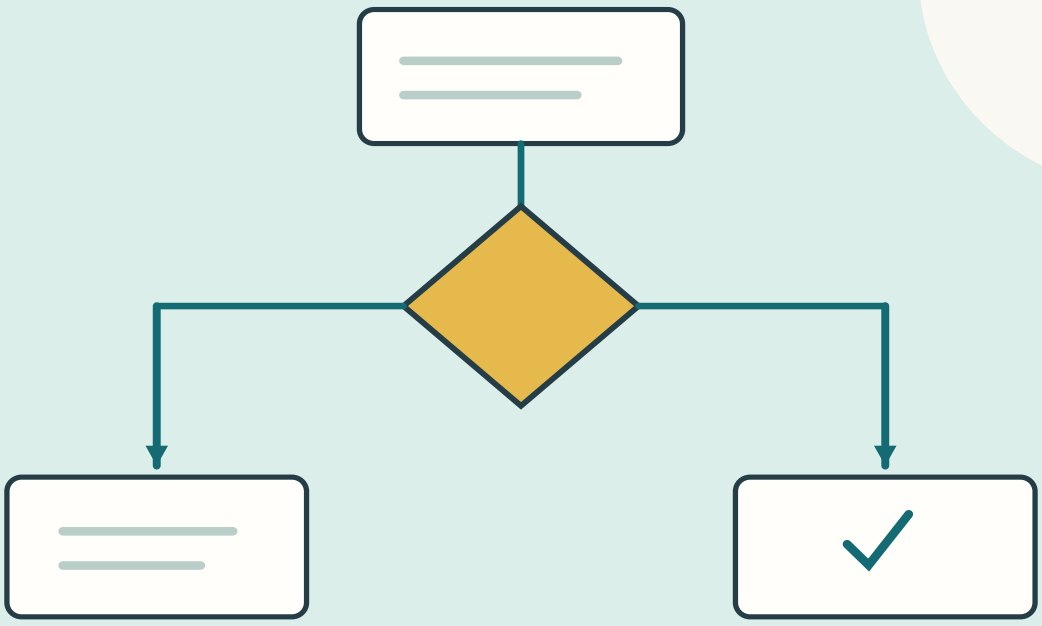
PROCEDURE DECISION TABLES

Converting Procedures into Decision Tables



CONDITION / ACTION

Converting Procedures into Decision Tables

Learning outcomes

- Translate three binary conditions into a complete decision table.
- Apply explicit precedence to four outcomes.
- Test eight combinations and flag unknown inputs.
- Separate procedure inputs, predicates and outcomes with explicit domains.
- Enumerate complete combination spaces and translate precedence into mutually exclusive guards.
- Distinguish wildcard conditions from unknown facts and test determinate outcomes under partial information.
- Find overlaps, gaps and boundary errors using independent test cases.
- Deliver a traced decision table, executable reference logic and a reproducible change-impact report.

Original teaching material. All organizations, people, records and operational rules in exercises are synthetic. Use only the supplied non-sensitive inputs. This is a practice course, not an accredited or regulated certification.

Study the foundations, work through the applied units, complete the practice tasks, and then attempt the assessment. Use only the supplied fictional practice data. Keep your original inputs and your revised work separate.

Core concepts

Extract conditions and outcomes separately

A procedure contains facts to test and actions to take. A condition is a question that can be evaluated, such as is the form complete. An outcome is what happens after conditions are evaluated, such as request missing information. Keep these concepts separate; complete is not an action and approve is not an input. For this exercise the three binary conditions are form_complete, stock_available and manager_approval. Each can be yes or no in the eight test cases. A missing condition value is unknown, not automatically no.

Make precedence explicit

The supplied procedure checks completeness first. If the form is incomplete, request information regardless of stock or approval. If complete but stock is unavailable, waitlist regardless of approval. If complete and stock exists but approval is absent, route for approval. Only complete, stocked and approved requests are released. This order prevents conflicting outcomes. A decision table can represent the same logic through eight fully enumerated rows or through compressed rules using a dash for irrelevant conditions. A dash means the value cannot change that rule’s result; it does not mean the source value is missing.

Check coverage and exclusivity

Three binary conditions create 2 times 2 times 2, or eight combinations. Enumerate them systematically so none is skipped. Each combination should produce exactly one outcome. If two rules apply with different outcomes, the table is ambiguous unless a precedence rule resolves them. If no rule applies, coverage is incomplete. The exercise also includes an unknown-input note: an unknown form status must be checked before taking a release action. Do not extend the binary table to unknown values by silently interpreting them as false.

Trace every rule back to the procedure

Add a source-clause column to the table. This is a compact proof that the logic came from the actual procedure rather than the learner’s intuition about what a good process should be. Run all eight test cases and compare the result with a direct reading of the prose. A correct table is not just nicely formatted; it behaves the same way as the original rule. If the source itself leaves a condition unresolved, identify that gap for the process owner instead of creating a new policy.

Worked example

Demonstration: for complete=yes, stock=no, approved=yes, the stock clause produces WAITLIST. Approval does not override missing stock. A compressed rule can read complete=yes, stock=no, approved=any. For complete=no, the first clause applies even if the other two fields are yes.

Foundation practice

Procedure

P1: If the form is incomplete, REQUEST_INFO.

P2: Otherwise, if stock is unavailable, WAITLIST.

P3: Otherwise, if manager approval is absent, SEEK_APPROVAL.

P4: Otherwise, RELEASE.

Cases

id	form complete	stock available	manager approval
T1	False	False	False
T2	False	False	True
T3	False	True	False
T4	False	True	True
T5	True	False	False

id	form complete	stock available	manager approval
T6	True	False	True
T7	True	True	False
T8	True	True	True

Unknown case

U1 has unknown form_complete, stock_available=yes, manager_approval=yes. The source gives no unknown handling beyond requiring confirmation before release.

Foundation tasks

Guided preparation

Instructions

Read the complete input pack. Create the output columns specified in the capstone, keep the original IDs, and state the unit of work. Complete the first two records manually before using an optional AI assistant. Record any assumption that could change the result.

