

THE RESEARCH UNIVERSITY PRESS

International
Computational Law Journal

VOLUME 2 / ISSUE 1 / CURATED SOURCE EDITION

AI-Driven Statutory Reasoning via Software Engineering Methods

Rohan Padhye

MIT Computational Law Report · Originally published 2024-09-10

Independent reader edition under CC BY 4.0. This attribution cover was added; original source PDF pages follow unchanged. Original authorship and source identity are retained. MIT, Stanford University, Stanford Law School and CodeX have not sponsored or endorsed this republication.

Source, license and version evidence accompany the online article.

MIT Computational Law Report

AI-Driven Statutory Reasoning via Software Engineering Methods

Rohan Padye

Published on: Sep 10, 2024

URL: https://law.mit.edu/pub/aidrivenstatutoryreasoning

License: Creative Commons Attribution 4.0 International License (CC-BY 4.0)

1. Introduction

The recent proliferation of generative artificial intelligence (GenAI) technologies such as pre-trained large language models (LLMs) has opened up new frontiers in computational law. An exciting area of development is the use of AI to automate the rule-based reasoning inherent in statutory and contract law. While this form of reasoning has long been studied using classical techniques of natural language processing (NLP) and formal logic, recent solutions increasingly make use of LLMs [11]; though they are far from perfect [4].

The advent of GenAI has made it possible to treat many of these natural language documents essentially as programs that compute a result given some set of facts. As such, it should be possible to understand, debug, maintain, evolve, and fix these documents using well-studied techniques from the field of *software engineering*. This article introduces several concepts of automated *software testing* and *program analysis* that could potentially be useful in computational law when applied to AI-driven analysis of statutes and contracts.

2. Background

Statutes are legal rules written by legislative or regulatory bodies (e.g., the *Internal Revenue Code* (IRC) deals with U.S. taxation). In prior research, the term *statutory reasoning* [13, 15] has been used to mean the automated application of such rules to a particular situation (e.g., whether an individual can claim a certain tax deduction in some year given their personal circumstances). Statutory reasoning is usually concerned with the identification of relevant rules and evaluating their composition.

2.1 Statutory Reasoning as *Interpretation*

Consider the task of reasoning about specific parts of the IRC that deal with standard deductions and computation of taxable income. The relevant statutes are provided in Appendix A.

Example 1. *Alice and Bob are married and filing taxes jointly for the year 2018. Alice was born on 1/1/1981 and Bob was born on 12/30/1975. In 2018, Alice and Bob's adjusted gross income was $216,350. Alice and Bob do not itemize their deductions and do not qualify for any deductions other than the standard deduction. What is their taxable income for 2018?*

Solution Approach. Although a seemingly simple question, correctly answering it requires applying multiple computational rules from the IRC (ref. Appendix A): §63(b) and §63(c)(1) to identify the calculation of standard deduction and taxable income; §63(c)(2)(A)(i) and §63(c)(2)(C) to determine the formula for basic standard deduction applicable to joint returns; §63(c)(7)(A)(ii) which overrides a value in the preceding formula; as well as realizing that several other rules such as §63(c)(3) and §63(f)(1) do not apply because Alice and Bob are not 65 years old in 2018, and that inflation adjustments decreed in §63(c)(4) and §63(c)(7)(B) do not apply in 2018. The final answer is $216,350-(2×$12,000)=$192,350.

The reasoning process thus includes resolution of cross-references, handling of exceptions that may in- validate seemingly applicable rules, and performing numerical computations; crucially, however, this process assumes that all the facts (as they pertain to the rule-based reasoning) are clearly known and so it does not deal with the subjective interpretation of potentially ambiguous terms [10]. As such, statutory reasoning technology can also be expected to apply to contract law in limited contexts (e.g., to what extent does an insurance policy cover a particular incident?).

A modern LLM like Anthropic's *Claude 3 Opus* can usually answer questions like Example 1 correctly when provided the relevant statutes in the prompt, and when instructed to explain its chain of thought [14]. Prior research has shown that on statutory reasoning datasets that include a subset of the IRC [13] and the Japanese Civil Code [11], LLM-based approaches have achieved accuracy upwards of 75–80% on tasks that do not involve numerical calculations.

