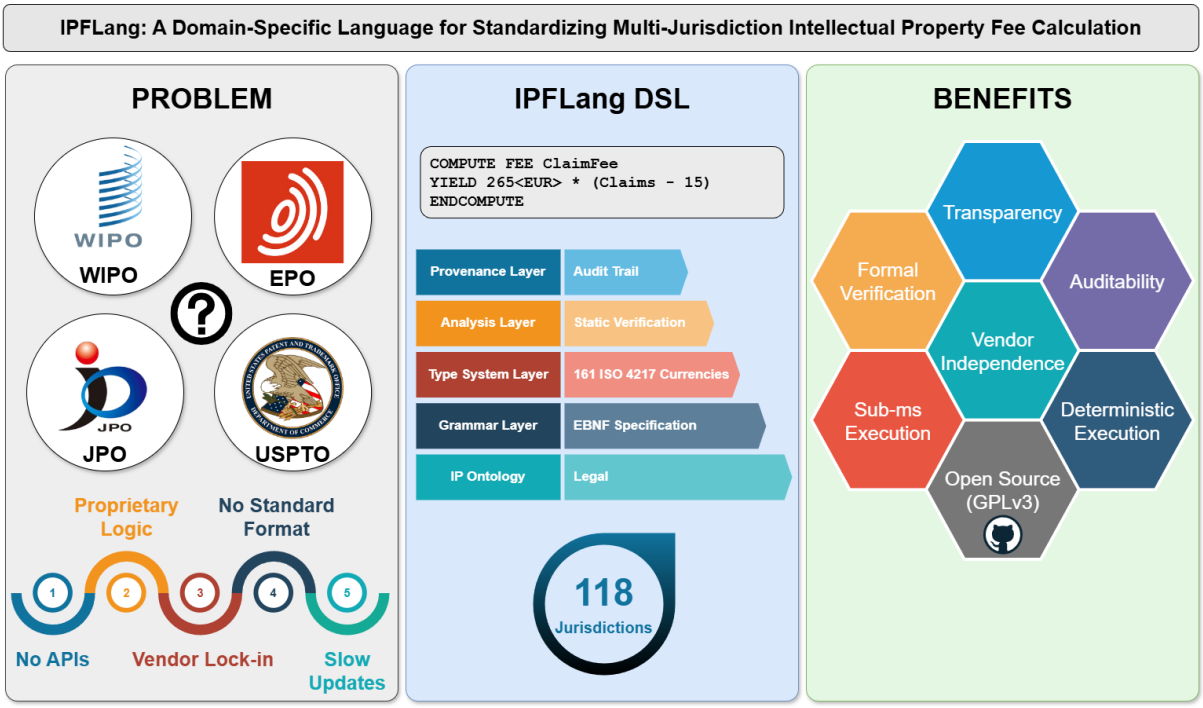
IPFLang: A Domain-Specific Language for Standardizing Multi-Jurisdiction Intellectual Property Fee Calculation

Valer Bocan, PhD, CSSLP
Department of Computer and Information Technology
Politehnica University of Timișoara, 300223 Timișoara, Romania
Email: valer.bocan@upt.ro
ORCID: 0009-0006-9084-4064



Abstract

The intellectual property management industry faces interoperability challenges due to fragmented, proprietary fee calculation implementations across patent offices - no standard exists for encoding or exchanging fee computation rules. This paper presents IPFLang, a domain-specific language specification for multi-jurisdiction fee calculation. IPFLang provides: (1) a formal EBNF grammar with declarative fee computation blocks; (2) a currency-aware type system supporting 161 ISO 4217 currencies that prevents cross-currency arithmetic errors at compile time; (3) static analysis of fee completeness (with formal guarantees for bounded domains) and monotonicity; and (4) provenance tracking for auditability. We present formal typing rules with a type safety argument, analysis algorithms with complexity bounds, and expert validation of production jurisdiction files. The open-source reference implementation includes 118

production jurisdiction files covering PCT national/regional phase entry fees validated by domain experts, a comprehensive test suite with 260 test methods, and sub-millisecond execution. IPFLang establishes a foundation for regulatory calculation standardization, enabling vendor-independent fee computation across patent offices.

Keywords

domain-specific language, intellectual property, standardization, interoperability, type systems, static verification, regulatory automation, formal specification

Highlights

- Formal DSL specification addressing the IP fee calculation standardization gap
- Currency-aware type system with 161 ISO 4217 currencies preventing cross-currency errors statically
- Static analysis of fee completeness (formal guarantees for bounded domains) and monotonicity
- Provenance tracking with counterfactual analysis for auditability
- Expert validation of 118 production jurisdiction files covering PCT national/regional phase entry fees
- Open-source reference implementation enabling vendor-independent fee computation

1. Introduction

1.1 The IP Technology Fragmentation Problem

The global intellectual property management ecosystem operates through a complex network of national and regional patent offices, each maintaining independent fee calculation systems with proprietary interfaces. Major offices such as the United States Patent and Trademark Office (USPTO), European Patent Office (EPO), Japan Patent Office (JPO), and World Intellectual Property Organization (WIPO) each provide web-based fee calculators [1-4], yet these systems exhibit fundamental interoperability deficiencies that impede efficient IP management workflows.

The first and perhaps most pressing issue is the absence of any standard data format across jurisdictions. Each patent office defines fee parameters using custom terminology, requiring manual interpretation and data entry for each calculation. Where the USPTO uses “entity type” classifications, the EPO employs “applicant category” with entirely different discount structures. WIPO PCT applications introduce additional complexity by requiring separate parameters for International Searching Authority selection, compounding the burden for practitioners handling multi-jurisdiction filings.

Equally problematic is the complete absence of programmatic interfaces for fee calculation. Government calculators operate exclusively through web browsers with no API access, preventing any form of automation or integration with IP management

workflows. Patent offices have invested significantly in digital transformation for application filing through systems like ePCT, EFS-Web, and Online Filing, yet they have not extended programmatic access to fee calculation services, creating a gap in their digital offerings.

The situation is further complicated by the proprietary nature of calculation logic itself. Fee computation rules remain embedded in server-side code, entirely inaccessible to practitioners who must verify accuracy against frequently updated official fee schedules. When discrepancies arise between calculated and actual fees, practitioners find themselves unable to diagnose whether errors stem from incorrect inputs or flawed calculation logic, as the underlying implementation remains opaque.

Commercial IP management platforms from vendors such as CPA Global, Anaqua, and PatSnap address some workflow needs but perpetuate fragmentation through vendor-specific implementations, limited jurisdiction coverage, and hardcoded fee structures requiring vendor patches for regulatory updates [5]. The global IP management software market operates without standardized interfaces, forcing enterprises into vendor lock-in and limiting interoperability between best-of-breed solutions. Research on de facto technical standards has documented how network effects and single-vendor control in IT ecosystems create barriers to interoperability and innovation [18], patterns that characterize the current IP technology landscape.

