is an extension of SAT where variables are universally ($\forall$) or existentially ($\exists$) quantified. A typical QBF has the form

$$Q_1 x_1\ Q_2 x_2\ \ldots\ Q_n x_n\ \Phi(x_1, x_2, \ldots, x_n),$$

where each $Q_i \in \{\forall, \exists\}$ and Φ is a propositional formula (often in CNF). The question is whether the quantified formula is *true* or *false*, given the sequence of quantifiers. Solving QBF requires reasoned assignment or proof for each variable in turn, respecting $\exists$ (can choose a witness) and $\forall$ (must hold for all choices).

Problem Example. A small QBF instance could be:

$$\forall x_1\ \exists x_2\ \forall x_3\ \ (\,(x_1 \vee \neg x_2 \vee x_3)\ \wedge\ (\neg x_1 \vee x_2)\,).$$

Here, we must check whether for *all* values of x_1, there is a choice for x_2 such that for *every* choice of x_3, the formula is satisfied. The solver must systematically analyze each quantifier layer.

Solution Example. A step-by-step approach might follow:

1. **Outer quantifier:** $\forall x_1$. Consider $x_1 =$ true and $x_1 =$ false separately.
2. For each x_1 value, check **next quantifier**: $\exists x_2$. Possibly guess or systematically pick an x_2 that satisfies the sub-formula for both possibilities of x_3.
3. Then **last quantifier**: $\forall x_3$. Evaluate the formula for $x_3 =$ true and $x_3 =$ false.
4. If for *every* x_1 one can pick an x_2 that works for *all* x_3, the formula is *true*. Otherwise, it is *false*.

Each sub-step can be turned into a short propositional check (like small SAT tasks), verifying the partial assignments plus universal checks.

Reuse component. Sub-parts of a QBF often reduce to simpler SAT or lower-level QBF sub-tasks. Once a smaller formula or quantifier structure is resolved in one step, it can be *cited* in later steps. For example, if $\exists x_2$ satisfies a sub-formula ψ regardless of x_3, that partial assignment is reusable. Also, known lemmas about certain standard QBF forms (like "$\forall x \exists y : x \neq y$ is always true if the domain is more than 1 bit") can be cited repeatedly.

Difficulty control. To scale **difficulty** in QBF:

- **Number of quantifiers**: short chain (2–3) vs. deeper nesting (5+).
- **Placement of universal vs. existential variables**: more frequent alternation $\forall\exists\forall\exists$ is typically more complex.
- **Size of underlying propositional formula**: simpler 2-3 clauses vs. large CNF.
- **Special structure**: e.g., formula structured in *prenex normal form* with a particular pattern can be easier or harder to solve.

Evaluation. To judge correctness of a QBF solution:

- Verify each quantifier step is checked properly: for $\forall x_i$, the solution must hold for both $x_i =$ true and false (if x_i is Boolean). For $\exists x_i$, there must be *some* setting of x_i making the sub-formula hold for all subsequent universal variables.
- Validate any sub-formula solves: e.g., each partial assignment is consistent with the original clauses or sub-formulas.
- If the solver claims *true*, confirm each universal branch truly has an existential witness. If *false*, confirm no existential assignment can satisfy all universal choices.

Hence, a thorough *step-by-step* argument or a known QBF solver trace can be used to confirm correctness.

Dataset Info. Dataset size

G Regular Expression

Problem Description. Regular expression tasks typically involve:

- *Constructing* a regular expression R that describes a certain language L (e.g., "all strings over $\{a, b\}$ of even length").
- *Transforming* an existing expression into a simplified or normal form (e.g., converting $(ab)^*$ to a certain canonical representation).
- *Proving equivalence* of two given regular expressions (or an expression and a DFA), showing that they define the same language.

Such tasks generally require a **step-by-step** application of known *regex* identities or constructive proofs (union, concatenation, Kleene star, etc.), with each step *verifiable* by recognized rewriting or equivalence checks.