2.2 Statutory Reasoning as *Analysis*

Another interesting though relatively less studied problem is the analysis of the statutes and/or contracts themselves, from the point of view of the drafters or other stakeholders interested in understanding how the rules apply in a variety of hypothetical situations. Such an analysis can be used to identify problematic edge cases in the legal text, or to ensure that certain consistency properties are met. For example, the *Shelter Check* [3] project aims to proactively identify tax loopholes in the IRC before they are exploited. A similar analysis in contract law might help identify if certain clauses were inconsistent or irrelevant when combined with other parts of the contract. GenAI is a promising technology for addressing such analysis.

With reference to the running example, we could potentially ask an LLM to determine if *taxable income, as defined by 26 USC §63(b)* can be negative and to demonstrate this with an example scenario. Claude 3 Opus generates the following example:

Example 2. *Charlie and Diana are married and filing taxes jointly for the year 2018. Charlie was born on 5/15/1952 and Diana was born on 8/22/1953. In 2018, Charlie and Diana's adjusted gross income was $20,000. Charlie and Diana do not itemize their deductions and do not qualify for any deductions other than the standard deduction. What is their taxable income for 2018?*

Author's analysis. It appears that a mechanical application of §63(b) when adjusted gross income is less than the standard deduction can lead to a negative result for the computation of *taxable income*. The author of this article (who is not a lawyer) is not aware of any specific law that prevents this interpretation. Interestingly, IRS Publication 525 (2023) [20] references the term "negative taxable income" to have this meaning in the context of recovering prior year adjustments to gross income.

Overall, LLMs appear to be useful for synthesizing examples to reason about hypothetical situations.

3. Software Engineering Methods for Statutory Reasoning

Our main argument is that the automated analysis of rule-based reasoning in statutes and contracts can now be viewed from the lens of software engineering. The legal text itself is a program that computes, given an *input* —i.e., a set of facts about a person or circumstance—some objective result such as whether a tax deduction is available (and how much) or whether an insurance policy covers an incident (and to what extent). Of course, the overall concept of treating rule-based legal documents as programs is not new: various efforts have been made towards formally defining statutes [18] or developing computable contracts [12]. This article proposes to treat natural language legal documents directly *as if they were programs* using LLMs, such that we can apply principled software engineering techniques in their analysis.

3.1 Example Generation via Coverage and Mutation Analysis

Example generation: Consider the task of generating one-paragraph example scenarios (as in Example 1) such that certain rules in the statute apply in specific ways. This could be useful for documentation or education —for instance, the IRS embeds hundreds of examples across its various publications and form instructions in order to demonstrate how federal tax law applies across various circumstances. Additionally, high quality examples can also aid in training language models to perform more accurate statutory reasoning—researchers have found that augmenting training data with additional examples improves accuracy on some tasks in the COLIEE dataset [21].

Software test generation: Given our view of treating statutory reasoning as execution of natural- language software, let's see how *example generation* is performed in software engineering. The analogous task in this field is that of *constructing a high-quality test suite*. Software tests are useful in many ways including validation (i.e., checking whether a program behaves correctly), specification (i.e., writing a test before implementing functionality, as in *test-driven development* [2]), and documentation (i.e., the test code demonstrates the behavior of a certain application programming interface (API)).

In software testing, the quality of a test suite is usually measured using criteria that track the diversity of program behaviors captured across all tests. The most widely used example is that of *code coverage*, which measures the fraction of source code (often counted at the level of *lines* of code) exercised across all tests. Consider the following Python function which computes the max of two integers:

```
def max ( a, b):
  if a > b: return a
  else : return b
```

To get full code coverage on this program, one may write two test cases, such that both sides of the conditional branch are exercised:

```
assert max (1 , 2) == 2  # Exercises lines  1  and  2
 assert max (2 , 1) == 2  # Exercises lines  1  and  3
```