1.2 The Need for Standardization

Successful technology domains achieve interoperability through open standards. SQL standardized database queries through ISO/IEC 9075, HTML standardized web content through W3C specifications, and XML Schema standardized data validation under the same organization. These standards enabled ecosystem growth by separating interface specifications from implementations, allowing multiple vendors to provide interoperable solutions. Research on intellectual property disclosure in standards development has examined how standards organizations balance IP rights with interoperability requirements [19], highlighting the complex dynamics that any IP technology standardization effort must navigate.

The IP technology domain lacks equivalent standards for fee calculation, resulting in three critical gaps. The first is a specification gap: no standard language exists for expressing jurisdiction-specific fee rules. LegalRuleML [6] addresses compliance checking but lacks arithmetic expressiveness for financial calculations. Catala [7] demonstrates sophisticated tax calculations but targets single-jurisdiction applications requiring formal methods expertise unsuitable for legal practitioners. The second gap concerns formal verification: existing calculators provide no guarantees that fee definitions cover all valid input combinations or behave predictably as inputs change. The third gap is one of transparency: proprietary implementations prevent independent verification of calculation correctness.

1.3 Research Questions and Contributions

This work addresses four fundamental questions regarding standardization of IP fee calculations.

The first research question concerns language design: can a domain-specific language provide sufficient expressiveness for complex regulatory fee structures while employing syntax designed for readability by domain experts? The second addresses formal correctness: what static guarantees can a type system and verification framework provide for multi-currency regulatory calculations? The third focuses on verification: how can completeness and monotonicity properties be statically verified to ensure fee definitions behave correctly? The fourth explores practical feasibility: can a DSL-based approach achieve acceptable performance for production use?

This paper presents IPFLang (Intellectual Property Fees Language), a domain-specific language specification for multi-jurisdiction fee calculations, with five principal contributions:

1. **Language Specification** (Section 3): Formal definition of IPFLang syntax using EBNF grammar, with declarative fee computation blocks, explicit input type declarations including a currency-aware AMOUNT type, temporal operators for date-dependent calculations, version management with effective dates, and jurisdiction composition for code reuse.

2. **Type System and Static Verification** (Section 4): A currency-aware type system supporting 161 ISO 4217 currencies with formal typing rules that prevent cross-currency arithmetic errors at compile time, along with verification algorithms for completeness and monotonicity with complexity analysis.

3. **Provenance and Auditability** (Section 5): Execution tracing showing how final amounts derive from input parameters, with counterfactual analysis enabling what-if scenarios for regulatory compliance.

4. **Reference Implementation** (Section 6): An open-source command-line tool released under GPLv3 demonstrating IPFLang execution with 260 test methods validating type safety and verification correctness.

5. **Validation** (Section 7): Expert validation of 118 production jurisdiction files covering PCT national/regional phase entry fees for major patent offices worldwide.

1.4 Paper Organization

Section 2 surveys related work in IP data standards, legal domain DSLs, and regulatory automation. Section 3 presents the complete IPFLang specification including formal grammar and input type system. Section 4 details the currency-aware type system with formal typing rules and static verification algorithms. Section 5 describes provenance tracking and counterfactual analysis. Section 6 presents the reference implementation architecture. Section 7 provides evaluation including expert validation of production jurisdiction files and threats to validity. Section 8 discusses advantages of DSL-based standardization, cross-domain applicability, and limitations. Section 9 concludes with contributions summary, impact assessment, and future directions.

2. Related Work and Standards Landscape

2.1 Existing IP Data Standards

The IP technology domain has achieved partial standardization in data exchange but lacks standards for computational tasks like fee calculation.

WIPO ST.96, commonly known as Patent XML, represents the World Intellectual Property Organization's standard for patent application data exchange [8]. The standard defines XML schemas covering bibliographic data, descriptions, claims, and drawings, but it explicitly excludes financial calculations from its scope. While the standard provides a foundation for data interoperability, fee calculations fall entirely outside its purview.

The European Patent Office provides Open Patent Services (OPS), which offers RESTful APIs for patent search and retrieval [9]. OPS provides family information, legal status, and bibliographic data, but omits fee calculation endpoints entirely. The existence of OPS demonstrates that patent offices can successfully deploy programmatic interfaces, suggesting that fee calculation APIs are technically feasible but simply have not been prioritized.

The gap becomes clear upon examination: IP data standards focus on informational exchange such as bibliographic data, legal status, and full-text search, while omitting computational tasks entirely. Fee calculation remains manual, preventing end-to-end workflow automation.

2.2 Domain-Specific Languages for Legal Domains

Academic research in computational law has produced several DSLs for legal rules, yet none address regulatory fee calculations with multi-currency and multi-jurisdiction requirements.

LegalRuleML, developed by Athan et al. [6], provides an XML-based specification language for legal rules with ontology-based reasoning. The language excels at representing deontic logic covering obligations, permissions, and prohibitions, and supports defeasibility for handling rule precedence. However, LegalRuleML emphasizes binary compliance checking (compliant or non-compliant) with minimal arithmetic support. Complex fee formulas involving thresholds, progressions, and conditional multipliers exceed the language's design scope. The XML syntax also presents accessibility challenges for legal professionals without technical training.

Catala, created by Merigoux et al. [7], represents a programming language specifically designed for tax law computation. The language demonstrates sophisticated financial calculations with formal verification guarantees through dependent type theory. However, Catala targets single-jurisdiction applications, primarily the French tax code, and requires formal methods expertise that limits adoption by legal practitioners. IPFLang and Catala represent complementary approaches: Catala employs dependent types for exhaustive case coverage targeting formal verification experts, while IPFLang prioritizes comprehensibility through keyword-based syntax (EQ, GT, AND rather than symbols) and domain-specific primitives (currency literals, temporal operators)

designed for IP practitioners to read and modify directly. While Catala's dependent types provide stronger theoretical guarantees, IPFLang's simpler type system (currency-parameterized amounts without dependent types) suffices for IP fee calculations where amounts are always non-negative and conditions are finite Boolean combinations.

Contract-oriented DSLs [10] focus on party obligations, temporal constraints, and conditional execution semantics. Monetary aspects receive minimal treatment, with basic arithmetic operations but lacking multi-currency precision, exchange rate management, and historical rate tracking required for cross-border IP portfolios.

The concept of encoding units in type systems originates with Kennedy's dimensional types [11], which prevent unit mismatch errors in scientific computing. IPFLang applies similar principles to currency, extending the concept with explicit conversion operators and polymorphic type variables for generic fee definitions. Recent work on graded modal types [20] demonstrates how type systems can track quantitative resource usage, providing theoretical foundations for systems that reason about resource consumption - a concept related to IPFLang's tracking of monetary values across fee computations.