Problem Example. A typical problem might say:

"Show that the language of the regular expression $(ab)^*$ is accepted by the following NFA. Then prove their equivalence or co

Or

"Simplify the regular expression $(a \cup \epsilon)^*(a \cup b)$ to a canonical form, using standard regex identities."

Solution Example. A step-by-step solution might illustrate how to rewrite:

$$(a \cup \epsilon)^*(a \cup b)$$

into an equivalent simpler expression. For instance:

1. Observe $(a \cup \epsilon)^* = (a \cup \epsilon)^*(\epsilon \cup a)$ by commutativity of union. Potentially factor or rewrite in a normal form.
2. Use the identity $R \cup \epsilon \equiv \epsilon \cup R$ which sometimes can be converted to optional forms, or repeated expansions.
3. Carefully apply distribution laws if needed:
$$(XY \cup XZ) = X(Y \cup Z) \quad \text{(for expressions } X, Y, Z\text{)}.$$
4. Eventually arrive at a more canonical or simpler expression such as $(a^*)(a \cup b) = a^*(a \cup b)$, etc.

Each step references a known equivalence rule (distributive law, absorbing law, or expansion of Kleene star).

Reuse component. Sub-results or partial transformations can be reused:

- If you have already shown $(ab)^* \cup (ab)^*a = (ab)^*$ in some prior step, you can *cite* that result in new proofs.
- A known minimal expression for a sub-language can be embedded within a more complicated expression.

Much like "citing a lemma" in mathematics, once a certain regex identity is established, it can be referenced repeatedly.

Difficulty control. Ways to scale **difficulty** in regular expression tasks:

- **Expression size**: More operators, repeated nesting of Kleene stars, or multiple unions.
- **Target normal forms**: Requiring rewriting to a standard normal form (like GNFA-based transformations) can introduce many steps.
- **Equivalence proofs**: Checking that two large expressions define the same language can be more difficult than simpler rewriting tasks.
- **Combined tasks**: Converting a DFA to a regex or vice versa for big automata is more involved.

Evaluation. To evaluate correctness:

- **Step-by-step identity checks**: Verify each rewrite rule is correctly applied (e.g., distribution, union/concatenation expansions).
- **Equivalence checks**: For final expressions, confirm they represent the same language as the initial expression (possibly by known theorems or by constructing NFAs).
- **No skipped transformations**: Each intermediate form must be justified by a standard regex equivalence law.

Because *regex rewriting* is typically *verifiable* by formal transformations or by mechanical checks (like testing string membership in both expressions), any incorrect step stands out upon closer inspection.

Dataset Info. Dataset size

H SAT (Boolean Satisfiability Problem)

Problem Description. SAT (Boolean satisfiability) problems present a Boolean formula in CNF (Conjunctive Normal Form), or some other equivalent form, and ask whether there exists a truth assignment to the variables that makes the entire formula *true*. For example, the formula might be a collection of clauses (disjunctions of literals) that must all be satisfied simultaneously. A step-by-step derivation typically involves applying a systematic decision procedure or resolution method to demonstrate either a **SAT** assignment or a proof of **UNSAT**.

Problem Example. A typical SAT instance might look like:

$$\Phi = (x_1 \vee \neg x_2 \vee x_3) \ \wedge \ (\neg x_1 \vee x_2) \ \wedge \ (x_2 \vee x_3) \ \wedge \ \dots$$

The question is: *Does a truth assignment to* $\{x_1, x_2, x_3, \dots\}$ *exist that satisfies all clauses? If so, provide one such assignment. Otherwise, show it is unsatisfiable.*

Solution Example. A valid step-by-step solution might adopt a **DPLL** (Davis-Putnam-Logemann-Loveland) approach or a **resolution** derivation:

1. **Pick a variable** x_1. Assume $x_1 =$ true.
2. Simplify the formula by removing any clauses containing x_1 (since they are satisfied) and removing $\neg x_1$ from any clauses where it appears.
3. If we reach a conflict (an empty clause), backtrack and assume $x_1 =$ false instead.
4. Continue branching on $x_2, x_3, \dots$ in a similar fashion.
5. If a complete assignment is found that satisfies all clauses, we are done (SAT). If all branches fail, the formula is UNSAT.