Independent capstone

Instructions

Produce a table covering all eight binary combinations, four outcomes and source clauses. Run all eight test cases. Record U1 separately as unresolved pending a form-completeness check, not as an approved release.

Peer verification and revision

Instructions

Exchange only your student work with a peer. Ask the peer to trace two ordinary results and every exception back to the supplied input. Record the exact item corrected, the original value, revised value and reason. Submit the revised artifact plus a short explanation of the most important remaining limitation.

01 Specify the decision before drawing the table

A decision table expresses how a defined set of input facts determines a defined output. It is not a calendar, a checklist of tasks already completed, or a diagram of every step in an organization. Before making columns, write the decision question in one sentence. For the foundation, the question is which routing outcome applies to one request given form completeness, stock availability and manager approval. The table evaluates a snapshot of those inputs; it does not fill in the form, replenish stock or grant approval.

Define the unit of work. One row in an input case file represents one request at one evaluation point. One row in a fully enumerated decision table represents one possible combination of condition values. These are not the same kind of row. Many real requests can

share the same combination and therefore the same rule. Conversely, one request can be evaluated again after facts change. Keep case_id, rule_id and evaluation observation separate so a rule row is not mistaken for a business record.

Write a data dictionary before translating prose. A condition should have a meaningful name, a precise question and an allowed domain. form_complete means the supplied form has all required information, not that the request has been approved. stock_available means the required stock is available for this request, not that the storeroom contains any stock. manager_approval means the required approval is present, not that a manager has seen the request. These distinctions prevent a table from appearing correct while using the wrong inputs.

An outcome is a result label such as REQUEST_INFO, WAITLIST, SEEK_APPROVAL or RELEASE. The label does not authorize performing the real action. In this course, the learner returns a recommendation label and its source clause. Keep output values stable and avoid substituting synonyms in machine-readable files. A human can understand "ask for details" as REQUEST_INFO, but a program comparing exact labels may treat it as a new outcome. Readable explanations belong beside, not instead of, the defined output code.

Finally define what the table does not cover. The foundation's eight combinations cover known binary inputs only. They do not establish a policy for absent or contradictory facts. The supplied unknown case requires verification before release, not automatic conversion of unknown to false. A clear boundary prevents later users from applying a binary table to a wider domain and assuming its completeness claim still holds.

A decision contract also identifies whether the output is a single label or a collection of actions. The foundation requests one label, so returning REQUEST_INFO and WAITLIST together is not a more helpful answer. It changes the output type and bypasses precedence. If a source genuinely requires several actions, that is a different contract and may need multiple outputs or a separate action list. Do not infer multi-action semantics merely because several conditions are false. Write the output cardinality explicitly: exactly one business outcome for each valid known tuple. Validation errors and unresolved partial inputs are reported through their separately declared channels.

E01-X1 Create a decision contract

For the foundation procedure, state the unit of work, three input names with domains, four output labels, and one limitation for unknown inputs.

Submit: A compact data dictionary and decision statement.

02 Read otherwise as a guard on every later rule

The word otherwise carries logic. In the foundation, P2 does not mean "If stock is unavailable, always waitlist." It means "If the earlier incompleteness condition did not apply, and stock is unavailable, waitlist." P3 similarly assumes the form is complete and stock exists before testing approval. These inherited conditions must appear either in rule order or in explicit guards. Omitting them creates overlapping rules with incompatible outcomes.

Write each clause as a complete predicate. P1 is not form_complete. P2 is form_complete and not stock_available. P3 is form_complete and stock_available and not manager_approval. P4 is form_complete and stock_available and manager_approval. The final otherwise is not an unrestricted default that may override an earlier outcome. It applies only when all earlier tests have failed. This expanded form is longer than the prose but makes the precedence visible.

A worked case is form_complete=false, stock_available=false, manager_approval=true. The result is REQUEST_INFO under P1. A rule that checks stock first would produce WAITLIST, which may seem reasonable operationally but is not the supplied procedure. Conversion is a fidelity task. If a learner thinks another priority would be better, that is a proposed policy change and should be discussed separately from whether the conversion is correct.

There are two defensible representations. An ordered first-match table can preserve the prose sequence and stop at the first matching rule. An order-independent table can make each rule's complete guard explicit so exactly one row matches. The second representation is easier to audit for exclusivity in this small exercise. Do not mix the two styles without declaring which semantics apply. A table with broad conditions and no stated ordering leaves the reader to guess how conflicts are resolved.

One useful translation practice is to write the path to each outcome in words before writing symbols. To RELEASE, the form must be complete, stock must be available and approval must be present. To SEEK_APPROVAL, the first two must be true and approval false. To WAITLIST, completeness must be true and stock false; approval cannot change that result. These verbal paths help catch reversed booleans and missing inherited conditions before they become compact but misleading table cells.

Read the procedure aloud using the word reached before every later test: if P2 is reached, form completeness has already passed. This path reasoning is useful when prose contains nested otherwise clauses. A common transcription mistake is to attach an otherwise to the nearest visual line rather than to its grammatical condition. Indentation can help, but it is not evidence by itself. Reconstruct the complete path from the actual wording, then compare the resulting guards with all example cases. If the source's nesting remains ambiguous, record the two possible interpretations and ask for clarification rather than choosing the more convenient table.

E02-X1 Expand all guards

Write the complete Boolean guard for each foundation outcome and evaluate the case false, false, true in the input order form, stock, approval.

Submit: Four guarded rules and one case trace.

03 Enumerate the combination space systematically

For independent binary conditions, each condition doubles the possible combinations. Three conditions give 2 × 2 × 2 = 8 combinations. This is a statement about the declared input domain, not a prediction of how often each combination occurs in practice. A rare combination still belongs in the completeness check unless the task supplies a valid constraint excluding it. Do not omit inconvenient cases simply because they seem unlikely.

Choose a fixed input order and a fixed value order. In the foundation, use form_complete, stock_available, manager_approval, with false before true. Toggle approval every row, stock every two rows and form every four rows. This creates all eight tuples without duplication. The technique is equivalent to binary counting, but retaining named columns is safer than relying on bit positions alone. A tuple is meaningful only when its column order is known.