Coverage analysis for statutory reasoning: Using LLMs, we can use the same principles as in software testing to measure the coverage of available examples and to generate more examples by targeting specific coverage criteria. For example, Claude 3 Opus correctly produces a list of rules whose conditions were satisfied in this example: "§63(b), §63(c)(1), §63(c)(2), §63(c)(7)". Going further, we can also ask the LLM to synthesize examples such that a particular rule is exercised in its coverage. For instance, asking Claude for an example such that §63(f)(1)(A) applies but §63(f)(1)(B) does not, produces:

Example 3. *Charlie and Diane are married and filing taxes jointly for the year 2018. Charlie was born on 3/15/1985 and Diane was born on 8/22/1953. In 2018, Charlie and Diane's adjusted gross income was $185,000. Charlie and Diane do not itemize their deductions and do not qualify for any deductions other than the standard deduction.*

Explanation (as per Claude). "In this scenario, Diane has attained age 65 before the close of the 2018 tax year, so §63(f)(1)(B) applies, entitling Charlie to an additional standard deduction amount for his spouse. However, Charlie himself has not attained age 65 before the close of the 2018 tax year, so §63(f)(1)(A) does not apply".

Software mutation testing: In software testing, a criticism of the *code coverage* measurement of test-suite quality is that it only cares about whether certain lines of source code were *exercised*, but not the meaning of those lines themselves. This can result in surprising test outcomes. For example, consider what would happen if in the definition of the `max` function above, a programmer would make a typographical mistake at Line 2 and write `return 2` instead of `return a`. Well, it turns out the two tests listed earlier asserting that `max(1, 2) == 2` and that `max(2, 1) == 2` continue to succeed, and show 100% code coverage. To identify the programmer's error, we need an additional test that would fail when the typo exists but succeed when the program is correct:

```
assert max (3 , 1) == 3  # Exercises lines  1  and  2  again
```

This approach of reasoning about test-suite quality as whether it would catch hypothetical programmer errors is called *mutation testing* [9]. The main idea is to introduce small changes in the source code—such as replacing `return a` with `return 2`, called a *mutation*—and then writing new tests that would distinguish the behavior of the original program from that of the mutation. This process can be repeated for dozens or hundreds of mutants before a test suite is deemed adequate—research has shown that test suites produced in this way can actually help catch real software bugs more reliably than a test suite that is only optimized for coverage [19].

Statutory example generation using mutation testing: We can apply the same principles as in software mutation testing to generate examples that not only exercise certain statutory rules, but also distinguish them from other incorrect rules. For example, we can ask an LLM to assume that §63(f)(1)(B) is modified to remove the second clause, as follows:

> (B) for the spouse of the taxpayer if the spouse has attained age 65 before the close of the taxable year ~~and an additional exemption is allowable to the taxpayer for such spouse under section 151(b)~~.

Then, we ask the model to synthesize an example such that §63(f)(1)(B) would apply with the modified rule but it would not apply with the original rule. Claude 3 Opus responds to this challenge with the following example:

Example 4. *Ellie and Frank are married, but they are filing their taxes separately for the year 2018. Ellie was born on 5/10/1980 and Frank was born on 2/28/1952. In 2018, Ellie's adjusted gross income was $75,000 and Frank's adjusted gross income was $50,000. Neither Ellie nor Frank itemize their deductions and they do not qualify for any deductions other than the standard deduction.*

Explanation (as per Claude). "The modified rule [...] would apply to Frank because he has attained age 65 before the close of the 2018 tax year. This means Frank would be entitled to an additional standard deduction amount for being aged 65 or older, even though he and Ellie are filing separately. However, under the original rule, §63(f)(1)(B) would not apply to Frank because it requires not only that the spouse has attained age 65, but also that "an additional exemption is allowable to the taxpayer for such spouse under section §151(b)." [...] In this scenario, Frank does have gross income, so even though he is over 65, the original §63(f)(1)(B) would not apply to him because Ellie cannot claim an additional exemption for him under §151(b)."

Mutation testing thus helps with generating examples that not only exercise specific parts of relevant statutes but also provide a basis for reasoning about why certain rules are written in a certain way. The mutation testing process can potentially benefit drafters in thinking through various edge cases.