OpenFisca [12] provides a Python-based platform for tax-benefit microsimulation, used by governments including France and New Zealand. While powerful, OpenFisca requires Python programming expertise and targets general fiscal policy rather than the specific requirements of IP fee calculation.

Existing legal DSLs address contract execution, compliance checking, or single-jurisdiction calculations, but none provide the combination of arithmetic expressiveness for complex fee formulas, multi-currency support with type safety, static verification of completeness and monotonicity, and multi-jurisdiction portability that IP fee calculation demands.

2.3 Regulatory Automation and Rules Engines

Automated compliance checking represents a related domain where technology assists regulatory interpretation.

Business Rules Management Systems such as Drools [13] provide general-purpose rules engines using production rules with Rete algorithm inference. These systems require substantial technical expertise, lack domain-specific abstractions for legal concepts, and impose expensive enterprise licensing. While powerful, BRMS platforms are fundamentally over-engineered for deterministic fee calculations.

Some governments have begun pursuing rules-as-code initiatives that encode regulations in executable formats [14, 15]. New Zealand's "Better Rules" program and similar initiatives in Australia and France explore machine-consumable legislation. The OECD has documented the international scope of these efforts, providing frameworks for encoding rules in machine-readable formats that emphasize transparency and regulatory automation [21]. More recent policy analysis explores how rules-as-code approaches can enable more efficient global economic governance, including applications to international trade regulations and intellectual property [22]. These

initiatives typically use general-purpose languages rather than domain-specific languages, limiting accessibility to legal experts who must rely on programmers for implementation.

Table 1 provides a systematic comparison of IPFLang with related approaches.

Feature	IPFLang	Catala	LegalRuleML	Drools	OpenFisca
Primary Domain	IP fees	Tax law	Compliance	General	Tax-benefit
Type Safety	Currency-aware	Dependent types	None	None	Runtime
Multi-currency	Built-in	User-definable	No	No	Limited
Static Verification	Completeness, Monotonicity	Exhaustiveness	Defeasibility	None	None
Syntax Style	Declarative keywords	Literate	XML	Drools DRL	Python
Target User	Domain experts	Formal methods experts	Ontology engineers	Developers	Developers
Multi-jurisdiction	Composition	No	No	No	Yes
Interoperability Focus	Multi-jurisdiction	Single jurisdiction	Ontology mapping	Integration APIs	Country modules
Provenance	Built-in	No	No	Audit log	No
License	GPLv3	Apache 2.0	OASIS	Apache 2.0	AGPL

Table 1- Comparison of legal DSLs and rules engines

IPFLang differentiates itself through the combination of domain-specific syntax designed for readability, first-class multi-currency type safety with all 161 ISO 4217 currencies as built-in primitives, static verification of completeness and monotonicity, and explicit support for multi-jurisdiction fee structures with inheritance and composition. While languages like Catala could theoretically support currency types through user-defined abstractions, IPFLang provides these as language primitives requiring no additional implementation effort. The design follows established principles for DSL development that emphasize matching language constructs to domain concepts [23].

3. IPFLang Language Specification

3.1 Design Principles

IPFLang design balances six competing objectives: expressiveness for complex regulatory logic, readability through domain-specific syntax designed for legal professionals, deterministic execution for financial accuracy, extensibility for evolving regulations, and formal verification potential.

The first principle is declarative semantics. Fee calculations specify what to compute rather than how to compute it. Declarative approaches facilitate correctness verification by legal experts who can validate fee formulas directly against official schedules without needing to understand imperative control flow.

The second principle requires explicit semantics. Operators use keyword syntax such as EQ, GT, AND, and OR rather than symbols like ==, >, &&, and ||. This design choice aims to improve readability for users with varying technical backgrounds, aligning with DSL design guidelines emphasizing domain-specific notation [16]. Empirical validation of this readability hypothesis remains future work (see Section 8.3).

The third principle establishes a static type system. Input declarations explicitly specify types including NUMBER, LIST, MULTILIST, BOOLEAN, DATE, and AMOUNT, enabling compile-time validation. Static typing prevents runtime errors from type mismatches and provides clear parameter documentation.

The fourth principle maintains minimal syntax complexity. Language features target the minimum necessary for regulatory fee calculations. Structured blocks such as DEFINE...ENDDEFINE and COMPUTE...ENDCOMPUTE with explicit terminators aid comprehension and prevent syntax errors common in expression-based languages.

The fifth principle ensures auditability and traceability. Fee computation produces step-by-step execution traces showing how final amounts derive from input parameters. This addresses legal requirements for calculation transparency and assists in dispute resolution.

The sixth principle guarantees jurisdiction independence. Language constructs make no assumptions about specific jurisdictions, enabling code reuse across patent offices. Jurisdiction-specific business rules reside in fee definitions rather than language syntax.

3.2 Language Syntax Overview

An IPFLang program begins with an optional version declaration that establishes metadata including version identifier, effective date, and references to authoritative fee schedules. Following the version declaration, group definitions organize inputs into logical categories for user interface presentation, with weights determining display order. The core of any IPFLang program consists of input definitions that declare the parameters required for fee calculation - these may be single-choice enumerations (LIST), multi-choice selections (MULTILIST), numeric values with optional constraints (NUMBER), binary choices (BOOLEAN), temporal values (DATE), or currency-aware monetary inputs (AMOUNT). Fee computation blocks contain the actual calculation logic using conditional yields and case statements. Programs may include verification directives that enable static analysis of completeness and monotonicity properties. Finally, return statements provide named outputs for integration with external systems.

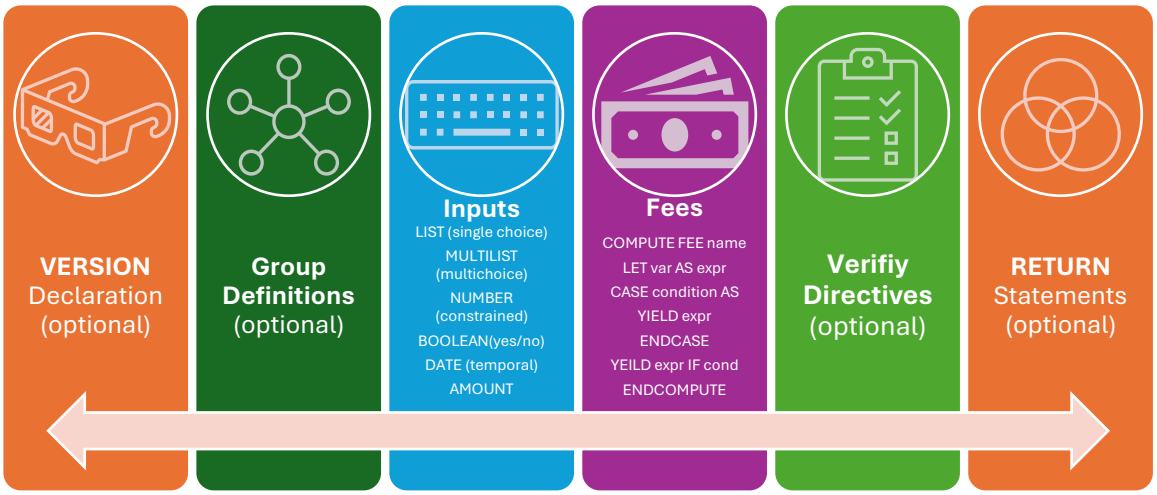


Figure 1 – DSL overview

3.3 Version Declaration

Fee schedules change frequently, requiring version tracking with effective dates.