The complete outputs in that order are REQUEST_INFO four times, WAITLIST twice, SEEK_APPROVAL once and RELEASE once. Those counts are useful as a secondary check, but they are not a complete proof. Swapping the outcomes of the last two rows preserves the distribution while breaking the procedure. Verify every tuple-to-outcome pairing, not just totals. The same distinction applies to case counts: eight rows can contain one duplicate and one missing combination.

If conditions are not independent, document the constraint separately. For example, a fictional procedure may define approval_present as impossible when form_complete is false. That restriction would reduce the allowed domain only if the source or task explicitly states it. The foundation does not state such a restriction, so all eight combinations remain in scope. Do not infer a domain rule from what a well-run process usually does.

A combination inventory is also a teaching tool. It shows which inputs affect each outcome and creates a reusable regression suite. When a procedure changes, compare outputs for the same tuples under the old and new rules. A difference is not automatically an error, but it must correspond to the declared change. If a revised implementation changes a tuple outside the intended scope, the inventory exposes unintended impact. Exhaustive enumeration is practical for small finite domains and much stronger than testing only one success case.

E03-X1 Construct the foundation truth table

List all eight tuples in false-before-true order and give the outcome and source clause for each. Also provide the outcome distribution.

Submit: Eight-row table and four outcome counts.

04 Compress rules without changing their meaning

A compressed table replaces an input value with a dash when that value cannot change the rule's result. The dash is a wildcard over the declared domain, not missing data. In the foundation, an incomplete form produces REQUEST_INFO for either stock value and either approval value. The compressed row false, dash, dash therefore represents four binary combinations. A complete form with unavailable stock produces WAITLIST for either approval value, so true, false, dash represents two combinations.

The remaining rows are true, true, false for SEEK_APPROVAL and true, true, true for RELEASE. Together the four compressed rows cover 4 + 2 + 1 + 1 = 8 combinations. This sum is meaningful because the guards are mutually exclusive. If rows overlap, simply adding their represented counts double-counts some inputs and may conceal gaps elsewhere. Always check intersections before relying on coverage arithmetic.

Compression is a proof obligation. To replace a condition with a dash, compare the outcomes for every allowed value of that condition while holding the other guarded conditions fixed. If all outcomes remain the same, the condition is irrelevant within that rule. It may still be important elsewhere. Approval is irrelevant in the incomplete-form branch but essential when the form is complete and stock exists. A dash does not mean the organization never needs approval information.

Do not put a dash into an input record to indicate unknown. A case file should use the declared unknown representation, such as null, and retain the fact that the value has not been established. The table's wildcard says both values lead to the same result in a branch. The case's unknown says the evaluator lacks a fact. Those concepts can interact, but they should not share an ambiguous notation. This course uses dash only in displayed rule tables and null only in data for unknown inputs.

A compressed table can be easier for a person to read than eight rows, yet more difficult to implement correctly if the matching semantics are unstated. Include column domains, the meaning of dash, source-clause references and the declared hit policy. For this foundation compression, the explicit guards make exactly one rule match every known binary tuple. A reader does not need to rely on row order to get the correct result.

To check a proposed compression manually, expand it back into known tuples. For true,false,dash, write true,false,false and true,false,true. Compare both with the uncompressed table; both must say WAITLIST. This reverse expansion is an independent readability check, not redundant formatting work. It catches an accidental dash in the stock column, which would wrongly include true,true,false and true,true,true. Compression should be reversible: the compressed table and the full table must describe the same set of input-output pairs. If they do not, the shorter representation is not an equivalent simplification, even if its outcome counts look familiar.

E04-X1 Compress and justify

Write the four compressed foundation rows in input order form, stock, approval, and give the number of binary combinations represented by each. Explain why dash is not null.

Submit: Four rows, coverage sum and notation distinction.

05 Declare how multiple matching rules are handled

A hit policy tells the reader what to do when more than one rule matches. This course uses two local concepts: unique-match tables require exactly one matching rule for every valid known input; ordered first-match tables use the first matching row. Decision-modeling standards also define hit policies, but the exercises here are plain teaching tables, not a claim of compliance with a particular engine or notation. State the semantics explicitly rather than assuming a spreadsheet or AI tool will infer them.

Consider a flawed table with A: form_complete=false → REQUEST_INFO; B: stock_available=false → WAITLIST; C: manager_approval=false → SEEK_APPROVAL; D: all three true → RELEASE. For false, false, false, rules A, B and C all match. Under unique-match semantics, this is invalid because the outcomes conflict. Under ordered first-match semantics with A before B before C, the selected outcome happens to match the procedure. Reordering the rows changes behavior, so the order is part of the logic.

First-match tables can be useful, but they can also hide unreachable rules. If a broad default row appears first and matches everything, later rows never determine an output. Such a table may have no missing cases and still be wrong. Review both which rules match and which rule wins. A test suite should expose cases that distinguish the order, not only cases in which all matching rules happen to agree.

For small procedures, an explicit unique-match table often offers clearer evidence. Add the inherited guards to B and C so their input regions no longer overlap A. Then row order can change without changing results. This order-independence is not a universal requirement, but it is a useful property when people will sort or copy a table. If sorting rows changes the decision, the table must carry a warning and retain its ordered semantics in every export.

Never resolve overlap by majority vote among matching rules unless the supplied decision contract explicitly defines that behavior. Nor should a system choose the most specific-looking row by intuition. A table is an executable interpretation of a procedure; its matching policy is part of that interpretation. Ambiguous matching is a defect to report or repair through source-supported guards, not an invitation to invent a new priority rule.

In a first-match table, a source-clause column should preserve the intended priority order as well as the clause identity. Exporting the table to a spreadsheet and sorting by outcome name can silently change its behavior. An order-independent table avoids this particular failure because guards already encode precedence. This does not mean sorting is always harmless: sorting case results can still lose the original presentation order unless stable IDs remain. Keep the distinction between semantic row order and display row order explicit. A review should ask what happens if the rule rows are reordered and whether that behavior is intended.