Each branching step and clause simplification can be checked mechanically.

Reuse component. When analyzing multiple SAT instances, we can reuse certain sub-formulas or *lemmas* about them. For instance:

- A known tautology or small sub-formula that is always satisfiable (like $(p \vee \neg p)$).
- A repeated *core* formula whose satisfiability is already proven in a separate question.

By referencing these known components, one can shortcut repeated expansions or branching steps.

Difficulty control. To scale the **difficulty** of SAT problems:

- **Number of variables**: Small (3–5) vs. large (20+).
- **Clause density or structure**: Some formulas have many short clauses, others have fewer but longer clauses.
- **Special forms**: 2-SAT (polynomial-time solvable) vs. general 3-SAT.
- **Branching complexity**: Adding symmetrical or *trick* clauses to force more branching in a DPLL approach.

Evaluation. To evaluate correctness:

- **Check each branching step**: ensuring any partial assignment is consistent, and each simplified formula is correct.
- **Verify the final assignment** (for SAT): plugging it into the original formula to confirm every clause is satisfied.
- **Resolution-based approach**: verify that each derived clause follows from previous ones, culminating in an empty clause (if unsatisfiable).

If any step misapplies the branching, resolution, or simplification rules, the solver's derivation is invalid. A valid chain, in contrast, demonstrates soundness and completeness for that specific instance.

Dataset Info. Dataset size

I Symbolic Differentiation

Problem Description. Symbolic differentiation tasks provide a symbolic function $f(x)$ (possibly involving polynomials, exponentials, products, quotients, or trigonometric functions) and ask the solver to *differentiate* it step by step. The goal is to produce a fully expanded or simplified derivative, following a specific rule sequence (e.g., product rule, chain rule) at each stage. This enforces a **multi-step** solution, and each intermediate step is **verifiable** by checking standard differentiation rules.

Problem Example. A typical question might read:

"Differentiate $f(x) = x^2 \cdot e^x$ with respect to x. Show each rule used (product rule, chain rule) at each step."

In this scenario, we identify the relevant differentiation rule (the product rule), apply it carefully, and optionally simplify the resulting expression.

Solution Example. A valid step-by-step solution for $f(x) = x^2 e^x$:

1. Let $u = x^2$ and $v = e^x$.
2. By the product rule: $\frac{d}{dx}[u \cdot v] = u' \cdot v + u \cdot v'$.
3. Compute $u' = \frac{d}{dx}(x^2) = 2x$.
4. Compute $v' = \frac{d}{dx}(e^x) = e^x$.
5. Therefore,
$$f'(x) = \big(2x\big)e^x \ + \ x^2\big(e^x\big) \ = \ e^x(2x + x^2).$$

One can verify each sub-step by checking the correctness of product rule application and basic derivative formulas.

Reuse component. Often, the same *mini-derivatives* appear across multiple questions. For instance, once we have established that $\frac{d}{dx}(x^n) = nx^{n-1}$ or that $\frac{d}{dx}(e^x) = e^x$, we can cite those directly in subsequent problems. Similarly, if we already derived that $\frac{d}{dx}(\sin(ax)) = a\cos(ax)$ for a constant a, we can reuse it for many tasks. This is analogous to *citing known sub-results* or lemmas, eliminating repeated derivations.

Difficulty control. To scale the **difficulty** of symbolic differentiation problems:

- **Number of factors**: Larger products (e.g., three- or four-term products) require multiple product rule expansions.
- **Composition depth**: Nested chain rules (e.g., $\sin\big(e^{x^2}\big)$).
- **Mixed functions**: Polynomials, exponentials, trigonometric functions, logarithms, or piecewise definitions.
- **Simplification complexity**: Requiring the final answer in a specific form (fully expanded vs. factored) can add steps.