Syntax:

```
VERSION '<VersionId>' EFFECTIVE <date> [DESCRIPTION '<description>'] [REFE
RENCE '<reference>']
```

Example:

```
VERSION '2024.1' EFFECTIVE 01.04.2024 DESCRIPTION 'EPO official fees 2024'
 REFERENCE 'EPO Official Journal 2024/03'
```

The VERSION directive enables temporal queries (calculating fees as they were on a specific date), version comparison (identifying changes between fee schedule revisions), and regulatory traceability (linking calculations to authoritative sources).

3.4 Input Type System

IPFLang provides six input types matching common parameter patterns across patent offices.

3.4.1 LIST (Single-Choice Enumeration)

The LIST type handles parameters with mutually exclusive options such as entity type, filing basis, or examination request.

```
DEFINE LIST EntityType AS 'Select entity size'
GROUP General
CHOICE NormalEntity AS 'Large Entity'
CHOICE SmallEntity AS 'Small Entity (50% discount)'
CHOICE MicroEntity AS 'Micro Entity (75% discount)'
DEFAULT NormalEntity
ENDDEFINE
```

3.4.2 MULTILIST (Multi-Choice Enumeration)

The MULTILIST type accommodates parameters allowing multiple selections. The special property accessor `!COUNT` returns the number of selections. IPFLang uses the exclamation mark (`!`) as the property accessor to distinguish property access from other syntactic elements and to provide visual emphasis for these computed properties, which derive values from the underlying data rather than representing stored values directly.

```
DEFINE MULTILIST Countries AS 'Select validation countries'
CHOICE VAL_DE AS 'Germany'
CHOICE VAL_FR AS 'France'
CHOICE VAL_GB AS 'United Kingdom'
DEFAULT VAL_DE,VAL_FR
ENDDEFINE
```

Usage: `YIELD 100 * Countries!COUNT`

3.4.3 NUMBER (Numeric Input)

The NUMBER type handles counts, quantities, and page numbers with optional constraints.

```
DEFINE NUMBER ClaimCount AS 'Number of claims in application'
BETWEEN 1 AND 500
DEFAULT 10
ENDDEFINE
```

3.4.4 BOOLEAN (Yes/No)

The BOOLEAN type represents binary choices.

```
DEFINE BOOLEAN RequestExamination AS 'Request substantive examination?'
DEFAULT TRUE
ENDDEFINE
```

3.4.5 DATE (Date Input)

The DATE type handles filing dates and priority dates for calculating time-dependent fees. Temporal properties enable calculations based on elapsed time:

- `!YEARSTONOW` - Years from date to current date
- `!MONTHSTONOW` - Months from date to current date
- `!DAYSTONOW` - Days from date to current date
- `!MONTHSTONOW_FROMLASTDAY` - Months from end of date's month

```
DEFINE DATE FilingDate AS 'Application filing date'
BETWEEN 01.01.2000 AND TODAY
DEFAULT TODAY
ENDDEFINE
```

Usage: `LET YearsSinceFiling AS FilingDate!YEARSTONOW`

3.4.6 AMOUNT (Currency-Aware Monetary Input)

The AMOUNT type represents monetary values with an associated ISO 4217 currency code, enabling type-safe arithmetic.

```
DEFINE AMOUNT PriorSearchFee AS 'Prior art search fee paid'
CURRENCY EUR
DEFAULT 0
ENDDEFINE
```

The AMOUNT type integrates with the currency-aware type system described in Section 4, preventing accidental cross-currency arithmetic.

3.5 Group Definitions

Groups organize inputs for user interface presentation, with weights determining display order.

```
DEFINE GROUP General AS 'General Information' WITH WEIGHT 10
DEFINE GROUP Claims AS 'Claims Information' WITH WEIGHT 20
DEFINE GROUP Options AS 'Fee Options' WITH WEIGHT 30
```

3.6 Fee Computation Blocks

Fee calculations use structured COMPUTE FEE blocks with conditional YIELD statements.

Basic syntax:

```
COMPUTE FEE <fee_name> [OPTIONAL]
  [LET <variable> AS <expression>]*
  [CASE <condition> AS
    YIELD <expression> [IF <condition>]
  ENDCASE]*
  YIELD <expression> [IF <condition>]
ENDCOMPUTE
```

The OPTIONAL keyword distinguishes fees that may or may not apply from mandatory fees.

Example - EPO claim fees with progressive rates:

```
COMPUTE FEE ExcessClaimsFee
LET ClaimFee1 AS 265<EUR>
LET ClaimFee2 AS 660<EUR>
CASE ClaimCount LTE 15 AS
  YIELD 0<EUR>
ENDCASE
CASE ClaimCount GT 15 AND ClaimCount LTE 50 AS
  YIELD ClaimFee1 * (ClaimCount - 15)
ENDCASE
CASE ClaimCount GT 50 AS
  YIELD ClaimFee1 * 35 + ClaimFee2 * (ClaimCount - 50)
ENDCASE
ENDCOMPUTE
```

This directly encodes the EPO's fee schedule: EUR 265 per claim for claims 16-50 and EUR 660 per claim beyond 50.

3.7 Currency Literals

Numeric values can be annotated with ISO 4217 currency codes:

```
100<EUR>      # 100 Euros
50.50<USD>    # 50.50 US Dollars
1000<JPY>     # 1000 Japanese Yen
```

The type system enforces currency compatibility at compile time, as detailed in Section 4.

3.8 Operators and Expressions

IPFLang provides:

Comparison operators: EQ (equality), NEQ (inequality), GT (greater than), LT (less than), GTE (greater or equal), LTE (less or equal)

Logical operators: AND, OR (with short-circuit evaluation)

Arithmetic operators: +, -, *, /, with ROUND, FLOOR, CEIL functions

Set operators for MULTILIST: IN (membership), NIN (non-membership), !COUNT (cardinality)

Operator precedence (highest to lowest): ROUND/FLOOR/CEIL, then *, /, then +, -, then comparisons, then AND, then OR. Parentheses override default precedence.