E05-X1 Diagnose an overlap

Using flawed rules A–D above, list all matches for false, false, false. State the unique-match result and first-match result with A,B,C,D order. Repair B and C for order independence.

Submit: Match list, two semantic outcomes and corrected guards.

06 Separate unknown facts from irrelevant conditions

Unknown does not mean false. If form completeness is unknown, stock is available and approval is present, both REQUEST_INFO and RELEASE remain possible under the foundation. Treating unknown as false chooses REQUEST_INFO without evidence; treating it as true chooses RELEASE without evidence. The supplied foundation directs verification before release. It does not authorize a universal unknown-handling policy for every possible input combination.

For the expanded worked exercises, use an explicit local analysis rule: replace each unknown with every allowed Boolean completion, evaluate the known-input procedure, and collect the possible outcomes. If all completions yield the same outcome, report that outcome as logically determined. If outcomes differ, report UNRESOLVED and list the possible outcomes. This is a declared analytical convention for these exercises, not a new operational policy added to the foundation. It demonstrates what follows from partial information without pretending missing facts have been observed.

For form_complete=false, stock_available=null, manager_approval=null, all four completions yield REQUEST_INFO. The unknown later inputs do not matter once the first branch is established. For form_complete=true, stock_available=null, manager_approval=false, the possibilities are WAITLIST and SEEK_APPROVAL. A request for stock confirmation is relevant because learning that value distinguishes the remaining outcomes. Asking for approval again would not resolve this particular uncertainty; its value is already known.

The completion-set method also explains the meaning of a wildcard. A compressed rule with false, dash, dash says every completion of the later conditions has the same result. An unknown case can therefore be determinate in that branch, but not because null equals dash. It is determinate because every compatible known case agrees. Keep this reasoning visible in the output so a reviewer can distinguish evidence-based determination from accidental defaulting.

In real workflows, an organization may still require collection of all fields for audit or later processing. Logical determination of the current routing outcome does not waive those requirements unless a source says it does. This course reports a decision label and the unresolved input fields; it does not authorize skipping data collection or executing the action. The distinction prevents a useful logical shortcut from becoming an unsupported operational shortcut.

A targeted clarification follows the remaining decision branches. For a complete request with stock unknown and approval absent, ask about stock because its two values separate WAITLIST from SEEK_APPROVAL. Asking for an unrelated date cannot resolve that set. For an incomplete form with later fields unknown, the route is already determined under the local convention, but the unknowns should remain in the audit record. This gives a reviewer two different pieces of information: what can be decided now and what facts are still unobserved. Neither should be lost merely to make every row appear complete.

E06-X1 Analyze two partial inputs

Under the declared completion-set convention, evaluate U2=(false,null,null) and U3=(true,null,false). Give possible outcomes, determination status and the field needed to resolve U3.

Submit: Two uncertainty records.

07 Validate domains before applying decision logic

A table's completeness claim applies only to its declared domain. The string "false" is not a Boolean false in a typed data file. The integer zero may be interpreted as false by some programming languages, but that does not make it a valid input under a Boolean-only contract. An empty string can mean missing data, failed extraction or a legitimate text value elsewhere. Do not let a language's convenience rules silently decide your business meaning.

Use a two-stage evaluator. First validate field membership and input types. Then apply the procedure to valid values. For the expanded examples, each required condition must be present and must be true, false or null. Extra fields in an input record can be allowed only if the contract says they are metadata; otherwise reject them to catch misspellings. A missing field is different from an explicitly supplied null if the submission format requires all keys. Report the defect rather than quietly substituting a default.

Python illustrates why exact checks matter. Boolean values are a distinct type but are also related to integers in Python's type system. A loose numeric test can accept true where an integer quantity was expected, while a truthiness test can interpret the nonempty string "false" as true. In the supplied reference code, a condition is valid only when type(value) is bool or value is None. This is a local validation choice, deliberately stricter than accepting every value a language can coerce.

Create negative tests alongside ordinary decision cases. Test "false", 0, an omitted condition, an unexpected misspelled key and a list in place of a Boolean. These tests should fail validation before any outcome is returned. A failed validation is not a policy outcome such as WAITLIST. Keep the error channel separate from the decision label so consumers do not mistake corrupted input for a legitimate branch result.

Domain validation cannot prove the facts are true. A record with four well-typed booleans may still misreport an inspection. The evaluator establishes only that the record fits the declared representation and maps to the procedure under those inputs. Source evidence and observation quality remain separate responsibilities. A careful handoff says "evaluated from supplied inputs" rather than "verified the crate is safe" or any other claim beyond the performed checks.

E07-X1 Classify malformed inputs

Under a required true/false/null contract, classify values true, false, null, "false", 0 and [] as accepted or rejected. Explain why rejected values must not become a business outcome.

Submit: Six decisions and one error-channel explanation.

08 Find gaps and overlaps with witnesses

A useful defect report includes a witness: a concrete input tuple that demonstrates the problem. "The table seems incomplete" is less actionable than "For true,true,false, no rule matches." A gap is a valid input with zero matching rules. An overlap is an input with more matches than the declared hit policy permits. A wrong mapping is an input with a selected outcome different from the source procedure. These defects can occur independently.

Consider a compressed foundation table that accidentally omits the SEEK_APPROVAL row. The tuple true,true,false is a gap. The other seven binary combinations may still produce correct outputs. A test suite that contains only the RELEASE case and one incomplete-form case will miss the defect. Exhaustive enumeration is inexpensive here and provides a complete witness search over the declared eight-case domain.

Now consider a row true,dash,false → SEEK_APPROVAL alongside the correct WAITLIST row true,false,dash → WAITLIST. The tuple true,false,false matches both. The mistake is the missing stock_available=true guard in the approval row. If the table is declared unique-match, the overlap is invalid even if an implementation happens to choose one row. A defect report should name the two rules, the witness tuple and the source path that resolves the intended outcome.