3.2 Interpreting Substitutions via Inlining

Inlining subroutines in software: Software frequently makes use of *subroutines* for re-using common pieces of functionality, such as the `max` function defined in the previous section. While this saves space, subroutine invocations are associated with a run-time performance overhead. Consider a program expression such as `w = max(x, x+1)`. Executing this code first requires allocating memory for the parameters of the max function, namely variables `a` and `b`. These variables are then assigned the values `x` and the result of executing `x+1` respectively. The conditional logic inside the definition of the max function is then executed, and the returned value is assigned to `w`.

In many situations, a programming language compiler, which converts source code to machine code for execution, performs an optimization called *inlining* [8]. The basic idea is to replace an expression containing a function invocation with the code that implements this function, while performing the necessary substitutions of variable names. Often, this allows the compiler to perform further optimizations. In the above example, the code `w = max(x, x+1)` can be optimized by first *inlining* the body of the max function as follows:

```
if x > x +1: w = x else : w = x + 1
```

At this point, the compiler realizes that `x > x+1` can never be true, and so the whole code can be further optimized to just a single line `w = x+1`. Executing this final form of the code is much faster than performing the subroutine.

Substitutions in AI-driven statutory reasoning: One of the challenges with using LLMs for automated statutory reasoning is the cross-application of other sections with substituted values. For example, consider the task of determining the standard deduction for taxable year 2025, which requires calculating cost-of-living adjustments using hypothetical inflation data provided in Appendix B.[1]

Example 5. *Alice and Bob are married and filing taxes jointly for the year 2025. What is the basic standard deduction available to them?*

Solving this task correctly requires several considerations:

1. For joint returns, §63(c)(2)(A)(i) declares the basic standard deduction to be 200% of the value in §63(c)(2)(C), which is stated to be $3,000.
2. §63(c)(4) applies "adjustments for inflation" to the value in §63(c)(2)(C)), but later §63(c)(7)(B)(i) overrides this substitution by declaring that "Paragraph (4) shall not apply" for 2025.
3. For 2025, §63(c)(7)(A)(ii) replaces the value "$3,000" in §63(c)(2)(C)) with the value "$12,000".
4. As per §63(c)(7)(B)(ii), the value "$12,000" in §63(c)(2)(C)) is further increased using a cost-of-living adjustment as per §1(f)(3), but applying the latter rule itself with a substitution of "2017" for "2016".
5. §1(f)(3)(A) provides a formula for computing a cost-of-living adjustment using the C-CPI-U value for the preceeding calendar year (i.e., 2024) and the CPI for calendar year 2016 multiplied by the ratio of the C-CPI-U for 2016 to the CPI for 2016 as in §1(f)(3)(B). However, §1(f)(3)(C) comes into play, changing the calculation because we are using "2017" instead of "2016", as determined in the previous step. Now, it appears we must only use the C-CPI-U values of 2025 and 2016; no multiplication is needed.
6. Plugging in the values from Appendix B, the adjustment is calculated to be 24.81%.
7. Applying the adjustment and the rounding specified in §63(c)(7)(B)(ii), the value "$12,000" from §63(c)(2)(C)) is therefore determined to change to "$15,000".
8. Finally, stepping back to §63(c)(2)(A)(i), the basic standard deduction for a joint return is therefore 200% of this adjusted value. So, the result is $30,000.

In our experiments, modern LLMs like Claude 3 Opus or GPT-4 are unable to solve this task correctly even with few-shot in-context learning [5] and using *chain-of-thought* reasoning [14]. There are simply too many *subroutines*, and the LLM's reasoning inevitably falters by using a wrong value or applying it in the wrong place. The end result is almost always incorrect.

Inlining substitutions in statutory reasoning: Taking inspiration from *inlining* in software optimiza- tion, we devised a prompting strategy where we asked the LLM not to calculate the final answer, but just perform one

step of the substitution and reproduce a modified version of the statute verbatim. For example, after step 2 above, the LLM will return the text in Appendix A except with §63(c)(4) completely removed. After applying step 3 above, the modified version of the statute will say the following in place of §63(c)(2)(C) (emphasis added):

2. Basic standard deduction
   ...
   (B) **$18,000** in the case of a head of household (as defined in section 2(b)), or
   (C) **$12,000** in any other case.