Evaluation. Each differentiation step is checked against a *known derivative rule* (e.g., product rule, chain rule, quotient rule, basic function derivatives). A solver's correctness can be assessed by:

- Verifying each intermediate expression matches the declared rule application (e.g., does $u' \cdot v + u \cdot v'$ truly expand to the solver's next line?).
- Optionally, performing a final check: re-differentiate the solver's result or substitute numeric values to see if it matches the true derivative at sample points.

Because differentiation is well-defined in terms of standard rules, any incorrect algebra or omitted factor stands out quickly during verification.

Dataset Info. Dataset size

J Trigonometric Identity Transform

Problem Description. Trigonometric transformation problems present an expression involving trigonometric functions (e.g., $\sin$, $\cos$, $\tan$, $\cot$, etc.), possibly combined with algebraic factors or exponentials, and require rewriting it into a specified *target form* (such as a single trig function, or only $\sin$ and $\cos$, or a sum-product form). The solver must apply **canonical trigonometric identities** (angle-sum formulas, double-angle formulas, product-to-sum identities, etc.) in a specific stepwise manner, ensuring each transformation is *verifiable*.

Problem Example. A typical question might say:

"Rewrite $\tan(x)\cdot\cos(2x)$ as an expression purely in terms of $\sin(x)$ and $\cos(x)$. Show each trig identity used."

For instance, one might expand $\cos(2x)$ with $\cos^2(x) - \sin^2(x)$, then rewrite $\tan(x)$ as $\sin(x)/\cos(x)$, and so on, step by step.

Solution Example. Here is a brief sketch of the solution steps for $f(x) = \tan(x) \cdot \cos(2x)$:

1. **Rewrite** $\tan(x)$:
$$\tan(x) = \frac{\sin(x)}{\cos(x)}.$$
2. **Use a double-angle identity** on $\cos(2x)$:
$$\cos(2x) = \cos^2(x) \; - \; \sin^2(x).$$
3. **Combine them**:
$$f(x) = \frac{\sin(x)}{\cos(x)} \; \cdot \; \big[\cos^2(x) - \sin^2(x)\big].$$
4. **Distribute** if desired:
$$f(x) = \sin(x) \; \cos(x) - \frac{\sin^3(x)}{\cos(x)}.$$
5. **Further simplification** may depend on the instructions (e.g., factor out $\sin(x)$, or keep it in that form). Each step references a known identity (double-angle, $\tan$ definition).

Reuse component. Once certain identities are proven or used (e.g., $\sin^2(x) + \cos^2(x) = 1$, $\sin(2x) = 2\sin(x)\cos(x)$, or product-to-sum formulas), the solver can *cite* these sub-results repeatedly. Similarly, any partial expression that has been successfully transformed in a prior question (e.g., rewriting $\tan(x)\cos(2x)$) can be plugged in as a "lemma" in a larger or more complicated expression.

Difficulty control. Ways to scale **difficulty** in trigonometric transformations:

- **More layers of angles**: $\sin(2x)$ vs. $\sin(3x + \pi/4)$, multiple nested sums or differences.
- **Complex compositions**: e.g., rewriting $\sin(\cos(x))$ is typically out of scope, but adding exponentials, $\tan^2(x)$, or $\sec(x)$ can make transformations more involved.
- **Number of steps**: A short expression with one or two identities vs. a longer expression requiring four or five.
- **Specific final form**: Requiring a sum/product form, or purely $\sin$-$\cos$ form, or factoring out common terms can add complexity.

Evaluation. Each step is evaluated by checking:

- **Identity correctness**: Did the solver apply the correct double-angle formula, sum-to-product identity, etc.?
- **Algebraic manipulation**: Are multiplications, divisions, and factorizations accurate?
- **Final form**: Does the final expression match the requested format (e.g., no leftover $\tan$, no powers of $\sin$, etc., if that was specified)?