Different outcomes are not the only concern. Two identical duplicate rules may produce the same output and still violate an exactly-one-match table contract. They also complicate maintenance: one copy can be changed while the other remains stale. If the chosen semantics permits multiple agreeing matches, document that explicitly. In this course's unique-match tables, duplicate matching rows are rejected so coverage and traceability stay simple.

A witness can guide repair, but one repaired witness is not the whole regression suite. Adding a guard may fix the overlap for true,false,false while accidentally excluding true,true,false. After every repair, rerun all valid combinations and compare to the direct procedure. Keep the original defect and corrected rule in a change ledger. This creates evidence that the repair addressed the cause rather than merely hard-coding the observed test case.

A compact defect ledger can have five columns: defect ID, witness tuple, matched rule IDs, expected behavior and repair. For a gap, the matched-rule list is empty. For an overlap, list every matching rule rather than only the selected one. For a wrong mapping, record the single selected rule and the source clause it misrepresents. This consistent structure helps distinguish defects that require adding a region, narrowing a guard or changing an output label. It also makes peer review concrete: another learner can replay the witness and see the same failure before discussing the proposed repair.

E08-X1 Produce two defect witnesses

Give a gap witness when the SEEK_APPROVAL row is omitted. Then give an overlap witness for true,dash,false → SEEK_APPROVAL alongside true,false,dash → WAITLIST, and write the corrected approval guard.

Submit: Two witnesses, their consequences and one repair.

09 Turn numeric thresholds into explicit predicates

Not every procedure begins with Boolean inputs. A rule may use quantity, age, weight or elapsed time. Translate such a rule into a predicate with a defined unit and boundary. In a separate fictional courier exercise, a parcel is SMALL when mass_kg is greater than zero and at most 5; it is LARGE when mass_kg is greater than 5 and at most 20. Mass above 20 is OUT_OF_RANGE. Nonpositive or missing mass is INVALID_INPUT. These are local exercise rules, not claims about any real courier.

The boundary values do most of the diagnostic work. At 5 kg, the SMALL predicate is true and LARGE false. At 20 kg, LARGE is true; above 20, OUT_OF_RANGE applies. Testing only 2 and 12 kg cannot reveal whether the implementation used less-than instead of less-than-or-equal. Include values just below, exactly at and just above each boundary, with a declared precision. In this exercise, use decimal values with two decimal places and do not introduce measurement tolerance.

Units belong in the contract. A mass value of 5000 is meaningless without knowing whether it represents grams or kilograms. If conversion is authorized, record it explicitly before evaluating the predicate. Five thousand grams equals 5 kilograms under the stated conversion of 1000 grams per kilogram. Do not infer a unit from the value's size or choose a rounding rule that changes category membership. A conversion ledger should preserve the raw number, unit, normalized number and operation.

Intervals must be both complete and non-overlapping for valid inputs. The predicates 0 < mass <= 5 and 5 < mass <= 20 share no value. By contrast, mass <= 5 and mass >= 5 overlap at 5. Predicates mass < 5 and mass > 5 leave a gap at 5. Writing interval endpoints explicitly helps reviewers identify these errors before testing. A table cell labeled "medium" without its numeric definition is not a reproducible condition.

Do not let an AI-generated decision table invent thresholds that sound reasonable. If the source says small parcels are light but gives no number, the table should record an unresolved definition. A numeric boundary is a policy choice, not a formatting convenience. This course supplies thresholds precisely so learners can test conversion fidelity rather than negotiate a real operational rule.

A decimal boundary exercise should not silently borrow floating-point tolerance from a programming library. If inputs are supplied as exact two-decimal teaching values, compare the intended decimal quantities. A value reported as approximately 5 kg would require an uncertainty rule that this courier exercise does not supply. Keep measurement uncertainty separate from a programming representation issue. The table may be mathematically precise while the physical measurement remains uncertain. In a real task, obtain a declared treatment of approximate measurements before assigning a definite category; do not round simply because one category seems operationally easier.

E09-X1 Test the threshold boundaries

Apply the courier exercise rules to mass_kg values 0, 0.01, 4.99, 5.00, 5.01, 20.00 and 20.01. Also normalize 5000 g using 1000 g/kg and classify it.

Submit: Eight concrete results and the conversion calculation.

10 Test equivalence with an independent procedure reading

A generated table should be compared with an independent interpretation of the prose. If the same function produces the table and produces the expected answers, matching outputs may only show that the function agrees with itself. Independence does not require a different person in every exercise, but it does require a separate representation or manually derived oracle that can reveal the original mistake. For the foundation, the ordered clause reading and the fully guarded table provide two forms of the same intended logic.

A reference evaluator can follow the prose directly: if the form is incomplete, return REQUEST_INFO; otherwise if stock is unavailable, return WAITLIST; otherwise if approval is absent, return SEEK_APPROVAL; otherwise return RELEASE. A separate table matcher checks each row's guard. Enumerate all eight known tuples and compare the two results. Also compare them to the manually stated expected sequence. This three-way check reduces the chance that a shared implementation mistake goes unnoticed.

A test should record input, expected outcome, actual outcome and source clause. Passing means the actual outcome equals the independently derived expected outcome for that input. It does not mean the real-world procedure is optimal or safe for every context. The procedure could itself contain a policy defect. Conversion validation establishes behavioral equivalence within the supplied domain, not approval of the source policy.

Negative testing strengthens the comparison. Deliberately swap the stock and approval tests in a copy of the evaluator. The tuple true,false,false should expose the change: the correct result is WAITLIST, not SEEK_APPROVAL. Deliberately change the final approval condition to true when it should be false; the SEEK_APPROVAL and RELEASE cases should fail. A test suite that cannot detect these plausible mutations is not exercising the distinctions that matter.

Keep tests read-only with respect to course content. A verification script should load current JSON and assets, compare them to independent expectations, and report failures. It should not regenerate the expected file just before checking it, because that can overwrite the evidence of a defect. Derive asset paths from the script's own location so moving the bundle does not change which data are tested. Reproducibility includes both the logic and the identity of the files under review.