Continuing in this way, applying step 5 above changes §1(f)(3)(A) to say (emphasis added):

3. Cost−of−living adjustment
   For purposes of this subsection—
   (A) In general
   The cost−of−living adjustment for any calendar year is the percentage (if any) by which— (i) the C−CPI−U for the preceding calendar year, exceeds
   (ii) **the C-CPI-U for calendar year 2017**.

and so on. Using a modified version of the statute with *inlined* substitutions in each subsequent step allows the LLM to slow down its chain of thought. With this process, Claude 3 Opus is able to correctly solve Example 5.

3.3 Metamorphic Property-Based Testing

Property-based testing in software: Traditional software testing validates the behavior of some piece of code on one or more specified inputs (e.g., asserting what `max(1, 2)` should produce). But in many cases, we are interested in reasoning about generic properties of the code that must hold regardless of what input it is provided. For example, we might want to ensure that that the function `max(a, b)` produces a result that is never less than `a` or `b`, or that the result is exactly equal to either `a` or `b`. These are properties that should hold true regardless of the values of `a` and `b`.

Property-based testing, originally invented for testing functions in the Haskell programming language [7], is a technique for writing generic tests that express what properties a function should satisfy given *any* imaginable input. We can express the properties of the `max(a, b)` function using tests that are *parameterized* on `a` and `b`. In Python, using a library for property-based testing called *Hypothesis* [17], this is:

```
@ given ( int () , int ()) # Given any two integers
def test_max_properties ( a, b): c = max ( a, b)
  assert c >= a and c >= b # These properties ...
  assert c == a or c == b # ... should hold
```

The key idea of property-based testing is to check such generic properties using auto-generated inputs values. For testing the properties of `max`, a property-based testing tool will randomly sample thousands or even

millions of values for `a` and `b` and run the above function to check whether the assertions hold. If any assertion fails on any input, we know that `max` is implemented incorrectly. The test code is also self-documenting: it expresses the expected behavior of the max function without knowing anything about how it is implemented.

Metamorphic testing in software: A special class of property-based testing checks properties that should hold across multiple invocations of some program function when the inputs are changed in a specified manner—this is called *metamorphic testing* [6]. For example, we can validate that the `max(a, b)` function is *monotonic*; that is, increasing the value of either of its inputs a or b should never produce a result that is smaller than the original invocation. Expressed as a property-based test in Python:

```
@ given ( int () , int ()) # Given any two integers
def test_max_monotonic ( a, b):
  assert max ( a + 1 , b) >= max ( a, b) # These properties ...
  assert max ( a, b + 1) >= max ( a, b) # ... should hold
```

Again, these properties are simply checked by running the above function (`text max monotonic`) by randomly generated values for `a` and `b` until a violation is found or until is one is satisfied that the property appears to hold.

Applications to statutory analysis: The analysis and validation of legal statutes can potentially be automated using the key principles of metamorphic property-based testing and combining these with AI- driven techniques of interpretation. For example, consider §63(c)(7)(B), which provides adjustments for inflation to the basic standard deduction applicable in tax years 2018–2025. One may want to analyze the code to ask: *Can the basic standard deduction available for single filers ever decrease year-over-year in this period?* Perhaps a legislator or other stakeholder is interested in understanding the effects of year-over- year deflation on the tax code. Answering this question involves not just interpreting the statutes but also reasoning about how they would apply in hypothetical edge-case scenarios.

Unfortunately, asking an LLM to analyze general properties of legal statutes directly does not always produce accurate results. When directly prompted with the aforementioned question, Claude 3 Opus de- termines that "for taxable years beginning after 2018, the $18,000 and $12,000 amounts are increased by a cost-of-living adjustment determined under §1(f)(3) [...] Since the adjustment can only increase the basic standard deduction amounts [...] the basic standard deduction *cannot decrease* during the 2018-2025 period" (emphasis added). However, it turns out that this is not fully correct.