3.9 Jurisdiction Composition

IPFLang supports inheritance for code reuse across related fee schedules. A child jurisdiction inherits inputs and fees from a parent, can add new definitions, and can override inherited fees.

Parent (EPO base):

```
COMPUTE FEE FilingFee
YIELD 135<EUR>
ENDCOMPUTE
```

Child (EPO Germany national phase):

```
# Inherits FilingFee from parent
# Adds Germany-specific fee
COMPUTE FEE GermanTranslationFee
YIELD 1050<EUR> IF NeedsTranslation EQ TRUE
YIELD 0<EUR> IF NeedsTranslation EQ FALSE
ENDCOMPUTE
```

Composition enables significant code reuse between related jurisdictions, simplifies maintenance when base fees change, and maintains clear traceability of fee origins. Jurisdiction composition is implemented at the tool level rather than as a language

construct. The `ipflang compose` command merges multiple IPFLang files, with the resulting combined program conforming to the grammar specification.

3.10 Formal Grammar (EBNF)

The complete formal grammar in Extended Backus-Naur Form:

```
(* Program Structure *)
<program>              ::= <comment>* <version>? <group>* <input_definition>*
                           <fee_computation>+ <verification>* <return>*

(* Comments *)
<comment>              ::= "#" {<any_char> - <newline>} <newline>

(* Version Declaration *)
<version>              ::= "VERSION" <string> "EFFECTIVE" <date>
                           ["DESCRIPTION" <string>] ["REFERENCE" <string>]

(* Group Definition *)
<group>                ::= "DEFINE" "GROUP" <identifier> "AS" <string>
                           "WITH" "WEIGHT" <number>

(* Input Definitions *)
<input_definition>     ::= "DEFINE" <input_type> <identifier> "AS" <string>
                           <type_specifics> ["DEFAULT" <default_value>]
                           ["GROUP" <identifier>] "ENDDEFINE"
<input_type>           ::= "LIST" | "MULTILIST" | "NUMBER" | "BOOLEAN" | "DATE" | "AMOUNT"
<type_specifics>       ::= <choices> | <numeric_constraint> | <date_constraint>
                         | <currency_spec> | ε
<choices>              ::= <choice>+
<choice>               ::= "CHOICE" <identifier> "AS" <string>
<numeric_constraint>   ::= "BETWEEN" <number> "AND" <number>
<date_constraint>      ::= "BETWEEN" <date> "AND" (<date> | "TODAY")
<currency_spec>        ::= "CURRENCY" <currency_code>
<default_value>        ::= <number> | <identifier> | <boolean_literal> | <date>
                         | <currency_literal> | <identifier_list>
<identifier_list>      ::= <identifier> ("," <identifier>)*
<boolean_literal>      ::= "TRUE" | "FALSE"

(* Fee Computation *)
<fee_computation>      ::= "COMPUTE" "FEE" <identifier> [<type_params>]
                           ["RETURN" <currency_code>] ["OPTIONAL"]
                           <let_statement>* <case_or_yield>*
                           "ENDCOMPUTE"
<type_params>          ::= "<" <type_variable> ">"
<type_variable>        ::= <upper_letter>
<let_statement>        ::= "LET" <identifier> "AS" <expression>
<case_or_yield>        ::= <case_block> | <yield_statement>
<case_block>           ::= "CASE" <condition> "AS" <yield_statement>+ "ENDCASE"
<yield_statement>      ::= "YIELD" <expression> ["IF" <condition>]

(* Verification Directives *)
<verification>         ::= <verify_complete> | <verify_monotonic>
<verify_complete>      ::= "VERIFY" "COMPLETE" "FEE" <identifier>
<verify_monotonic>     ::= "VERIFY" "MONOTONIC" "FEE" <identifier>
                           "WITH" "RESPECT" "TO" <identifier>
                           ["DIRECTION" <direction>]
<direction>            ::= "NonDecreasing" | "NonIncreasing"
                         | "StrictlyIncreasing" | "StrictlyDecreasing"

(* Return Statement *)
<return>               ::= "RETURN" <identifier> "AS" <string>
```

```
(* Expressions *)
<expression>           ::= <term> (("+" | "-") <term>)*
<term>                 ::= <factor> (("*" | "/") <factor>)*
<factor>               ::= <number> | <currency_literal> | <identifier>
                         | <property_access> | <function_call> | "(" <expression> ")"
<currency_literal>     ::= <number> "<" (<currency_code> | <type_variable>) ">"
<property_access>      ::= <identifier> "!" <property_name>
<property_name>        ::= "COUNT" | "YEARSTONOW" | "MONTHSTONOW" | "DAYSTONOW"
                         | "MONTHSTONOW_FROMLASTDAY"
<function_call>        ::= ("ROUND" | "FLOOR" | "CEIL") "(" <expression> ")"
                         | "CONVERT" "(" <expression> "," <currency_code> "," <currency_cod
e> ")"

(* Conditions *)
<condition>            ::= <or_condition>
<or_condition>         ::= <and_condition> ("OR" <and_condition>)*
<and_condition>        ::= <primary_condition> ("AND" <primary_condition>)*
<primary_condition>    ::= <expression> <comparison_op> <expression>
                         | <identifier> ("IN" | "NIN") <identifier>
                         | <identifier>   (* Boolean variable reference *)
                         | "(" <condition> ")"

<comparison_op>        ::= "EQ" | "NEQ" | "GT" | "LT" | "GTE" | "LTE"

(* Lexical Elements *)
<identifier>           ::= <letter> (<letter> | <digit> | "_")*
<number>               ::= <digit>+ ("." <digit>+)?
<string>               ::= "'" {<any_char> - "'"} "'"
<currency_code>        ::= <upper_letter> <upper_letter> <upper_letter>
<date>                 ::= <day> "." <month> "." <year> | "TODAY"
<day>                  ::= <digit> <digit>?
<month>                ::= <digit> <digit>?
<year>                 ::= <digit> <digit> <digit> <digit>
<letter>               ::= "A" | ... | "Z" | "a" | ... | "z"
<upper_letter>         ::= "A" | ... | "Z"
<digit>                ::= "0" | ... | "9"
```

A bare `<identifier>` in a condition context must reference a BOOLEAN input variable; it evaluates to that variable's Boolean value.

4. Type System and Static Verification

4.1 Currency-Aware Type System

IPFLang employs a dimensional type system preventing cross-currency arithmetic errors at compile time, analogous to units-of-measure checking in scientific computing [11]. The system supports all 161 ISO 4217 currency codes. The approach to currency as a type parameter draws on similar patterns in language standards for monetary computation, such as JSR-354 (Java Money and Currency API) [24], which defines standard interfaces for representing and manipulating monetary amounts in type-safe ways.