E10-X1 Design a mutation test

State one concrete tuple that detects swapping the stock and approval branches, and one that detects a reversed approval outcome. Give correct and mutated outcomes under these exact mutations: M1 checks missing approval before missing stock; M2 swaps SEEK_APPROVAL and RELEASE after form and stock pass.

Submit: Two test records with expected and mutated outcomes.

11 Write readable reference logic without coercing inputs

The following reference logic demonstrates the expanded completion-set convention for the foundation. It is not a workflow automation: it returns a label and possible outcomes but performs no external action. Read the code together with the data dictionary. The input order is form_complete, stock_available, manager_approval. A tuple is accepted only when it has exactly three values and each value is a Boolean or None. The strict type check rejects strings and numeric substitutes.

The known function follows the prose in order. The analyze function replaces each None with both Boolean values, evaluates every completion and sorts the distinct results. Sorting is only for stable presentation of the possible-outcome list; it does not create a priority among outcomes. When the set has one member, the result is determined under the local convention. When it has more than one, the result is UNRESOLVED. The missing-field list remains available even when the current outcome is determined.

Notice that this method does not choose the most frequent outcome among completions. Possible completions are logical alternatives, not probability-weighted observations. If three completions yield one outcome and one yields another, the result is still unresolved. Majority voting would invent an assumption about the missing facts. Similarly, alphabetical order in the sorted result list does not make its first member the recommended action.

Run the provided code only with the supplied fictional cases or similarly non-sensitive local data. No network service is needed. If Python is unavailable, the same logic can be performed manually by listing completions and comparing outcomes. Record the method actually used. Do not claim that code ran when you only inspected it, and do not treat a successful run as verification of source facts that were never checked by the code.

For maintenance, keep the evaluator small and the tests separate. A concise conditional function is easier to compare with the prose than a clever compressed expression. Use exact outcome names and explicit input ordering. If the procedure changes, update its version and expected cases together under review, but keep old expected outputs available for impact comparison. Readability is a verification feature when the audience includes people who must trace each branch back to a paragraph.

```
from itertools import product

NAMES = ("form_complete", "stock_available", "manager_approval")

def known(values):
    form, stock, approval = values
    if not form:
        return "REQUEST_INFO"
    if not stock:
        return "WAITLIST"
    if not approval:
        return "SEEK_APPROVAL"
    return "RELEASE"

def analyze(values):
```

```
    if len(values) != 3:
        raise ValueError("Exactly three conditions are required")
    if any(v is not None and type(v) is not bool for v in values):
        raise ValueError("Conditions must be Boolean or None")
    domains = [(False, True) if v is None else (v,) for v in values]
    possible = sorted({known(x) for x in product(*domains)})
    result = possible[0] if len(possible) == 1 else "UNRESOLVED"
    unknown = [n for n, v in zip(NAMES, values) if v is None]
    return {"result": result, "possible": possible, "unknown": unknown}
```

E11-X1 Trace the evaluator

Using the code above, determine results for (false,null,true), (true,true,null) and (true,true,true). For each, list all possible outcomes and the returned result.

Submit: Three evaluator traces.

12 Preserve source traceability through table revisions

Every rule row should point to the clause that justifies its outcome and inherited conditions. A clause label alone is useful, but a short guard explanation makes later review easier. For the foundation WAITLIST row, cite P2 and note that otherwise inherits form completeness from the failed P1 condition. This avoids presenting the rule as an isolated stock test. Traceability should explain the path, not merely attach a decorative reference.

Suppose a fictional amendment says, "For complete requests, missing manager approval is checked before stock availability; incomplete forms still receive REQUEST_INFO." This is an explicit precedence change. Under the new procedure, true,false,false changes from WAITLIST to SEEK_APPROVAL. The tuples true,false,true, true,true,false and true,true,true retain WAITLIST, SEEK_APPROVAL and RELEASE respectively. All incomplete-form cases retain REQUEST_INFO. Only one of the eight binary tuples changes outcome.

A revision ledger should include procedure version, rule change, affected tuples, old outcome, new outcome and source of the change. Do not use an amendment's source-message date as the time you executed tests. If test execution time matters, record the actual observation separately. The packet may contain no such timestamp, in which case it should remain unstated. Provenance about a procedure and provenance about verification are related but different records.

Keep the old and new tables distinguishable. Overwriting the old expected outcomes removes the ability to explain why a prior case received a different result. This is particularly important when decision outputs are used in reports. A difference may reflect changed facts, changed rules or a corrected implementation. A clear version record lets a reviewer identify which kind of change occurred instead of assuming all differences are errors.

After a policy change, rerun the entire domain rather than only the intended affected tuple. The single expected difference becomes a strong regression property: every other tuple should retain its prior output. This is a practical way to detect accidental changes introduced while editing guards. The change-impact report should state both what changed and what was verified unchanged, with a bounded claim about the declared binary domain.

When a case changes outcome, preserve enough context to explain whether the cause was a fact update or a procedure update. A request can move from WAITLIST to RELEASE because stock became available and approval was already present, without any rule change. The same stored tuple can change from WAITLIST to SEEK_APPROVAL after the explicit precedence amendment, without any fact change. Those histories answer different audit questions. A case ledger should therefore identify the input snapshot and procedure version separately. A source-message date alone identifies neither the exact observation snapshot nor the time a test was performed.

E12-X1 Compute a precedence-change ledger

Apply the amendment above to all eight foundation tuples. List the changed tuple with old/new outcomes and state the count of unchanged tuples.

Submit: Exact impact ledger and regression expectation.

13 Independent procedure: route the Meridian crate intake

The Meridian gallery uses a fictional intake routing procedure for exhibition crates. The decision is a routing label for one crate record, not a claim that a physical crate has been inspected and not permission to move it. The four inputs are damage_present, contents_match, label_valid and bay_available. Each is Boolean when observed; null means the fact is unknown. The source clauses below define precedence. Use those clauses exactly rather than importing the foundation's form, stock or approval rules.