We can help Claude by expressing our question as a metamorphic property-based test to verify the *monotonicity* of the inflation adjustments—Given any two years $\boldsymbol{X}$ and $\boldsymbol{Y}$, such that $\mathbf{2018} \leq \boldsymbol{X} < \boldsymbol{Y} \leq \mathbf{2025}$, and hypothetical C-CPI-U values for the correspondingly preceding calendar years $\boldsymbol{I_x}$ and $\boldsymbol{I_y}$ in the numeric range [100, 200], calculate $\boldsymbol{D(X, I_x)}$ and $\boldsymbol{D(Y, I_y)}$ respectively, where function $\boldsymbol{D}$ is the computation of basic standard deduction for single filers in a given year assuming a given C-CPI-U value for the preceding calendar year; then check

whether $\boldsymbol{D(X, I_X) \leq D(Y, I_Y)}$ is true. Now do this for random values of $\boldsymbol{X, Y, I_X, I_Y}$ in a loop—in each iteration, use the LLM to calculate the inflation adjustment using the inlining technique described in the previous section.

It does not take many iterations to discover that it is indeed possible to violate the stated property. Although the adjustments defined in §63(c)(7)(B) and §1(f)(3) do indeed only allow for *increases* to the basic standard deduction value listed in §63(c)(2)(C) (which is "$12,000" in the period 2018–2025), the amount by which this value is increased can seemingly decrease year-over-year if there is deflation during those subsequent years. Property-based testing can reveal a situation where the monotonicity property is violated: $\boldsymbol{D}$(2024, 168.1)=$14,600 and $\boldsymbol{D}$(2025, 167.0)=$14,500; that is, if there was a hypothetical 0.65% deflation in 2024, then the basic standard deduction for single filers in tax year 2025 as computed by §63(c)(2)(C) and §1(f)(3) appears to be $100 lower than that computed for the previous year.

3.4 Other Applications of Software Engineering Methods

Once we start treating AI-driven statutory reasoning as a software program analysis problem, we can take inspiration from a host of other techniques studied in the field of software engineering.

- *Delta debugging* [22] is a technique for isolating the root cause of a program failure in terms of input data (e.g., think of a large file that causes a word processor to crash). The key idea is to repeatedly slice and dice the input until we obtain the smallest possible chunk that also reproduces the failure.
  - Can we perhaps use delta debugging for AI-driven statutory reasoning in order to generate minimal examples such that a particular statutory rule applies?
- *Change impact analysis* [1] is a technique for identifying which modules in a software get affected by a small patch applied to the program. This can be used for optimizations such as choosing a small subset of test cases to run for quick validation purposes instead of spending a lot of time executing the whole test suite. The key idea to performing change impact analysis is to first build a directed graph of various program modules and their inter-dependencies using both static source code information and dynamic hints from previous test executions. The impact of changes can then be identified by following paths through such a graph.
  - Can we perhaps use this technique to identify how proposed changes to statutes (e.g., a bill introduced in the legislature) or contracts affect various other provisions and how they may be applied?
- *Automatic program repair* (APR) [16] is a technique for fixing bugs in software. Given a program and a test input on which it produces the wrong output, the key idea is to mechanically search over a space of small changes to the program such that the test execution results in the correct expected value.
  - Can we perhaps use APR techniques with LLMs to prototype changes to certain rules or clauses in order to achieve a desired interpretation for some given circumstance?

4. Conclusion

Generative AI has unlocked new ways of automating statutory reasoning and contract analysis. The intri- cacies and peculiarities of legalese make it much more suitable to treat statutes and contracts as computer programs rather than as natural language databases of information. Consequently, we argue that this prob- lem can benefit from the principled application of techniques well-studied in the field of software engineering such as code coverage, function inlining, mutation testing, metamorphic property-based testing, and more. We hope that this article forms the basis for researchers to investigate further synergies across software analysis and computational law.