Because trigonometric identities are standard, a mismatch at any step indicates an incorrect application or algebra slip. A correct chain of transformations is easily *verifiable* by substituting numeric values to confirm equivalence or by standard identity checks.

Dataset Info.

K Turing Machine Simulation

Problem Description. Turing Machine (TM) simulation problems present a deterministic TM description (states, tape alphabet, transition function, etc.) along with an initial tape configuration. The solver must step through each configuration of the TM until it halts, then report whether it *accepts* or *rejects* (or possibly provide the final tape contents). Such tasks require **step-by-step** verification of transitions and enforce a **unique reasoning path** because the machine is deterministic.

Problem Example. Consider a TM M with:

- States: $\{q_0, q_1, q_{\text{accept}}\}$, where q_0 is the start state and q_{accept} is the accepting state.
- Tape alphabet: $\{\sqcup, 0, 1\}$, where $\sqcup$ represents a blank symbol.
- Input alphabet: $\{0, 1\}$.
- Transition function (partial example):

$$\delta(q_0, 0) = (q_1, 1, R), \quad \delta(q_0, 1) = (q_{\text{accept}}, 1, R), \quad \dots$$

If the initial tape is $\sqcup\sqcup 0\,1\,0\sqcup\sqcup\dots$ with the head on the first 0, the question is: *Does the machine accept or reject this input?*

To solve, one must enumerate each configuration: (current state, tape contents, head position) and apply δ until reaching either q_{accept} (accept) or a non-existent transition (reject).

Solution Example. A valid step-by-step solution might look like this (abbreviated):

1. **Configuration 1:** $(q_0,\ \_\_\underline{0}10\ \_\_\dots)$.
   $\delta(q_0, 0) = (q_1, 1, R)$.
   $\Rightarrow$ **Configuration 2:** $(q_1,\ \_\_1\underline{1}0\ \_\_\dots)$.
2. **Configuration 2:** $(q_1,\ \_\_1\underline{1}0\ \_\_\dots)$.
   Suppose $\delta(q_1, 1) = (q_{\text{accept}}, 1, R)$.
   $\Rightarrow$ **Configuration 3:** $(q_{\text{accept}},\ \_\_11\underline{0}\ \_\_\dots)$.
3. Since the machine is now in q_{accept}, it halts and **accepts** the input.

Each transition is mechanically checkable by looking up the relevant $\delta(q, \text{symbol})$.

Reuse component. A sub-routine or partial sequence of moves can be reused across problems. For instance, if a portion of the TM "rewinds the tape to the left," and we have already traced that behavior in one question, we can cite that known sub-computation in another task where the same sequence of transitions occurs. This is analogous to "citing theorems" in proofs. Once established, the user can simply reference the known step-by-step transitions (e.g., "Subroutine $\mathcal{R}$ that scans right until a blank symbol is found") without re-deriving them.

Difficulty control. To scale **difficulty** in Turing Machine simulation tasks:

- **Tape length.** A short tape with 2–3 symbols vs. a longer tape with 10+ symbols.
- **Number/complexity of states.** A TM with 2–3 states vs. 5–10 states (or more intricate transitions).
- **Special sub-procedures.** Inclusion of loops or sub-routines that require multiple passes over the tape.
- **Time complexity.** TMs that run for many steps vs. those that halt quickly.

Larger tapes, more states, or complicated transition functions yield longer simulation traces and more verification steps.

Evaluation. To **evaluate** a solver's correctness:

- Check whether each transition from configuration i to $i + 1$ matches the TM's δ.
- Verify the final state (accept/reject) matches the official outcome.
- Optionally, compare the entire configuration trace to a "ground truth" reference to ensure no step is skipped or incorrectly applied.

Because the TM is deterministic, if the solver's trace diverges from the official reference at any point, the solver's answer is incorrect.

Dataset Info.