M1: If damage_present is true, return ISOLATE. M2: Otherwise, if contents_match is false, return RECOUNT. M3: Otherwise, if label_valid is false, return RELABEL. M4: Otherwise, if bay_available is false, return HOLD. M5: Otherwise, return ACCEPT. M6: For this exercise, analyze unknown inputs by all Boolean completions. If every completion gives the same result, return that result and retain the unknown-field list; otherwise return UNRESOLVED and the complete possible-outcome set. Invalid representations must be rejected before evaluation.

Create a fully enumerated table for the sixteen known-input combinations, using false before true and the input order given above. Assign case IDs K01 through K16 in that order. Include outcome and source clause on every row. Then create a compressed unique-match table with explicit inherited guards. Explain how many known combinations each compressed row represents. Do not assume the first condition uses the same true/false orientation as the foundation: here true damage triggers the first outcome.

Next evaluate the supplied partial-input cases U01 through U06. The case file also includes invalid records V01 through V04, which must produce validation errors rather than business labels. For every valid partial case, list compatible outcomes, the returned result and unknown input names. Give a targeted information request only when more than one outcome remains possible. Do not claim that a logically determined route removes the need to collect other facts for an actual organization.

Your submission consists of the two rule tables, all sixteen known-case results, six partial-case records, four validation outcomes, and a read-only test report. Include a source-clause map and a short limitation statement. You may implement your own evaluator, but a manual trace is acceptable if it covers the same cases. Keep the independent expected results separate from any code that generates candidate outputs. The goal is an auditable conversion of the complete procedure, not a plausible-looking set of five outcome labels.

ID	damage present	contents match	label valid	bay available
U01	true	null	null	null
U02	false	false	null	null
U03	false	true	null	true
U04	null	true	true	true
U05	false	true	true	null
U06	null	false	false	false
V01	"false"	true	true	true
V02	0	true	true	true
V03	false	true	true	missing
V04	false	true	true	true

V04 also contains the unexpected extra key bay_availble=true. This is a misspelling, not an additional approved metadata field. All records must contain exactly id and the four named conditions. The JSON asset supplies all sixteen known tuples as well as these ten special cases.

E13-X1 Build the Meridian decision package

Using M1–M6 and assets/meridian_cases.json, build the sixteen-row truth table and five-row compressed table. Evaluate U01–U06 and V01–V04. Include exact source clauses, possible outcomes, unknown fields and validation errors.

Submit: Complete decision package and independent checks.

14 Strengthen the capstone with adversarial and metamorphic checks

A complete truth table establishes all known Boolean mappings, but a robust submission also tests how the implementation reacts to malformed and partially known inputs. These are different domains. Six selected partial cases are useful examples, not exhaustive coverage of every null pattern. Four conditions with three allowed values each create $3 \times 3 \times 3 \times 3 = 81$ tuples. The completion-set convention can be tested exhaustively over that small domain without making any assumption about real-world frequencies.

Metamorphic checks test a relationship between inputs when the exact output is already constrained by the procedure. In Meridian, once damage_present=true, changing any later input must not change ISOLATE. Once damage_present=false and contents_match=false, changing label or bay availability must not change RECOUNT. These properties follow from precedence. They can expose accidental use of a later condition even when a few ordinary examples happen to pass.

A different relationship tests information refinement. Replacing null with an observed Boolean narrows the set of compatible outcomes; it cannot introduce a new outcome outside the original completion set. If a candidate evaluator reports possible outcomes {ACCEPT, RELABEL} for a partial case and later reports HOLD after only resolving label_valid, something is inconsistent. This property depends on the input facts and procedure remaining otherwise unchanged. A policy amendment is not information refinement and must be analyzed separately.

Use mutation tests to challenge your test suite. In a copy of the Meridian logic, reverse the damage test, omit the contents guard from the relabel rule, or change the HOLD branch to ACCEPT. Identify a concrete witness for each mutation before running it. A test suite that catches only syntax errors may still miss a dangerous semantic inversion. The purpose of a mutation is not to add a new policy but to demonstrate that the checks would detect a plausible implementation defect.

Record test scope honestly. If you enumerate all sixteen known tuples and all 81 ternary tuples, state both domains and the declared completion convention. If you test only the supplied cases, state their IDs. Do not call a table universally correct because those cases pass. The source procedure may have omitted a real operational exception, and no finite test of the supplied domain can establish facts outside it. Strong verification is specific about what it proves.

For an exhaustive partial-domain check, do not store 81 rows merely to claim volume. Summarize the domain and retain a reproducible evaluator plus independent invariants. Test each refined input by replacing one null with false and then true, confirming that its possible-outcome set stays within the original set. Test all damage-true tuples against the M1 dominance property. These checks examine structural relationships across many cases rather than relying only on a long list of plausible outputs. If any invariant fails, report the exact original tuple, refinement or mutation and resulting outputs so the defect can be reproduced.

E14-X1 Design three semantic mutation witnesses

For Meridian, identify a known tuple that exposes each mutation: reverse the M1 damage test; remove contents_match=true from the M3 guard in a unique-match table; change M4 HOLD to ACCEPT. Give the correct result and defect.

Submit: Three concrete witness records.

15 Deliver the table with a bounded verification record

A decision package should be usable by someone who did not watch its construction. Include the decision question, input dictionary, source procedure identifier, declared matching semantics, rule table, case results, and verification scope. Keep the distinction between a rule and a case visible in filenames and headings. A recipient should not need to infer whether K07 is a source clause, a rule row or a test case.

The handoff should explain unresolved inputs without turning them into rejections. UNRESOLVED means the supplied facts permit more than one outcome under the declared convention. INVALID_INPUT means the representation violates the contract and was not evaluated. HOLD is a legitimate Meridian business label when the earlier conditions pass and no bay is available. These three results are not interchangeable. Collapsing them into "failed" destroys information needed to correct the record or follow the procedure.

A short worked handoff for a different demonstration table might say: "The package contains a unique-match table for two Boolean inputs and all four known combinations. One supplied null case remains unresolved between REVIEW and ACCEPT, and one malformed record was rejected before evaluation. Checks compare each mapping to the stated procedure; they do not verify the truth of the supplied observations or execute any routing action." Use your own verified counts for Meridian rather than copying these demonstration numbers.