4.1.1 Type Language

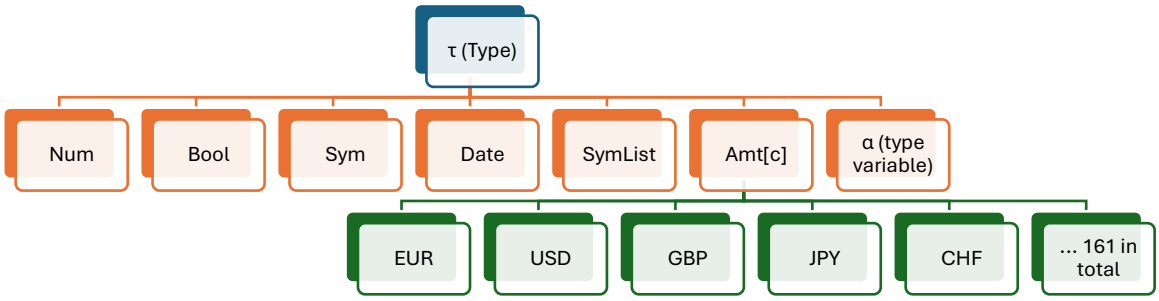


Figure 2 – IPFLang type hierarchy and currency typing rules

Figure 2 shows the type language which extends basic types with currency-parameterized amounts:

```
τ ::= Num       -- dimensionless numbers
    | Bool      -- Boolean values
    | Sym       -- symbolic identifiers (LIST choices)
    | Date      -- date values
    | SymList   -- symbol lists (MULTILIST selections)
    | Amt[c]    -- monetary amounts, c ∈ ISO-4217
    | α         -- type variables (for polymorphic fees)
σ :: = τ | ∀α. σ-- type schemes (polymorphic types)
```

where:

- `Num` represents dimensionless numbers used for counts and arithmetic
- `Bool` represents Boolean values TRUE and FALSE
- `Sym` represents symbolic identifiers from LIST input choices
- `Date` represents date values with temporal properties
- `SymList` represents symbol lists from MULTILIST input selections
- `Amt[c]` represents monetary amounts denominated in currency `c`, where c ∈ ISO-4217
- α represents currency type variables for polymorphic fee definitions
- ∀α. σ represents universally quantified type schemes

Currency type variables range over ISO-4217 currency codes. In the typing context Γ, the notation `α : Currency` indicates that α is bound as a currency type variable.

4.1.3 Typing Rules

Let Γ denote a typing environment mapping identifiers to types, written Γ(x) = τ. We define the typing judgment Γ ⊢ e : τ, meaning *under environment Γ, expression e has type τ.*

Literals and Variables

The foundation of the type system rests on three base rules that assign types to the simplest expressions in the language. These rules handle constants (both numeric and currency-annotated) and variable references, forming the leaves of every expression tree that the type checker analyzes.

Rule **T-NUM** states that any bare numeric literal - such as 42, 3.14, or 0 - receives type Num, representing a dimensionless number without currency association. Such values are used for counts (number of claims, pages, or designated countries), ratios, and multipliers throughout fee calculations.

Rule **T-VAR** is the standard variable lookup rule found in all typed languages: when an identifier x appears in an expression, its type is determined by consulting the typing environment Γ (gamma), which records the declared types of all variables currently in scope. If the environment contains the binding x : τ, then the expression x has type τ.

Rule **T-CURR** handles currency-annotated literals such as 100<EUR> or 50.50<USD>. The rule first verifies that the currency code c is a valid ISO-4217 identifier (one of 161 recognized codes), then assigns the expression the type Amt[c] - read as "amount denominated in currency c." This explicit annotation is the origin of all currency type information in a program; every currency amount ultimately traces back to either a currency literal or an AMOUNT input declaration.

$$[\text{T-NUM}]\ \frac{}{\Gamma \vdash n : \text{Num}}$$

$$[\text{T-VAR}]\ \frac{\Gamma(x)=\tau}{\Gamma \vdash x : \tau}$$

$$[\text{T-CURR}]\ \frac{c \in \text{ISO}-4217}{\Gamma \vdash n\langle c\rangle : \text{Amt}[c]}$$

Arithmetic Operations

Arithmetic operations form the computational core of fee calculations, and the type system enforces strict currency discipline to prevent nonsensical cross-currency computations. The rules partition into four groups - addition, subtraction, multiplication, and division - each with distinct typing behavior reflecting the dimensional semantics of currency amounts.

Addition. Two distinct rules govern addition, reflecting the fundamental distinction between dimensionless numbers and currency amounts.

Rule **T-ADD-NUM** permits adding two dimensionless numbers, yielding a dimensionless result. This covers operations like summing claim counts or computing page totals where no currency is involved.

Rule **T-ADD-AMT** is more restrictive and embodies the core currency safety guarantee: it permits adding two monetary amounts *only when both operands share the identical currency tag* c. The expression 100<EUR> + 50<EUR> is well-typed with result type Amt[EUR], but 100<EUR> + 50<USD> fails type-checking because no rule permits adding amounts with different currency tags. The type system provides no implicit currency conversion - all conversions must be explicit via the CONVERT function.

Notably absent is any rule permitting Amt[c] + Num; the expression 100<EUR> + 50 is rejected because adding a dimensionless quantity to a currency amount introduces ambiguity about the intended currency of the dimensionless operand.

$$\text{[T-ADD-NUM]}\ \frac{\Gamma \vdash e1 : \text{Num} \quad \Gamma \vdash e2 : \text{Num}}{\Gamma \vdash e1 + e2 : \text{Num}}$$

$$\text{[T-ADD-AMT]}\ \frac{\Gamma \vdash e1 : \text{Amt}[c] \quad \Gamma \vdash e2 : \text{Amt}[c]}{\Gamma \vdash e1 + e2 : \text{Amt}[c]}$$

Multiplication. Multiplication exhibits different behavior from addition because scalar multiplication is dimensionally sound - multiplying a currency amount by a dimensionless count produces another currency amount, not a dimensional mismatch.

Rule **T-MUL-NUM** handles multiplication of two dimensionless numbers, yielding a dimensionless result. This supports calculations like computing the product of claim count and page count.