Appendix A. Excerpt from Internal Revenue Code

In this appendix, we list excerpts from 26 U.S.C. §63, §1, and §151 (as of April 2024) that are provided to an LLM for reasoning about tasks described in the article above.

§63. Taxable income defined

…

(b)Individuals who do not itemize their deductions In the case of an individual who does not elect to itemize his deductions for the taxable year, for purposes of this subtitle, the term “taxable income” means adjusted gross income, minus—

(1) the standard deduction,

…

(c)Standard deduction For purposes of this subtitle—

(1)In general Except as otherwise provided in this subsection, the term “standard deduction” means the sum of —

(A) the basic standard deduction, and

(B) the additional standard deduction.

(2)Basic standard deduction For purposes of paragraph (1), the basic standard deduction is—

(A) 200 percent of the dollar amount in effect under subparagraph (C) for the taxable year in the case of—

(i) a joint return, or

(ii) a surviving spouse (as defined in section 2(a)),

(B) $4,400 in the case of a head of household (as defined in section 2(b)), or

(C) $3,000 in any other case.

(3)Additional standard deduction for aged and blind

For purposes of paragraph (1), the additional standard deduction is the sum of each additional amount to which the taxpayer is entitled under subsection (f).

(4)Adjustments for inflation In the case of any taxable year beginning in a calendar year after 1988, each dollar amount contained in paragraph (2)(B), (2)(C), or (5) or subsection (f) shall be increased by an amount equal to—

…

(7)Special rules for taxable years 2018 through 2025 In the case of a taxable year beginning after December 31, 2017, and before January 1, 2026—

(A) Increase in standard deductionParagraph (2) shall be applied—

(i) by substituting “$18,000” for “$4,400” in subparagraph (B), and

(ii) by substituting “$12,000” for “$3,000” in subparagraph (C).

(B) Adjustment for inflation

(i) In general

Paragraph (4) shall not apply to the dollar amounts contained in paragraphs (2)(B) and (2)(C).

(ii) Adjustment of increased amountsIn the case of a taxable year beginning after 2018, the $18,000 and $12,000 amounts in subparagraph (A) shall each be increased by an amount equal to—

(I) such dollar amount, multiplied by

(II) the cost-of-living adjustment determined under section 1(f)(3) for the calendar year in which the taxable year begins, determined by substituting “2017” for “2016” in subparagraph (A)(ii) thereof.

If any increase under this clause is not a multiple of $50, such increase shall be rounded to the next lowest multiple of $50.

…

(f)Aged or blind additional amounts

(1)Additional amounts for the aged The taxpayer shall be entitled to an additional amount of $600—

(A) for himself if he has attained age 65 before the close of his taxable year, and

(B) for the spouse of the taxpayer if the spouse has attained age 65 before the close of the taxable year and an additional exemption is allowable to the taxpayer for such spouse under section 151(b).§151. Allowance of deductions for personal exemptions

§1. Tax imposed

…

(f)Phaseout of marriage penalty in 15-percent bracket; adjustments in tax tables so that inflation will not result in tax increases

(1)In general

Not later than December 15 of 1993, and each subsequent calendar year, the Secretary shall prescribe tables which shall apply in lieu of the tables contained in subsections (a), (b), (c), (d), and (e) with respect to taxable years beginning in the succeeding calendar year.

…

(3)Cost-of-living adjustment For purposes of this subsection—

(A) In generalThe cost-of-living adjustment for any calendar year is the percentage (if any) by which—

(i) the C-CPI-U for the preceding calendar year, exceeds

(ii) the CPI for calendar year 2016, multiplied by the amount determined under subparagraph (B).

(B) Amount determinedThe amount determined under this clause is the amount obtained by dividing—

(i) the C-CPI-U for calendar year 2016, by

(ii) the CPI for calendar year 2016.

(C) Special rule for adjustments with a base year after 2016

For purposes of any provision of this title which provides for the substitution of a year after 2016 for “2016” in subparagraph (A)(ii), subparagraph (A) shall be applied by substituting “the C-CPI-U for calendar year 2016” for “the CPI for calendar year 2016” and all that follows in clause (ii) thereof.