Before release, compare every source-clause reference with the manifest and every expected case ID with the submitted results. Check the complete tuple membership, not only row counts. Confirm that the displayed table and runnable code implement the same guards. If code was not run, label the result as a manual or static review. If code was run, preserve the exact command, tested file identities and observed result. Avoid using a source date as an invented execution timestamp.

Finally, keep policy questions separate from conversion defects. An overlap introduced by your translation should be repaired from the source. An ambiguity already present in the source should be reported to the responsible owner rather than silently resolved. A new exception proposed during review is a proposed policy change, not an error correction. This distinction protects the trace from source to table and makes future revisions explainable. A high-quality decision table is compact, but the evidence supporting its compactness must remain available.

E15-X1 Write the capstone release note

Write a concise Meridian handoff naming the known-input scope, six partial cases, four invalid cases, unique-match semantics and limits. Distinguish UNRESOLVED, INVALID_INPUT and HOLD without making a real-world routing action.

Submit: Release note with three result definitions.

Assessment

Choose one best answer per question. Questions 1–10 check the foundation concepts; questions 11–20 apply the extended study units. Complete and submit the practical tasks separately. A strong quiz score does not replace correct practical work.

1. Three independent Boolean conditions with both values allowed create how many known combinations?

A. 9, because there are three conditions squared.

B. 8, because the domain is 2 × 2 × 2.

C. 6, because each condition has two values.

D. 3, because one row per condition is enough.

2. Under the foundation procedure, an incomplete form has stock and approval. Which outcome applies?

A. SEEK_APPROVAL

B. RELEASE

C. WAITLIST

D. REQUEST_INFO

3. A complete request has no stock but has approval. What follows?

A. REQUEST_INFO

B. RELEASE

C. SEEK_APPROVAL

D. WAITLIST

4. A complete request has stock but lacks approval. Which outcome follows?

A. SEEK_APPROVAL

B. RELEASE

C. REQUEST_INFO

D. WAITLIST

5. In a compressed rule, what does dash mean?

A. The input record omitted that field.

B. The source value is unknown.

C. Either allowed value gives the same result within that rule.

D. The condition must be false.

6. The foundation U1 has unknown form status and known stock and approval. What is justified?

A. Drop the form condition from the procedure.

B. Treat the form as complete.

C. Treat the form as incomplete.

D. Flag the missing fact for verification before release.

7. A unique-match table gives two incompatible outcomes for one valid tuple. What is the defect?

A. The table is too detailed.

B. The table has too many input columns.

C. Rules overlap without an allowed resolution.

D. The more specific-looking result should be chosen.

8. Why include source clauses beside guarded rules?

A. To avoid testing the table.

B. To allow intuitive policy changes.

C. To prove every input observation is true.

D. To trace each outcome and inherited guard to the procedure.

9. Which item is an outcome rather than an input condition?

A. WAITLIST

B. stock_available

C. form_complete

D. manager_approval

10. Which check best validates the foundation conversion within its known domain?

A. The RELEASE case alone passes.

B. All eight distinct tuples match direct procedure reading.

C. The table contains all four output names.

D. The output distribution has the expected counts.

11. In an order-independent foundation table, what guard belongs on WAITLIST?

A. not stock_available

B. form_complete and not stock_available

C. not form_complete and not stock_available

D. manager_approval and not stock_available

12. Under the declared completion-set convention, (false,null,null) yields what foundation result?

A. REQUEST_INFO because every completion reaches P1.

B. RELEASE because later values are ignored.

C. UNRESOLVED because every null always blocks a result.

D. INVALID_INPUT because null is forbidden in the expanded convention.

13. A valid partial case has possible outcomes WAITLIST and SEEK_APPROVAL. What may the evaluator do?

A. Choose the more frequent outcome.

B. Choose the alphabetically first outcome.

C. Return UNRESOLVED with both possibilities.

D. Treat null as false to obtain one outcome.

14. A Boolean-only field contains the string "false". Which response follows the strict contract?

A. Treat it as true because it is nonempty.

B. Treat it as false because of its spelling.

C. Reject it as a type error before evaluating.

D. Return the default business outcome.

15. Rules mass <= 5 and mass >= 5 share which problem at 5?

A. An overlap.

B. A gap.

C. An unknown input.

D. A unit conversion failure.

16. A test suite and expected file are regenerated by the same candidate function before comparison. What is the main risk?

A. The tests are too slow.

B. The files cannot be read.

C. The function can agree with its own mistake.

D. The source clauses become invalid.

17. Meridian has damage=true and all later conditions unknown. Under M6, which result follows?

A. ISOLATE, while retaining the unknown-field list.

B. ACCEPT because the unknown fields are wildcards.

C. HOLD until every fact is collected.

D. UNRESOLVED among all five outcomes.

18. Four conditions each allowing false, true and null produce how many partial-domain tuples?

A. 64

B. 16

C. 12

D. 81

19. Replacing one null with a known Boolean while keeping all other facts and rules fixed should do what to the possible-outcome set?

A. Always increase it.

B. Keep it the same or reduce it.

C. Choose the most common outcome.

D. Add a new default outcome.

20. A test report says the table passed all sixteen Meridian Boolean cases. What does that establish?

A. The physical crates were inspected correctly.

B. The tested mappings match expectations within that declared domain.

C. The procedure is optimal for every gallery.

D. All unknown and malformed inputs were also tested.

Sources and further reading

W3C PROV Overview: provenance and derivation
https://www.w3.org/TR/prov-overview/

W3C Model for Tabular Data
https://www.w3.org/TR/tabular-data-model/

OMG Decision Model and Notation 1.5 specification landing page
https://www.omg.org/spec/DMN/1.5/

Python 3.13 itertools.product documentation
https://docs.python.org/3.13/library/itertools.html#itertools.product

Python 3.13 built-in types: truth-value testing
https://docs.python.org/3.13/library/stdtypes.html#truth-value-testing