Rules **T-MUL-SCALAR-R** and **T-MUL-SCALAR-L** together establish that multiplying a currency amount by a dimensionless scalar - in either order - preserves the currency tag. Both 100<EUR> * 2 and 2 * 100<EUR> have type Amt[EUR]. The provision of both left and right variants ensures that scalar multiplication is commutative at the type level, matching mathematical intuition and enabling natural fee expressions like ClaimFee * ExcessClaimCount where the fee is currency-denominated and the count is dimensionless. The two rules are semantically equivalent but syntactically necessary because inference rule premises distinguish operand positions.

$$\text{[T-MUL-NUM]}\ \frac{\Gamma \vdash e1 : \text{Num} \quad \Gamma \vdash e2 : \text{Num}}{\Gamma \vdash e1 * e2 : \text{Num}}$$

$$\text{[T-MUL-SCALAR-R]}\ \frac{\Gamma \vdash e1 : \text{Amt}[c] \quad \Gamma \vdash e2 : \text{Num}}{\Gamma \vdash e1 \times e2 : \text{Amt}[c]}$$

$$\text{[T-MUL-SCALAR-L]}\ \frac{\Gamma \vdash e1 : \text{Num} \quad \Gamma \vdash e2 : \text{Amt}[c]}{\Gamma \vdash e1 \times e2 : \text{Amt}[c]}$$

Subtraction. The subtraction rules mirror addition exactly. Rule **T-SUB-NUM** allows subtracting dimensionless numbers, while **T-SUB-AMT** permits subtracting monetary amounts only when currencies match. The expression 500<EUR> - 100<EUR> yields

Amt[EUR], enabling calculations such as computing a net fee after deducting a prior payment.

As with addition, mixed-currency subtraction (100<EUR> - 50<USD>) and subtraction between dimensioned and dimensionless values (100<EUR> - 50) are both prohibited. This consistent treatment ensures that the type system's guarantees apply uniformly across all additive operations.

$$\text{[T-SUB-NUM]}\ \frac{\Gamma \vdash e1 : \mathrm{Num} \quad \Gamma \vdash e2 : \mathrm{Num}}{\Gamma \vdash e1 - e2 : \mathrm{Num}}$$

$$\text{[T-SUB-AMT]}\ \frac{\Gamma \vdash e1 : \mathrm{Amt}[c] \quad \Gamma \vdash e2 : \mathrm{Amt}[c]}{\Gamma \vdash e1 - e2 : \mathrm{Amt}[c]}$$

Division. Division follows principles similar to multiplication. Rule **T-DIV-NUM** permits dividing one dimensionless number by another, yielding a dimensionless result - useful for computing ratios or averages.

Rule **T-DIV-AMT-SCALAR** allows dividing a currency amount by a dimensionless scalar while preserving the currency tag; for example, 100<EUR> / 2 has type Amt[EUR]. This enables calculations such as splitting fees among multiple applicants, computing per-unit costs, or applying percentage-based reductions.

Note the asymmetry: while Amt[c] / Num is well-typed (yielding Amt[c]), the reverse Num / Amt[c] has no typing rule and is therefore rejected. Dividing a dimensionless number by a currency amount would yield a nonsensical "inverse currency" type with no practical meaning in fee calculations.

$$\text{[T-DIV-NUM]}\ \frac{\Gamma \vdash e1 : \mathrm{Num} \quad \Gamma \vdash e2 : \mathrm{Num}}{\Gamma \vdash e1 / e2 : \mathrm{Num}}$$

$$\text{[T-DIV-AMT-SCALAR]}\ \frac{\Gamma \vdash e1 : \mathrm{Amt}[c] \quad \Gamma \vdash e2 : \mathrm{Num}}{\Gamma \vdash e1 / e2 : \mathrm{Amt}[c]}$$

Currency Conversion

The CONVERT function provides the sole mechanism for changing currency tags, making all cross-currency operations explicit and auditable in the source code. Without this function, a well-typed program cannot transform an amount from one currency to another - this is by design, ensuring that currency conversions are deliberate, visible, and traceable.

Rule **T-CONVERT** requires four premises to be satisfied:

1. The expression e must have type Amt[c] for some currency c;
2. The declared source currency c_1 in the function call must exactly equal the inferred currency c of the expression;
3. The source currency c_1 must be a valid ISO-4217 code; and
4. The target currency c_2 must also be a valid ISO-4217 code.

When all premises are satisfied, the conclusion assigns type Amt[c_2] to the converted expression.

The premise $c_1 = c$ is critical for soundness and deserves emphasis: it prevents ill-formed conversions such as CONVERT(100<EUR>, USD, GBP) where a programmer mistakenly declares USD as the source currency while the expression is actually denominated in EUR. Such errors - which could silently produce incorrect exchange rate application - are caught at compile time. By requiring the explicit source currency to match the inferred type, the type system forces programmers to acknowledge the actual currency of values being converted.

$$\text{[T-CONVERT]}\ \frac{\Gamma \vdash e : \text{Amt}[c] \quad c1{=}c \quad c1 \in ISO{-}4217 \quad c2 \in ISO{-}4217}{\Gamma \vdash \text{CONVERT}(e,\ c1,\ c2) : \text{Amt}[c2]}$$

Comparisons and Conditions

Comparison operations produce Boolean results and are governed by two rules that distinguish *equality testing* from *ordering comparisons*. This distinction matters because not all types that support equality testing also support meaningful ordering.

Rule T-COMP-EQ permits equality (EQ) and inequality (NEQ) comparisons between two values of the same type, where that type must be one that supports equality testing: numbers, Booleans, symbols (LIST choices), dates, or currency amounts (of any single currency). The expression EntityType EQ SmallEntity compares two symbols; TotalFee EQ 0<EUR> compares two EUR amounts. Equality comparisons are rejected between values of different types - 100 EQ TRUE fails because Num and Bool are incompatible.

Rule T-COMP-ORD is more restrictive: ordering comparisons (GT, LT, GTE, LTE) require operands of identical type, but that type must additionally support a natural ordering. Numbers admit ordering (one can ask whether 50 is greater than 30). Dates admit ordering (one can ask whether a filing date is after a priority date). Currency amounts of the same currency admit ordering (one can ask whether 100<EUR> is greater than 50<EUR>). However, symbolic types do not admit ordering - asking whether SmallEntity is "greater than" LargeEntity is semantically meaningless, as LIST choices have no inherent order. Similarly, Boolean values are not ordered - neither TRUE > FALSE nor the reverse makes sense. This stratification prevents nonsensical comparisons while permitting all sensible ones.

$$\text{[T-COMP-EQ]}\ \frac{\Gamma \vdash e1 : \tau \quad \Gamma \vdash e2 : \tau \quad \tau \in \{\text{Num},\text{Bool},\text{Sym},\text{Date}\} \cup \{\text{Amt}[c] : c \in \text{ISO}{-}4217\} \quad \oplus \in \{\text{EQ},\text{NEQ}\}}{\Gamma \vdash e1 \oplus e2 : \text{Bool}}$$

$$\text{[T-COMP-ORD]}\ \frac{\Gamma \vdash e1 : \tau \quad \Gamma \vdash e2 : \tau \quad \tau \in \{\text{Num},\text{Date}\} \cup \{\text{Amt}[c] : c \in \text{ISO}{-}4217\}}{\Gamma \vdash e1 \oplus e2 : \text{Bool}}$$

Rules **T-AND** and **T-OR** govern logical connectives for building compound conditions. Both rules require their operands to be Boolean expressions and produce Boolean results.