(4)CPI for any calendar year

For purposes of paragraph (3), the CPI for any calendar year is the average of the Consumer Price Index as of the close of the 12-month period ending on August 31 of such calendar year.

…

(6)C-CPI-U For purposes of this subsection—

…

(B) Determination for calendar year

The C-CPI-U for any calendar year is the average of the C-CPI-U as of the close of the 12-month period ending on August 31 of such calendar year.

…

§151. Allowance of deductions for personal exemptions

(a)Allowance of deductions

In the case of an individual, the exemptions provided by this section shall be allowed as deductions in computing taxable income.

(b)Taxpayer and spouse

An exemption of the exemption amount for the taxpayer; and an additional exemption of the exemption amount for the spouse of the taxpayer if a joint return is not made by the taxpayer and his spouse, and if the spouse, for the calendar year in which the taxable year of the taxpayer begins, has no gross income and is not the dependent of another taxpayer.

Appendix B. Inflation data for cost-of-living adjustments

The following table provides (hypothetical) values for the Chained Consumer Price Index for All Urban Consumers (C-CPI-U) for calendar years 2017–2024, calculated using the averaging methodology in §1(f)(6)(B).

Year	Value
2017	138.2
2018	141.0
2019	143.2

2020	144.8
2021	149.1
2022	159.9
2023	168.3
2024	172.5

References

1. Robert S Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.
2. Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.
3. Andrew Blair-Stanek, Nils Holzenberger, and Benjamin Van Durme. Shelter Check: Proactively finding tax minimization strategies via AI. *Tax Notes Federal, Dec*, 12, 2022.
4. Andrew Blair-Stanek, Nils Holzenberger, and Benjamin Van Durme. Can GPT-3 perform statutory reasoning? *arXiv preprint arXiv:2302.06100*, 2023.
5. Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learn- ers. *Advances in neural information processing systems*, 33:1877–1901, 2020.
6. T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, The Hong Kong University of Science and Technology, 1998.
7. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, ICFP, 2000.
8. Jack W. Davidson and AM Hollersnm. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89, 1992.
9. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
10. Damien Falato. Is it a trade or business? or a hobby or investment? https://www.thetaxadviser.com/newsletters/2023/may/is-it-a-trade-or-business-or-a-hobby-or-investment.html, May 2013. Online; Retreived March 9, 2024.
11. Randy Goebel, Yoshinobu Kano, Mi-Young Kim, Juliano Rabelo, Ken Satoh, and Masaharu Yoshioka. Summary of the competition on legal information, extraction/entailment (COLIEE) 2023. In *Proceedings of the Nineteenth International Conference on Artificial Intelligence and Law*, pages 472–480, 2023.
12. Oliver Goodenough and Susan Salkind. Computable Contracts and Insurance: An Introduction. *MIT Computational Law Report*, Aug 22 2022.

13. Nils Holzenberger, Andrew Blair-Stanek, and Benjamin Van Durme. A dataset for statutory reasoning in tax law entailment and question answering. *arXiv preprint arXiv:2005.05257*, 2020.
14. Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.
15. Sarah B Lawsky. A logic for statutes. *Fla. Tax Rev.*, 21:60, 2017.
16. Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Software*, 38(4):22–27, 2021.
17. David R MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.
18. Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: a programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021.
19. Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th international conference on software engineering*, pages 537–548, 2018.
20. Internal Revenue Service. Publication 525—taxable and nontaxable income. *Department of the Treasury—IRS Publications*, 2023.
21. Masaharu Yoshioka, Yasuhiro Aoki, and Youta Suzuki. BERT-based ensemble methods with data augmentation for legal textual entailment in COLIEE statute law task. In *Proceedings of the eighteenth international conference on artificial in. telligence and law*, pages 278–284, 2021.
22. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transac- tions on Software Engineering*, 28(2):183–200, 2002.

Footnotes

1. At the time of writing (April 2024), the inflation data for 2024 and consequently the standard deduction for 2025 is not yet known; so, we are certain that an LLM will not be able to use external knowledge to solve this problem. ↩