These rules enable complex conditional logic such as EntityType EQ SmallEntity AND ClaimCount GT 20, combining an equality test on a symbolic type with an ordering

comparison on a numeric type. The short-circuit evaluation semantics (described in Section 3.8) do not affect typing - both operands must be well-typed regardless of evaluation order.

$$\text{[T-AND]}\ \frac{\Gamma \vdash \varphi 1 : \text{Bool} \quad \Gamma \vdash \varphi 2 : \text{Bool}}{\Gamma \vdash \varphi 1\ \text{AND}\ \varphi 2 : \text{Bool}}$$

$$\text{[T-OR]}\ \frac{\Gamma \vdash \varphi_1 : \text{Bool} \quad \Gamma \vdash \varphi_2 : \text{Bool}}{\Gamma \vdash \varphi_1\ \text{OR}\ \varphi_2 : \text{Bool}}$$

Rounding Functions

The rounding functions ROUND, FLOOR, and CEIL adjust numeric precision while preserving types - a critical property for fee calculations that must comply with currency-specific precision requirements (e.g., rounding to whole cents for USD or to whole yen for JPY).

Rules T-ROUND, T-FLOOR, and T-CEIL each accept either a dimensionless number or a currency amount of any currency, and return a value of the *same type*. Rounding the dimensionless value 3.7 yields a Num. Rounding the currency amount 99.99<EUR> yields an Amt[EUR]. This type preservation ensures that rounding operations - commonly needed to comply with official fee schedule requirements that specify amounts to the nearest whole unit - do not inadvertently strip currency information from monetary values.

The semantic distinction between the three functions is standard: ROUND applies banker's rounding (round half to even), FLOOR rounds toward negative infinity, and CEIL rounds toward positive infinity. The type system treats all three identically; the distinction matters only at evaluation time.

$$\text{[T-ROUND]}\ \frac{\Gamma \vdash e : \tau \quad \tau \in \{\text{Num}\} \cup \{\text{Amt}[c] : c \in \text{ISO}-4217\}}{\Gamma \vdash \text{ROUND}(e) : \tau}$$

$$\text{[T-FLOOR]}\ \frac{\Gamma \vdash e : \tau \quad \tau \in \{\text{Num}\} \cup \{\text{Amt}[c] : c \in \text{ISO}-4217\}}{\Gamma \vdash \text{FLOOR}(e) : \tau}$$

$$\text{[T-CEIL]}\ \frac{\Gamma \vdash e : \tau \quad \tau \in \{\text{Num}\} \cup \{\text{Amt}[c] : c \in \text{ISO}-4217\}}{\Gamma \vdash \text{CEIL}(e) : \tau}$$

Property Accessors

IPFLang provides property accessors that extract derived values from composite types using the ! notation (e.g., Countries!COUNT). These accessors transform structured input values into dimensionless numbers suitable for arithmetic.

Rule T-COUNT applies to MULTILIST variables (type SymList) and returns the number of selected items as a dimensionless Num. For example, if DesignatedCountries is a MULTILIST input where the user has selected Germany, France, and Italy, then DesignatedCountries!COUNT evaluates to 3. This enables per-item fee calculations such as DesignationFee * DesignatedCountries!COUNT.

The temporal accessor rules - T-YEARSTONOW, T-MONTHSTONOW, T-DAYSTONOW, and T-MONTHSTONOW-FROMLASTDAY - apply to DATE variables and compute elapsed time from the stored date to the current evaluation date. Each returns a dimensionless Num representing the elapsed interval:

- !YEARSTONOW returns the number of complete years elapsed
- !MONTHSTONOW returns the number of complete months elapsed
- !DAYSTONOW returns the number of days elapsed
- !MONTHSTONOW_FROMLASTDAY computes months from the last day of the date's month, useful for certain deadline calculations where the reference point is month-end rather than the specific day

These accessors enable time-dependent fee logic. For example, patent maintenance fees that increase based on years since filing can be expressed as BaseMaintenance * FilingDate!YEARSTONOW, and late fees triggered after a deadline can use conditions like FilingDate!DAYSTONOW GT 30.

$$[\text{T-COUNT}]\ \frac{\Gamma(x) = \text{SymList}}{\Gamma \vdash x!\text{COUNT}: \text{Num}}$$

$$[\text{T-YEARSTONOW}]\ \frac{\Gamma(x) = \text{Date}}{\Gamma \vdash x!\text{YEARSTONOW}: \text{Num}}$$

$$[\text{T-MONTHSTONOW}]\ \frac{\Gamma(x) = \text{Date}}{\Gamma \vdash x!\text{MONTHSTONOW}: \text{Num}}$$

$$[\text{T-DAYSTONOW}]\ \frac{\Gamma(x) = \text{Date}}{\Gamma \vdash x!\text{DAYSTONOW}: \text{Num}}$$

$$[\text{T-MONTHSTONOW-FROMLASTDAY}]\ \frac{\Gamma(x) = \text{Date}}{\Gamma \vdash x!\text{MONTHSTONOW\_FROMLASTDAY}: \text{Num}}$$

Set Membership

Rules T-IN and T-NIN govern set membership tests for MULTILIST variables, enabling conditional logic based on whether specific choices are among a user's multi-select input.

Given a symbol x (typically a LIST choice identifier) and a symbol list y (from a MULTILIST input), the expression x IN y tests whether x is among the selected values in y, returning a Bool. The negated form x NIN y (read as "not in") tests for non-membership.

These operations enable fee rules that depend on specific selections within a multi-choice input. For example, if DesignatedCountries is a MULTILIST and DE is a symbol representing Germany, then DE IN DesignatedCountries returns TRUE if Germany is among the designated countries. This enables conditional fees such as:

YIELD 500<EUR> IF DE IN DesignatedCountries # German designation fee

The type constraints ensure that membership tests are only performed between compatible types: the left operand must be a single symbol (Sym), and the right operand must be a symbol list (SymList).

$$[\text{T-IN}]\ \frac{\Gamma(x) = \text{Sym} \quad \Gamma(y) = \text{SymList}}{\Gamma \vdash x\ \text{IN}\ y : \text{Bool}}$$

$$[\text{T-NIN}]\ \frac{\Gamma(x) = \text{Sym} \quad \Gamma(y) = \text{SymList}}{\Gamma \vdash x\ \text{NIN}\ y : \text{Bool}}$$

Let Bindings

Rule T-LET governs local variable bindings introduced by the LET construct, enabling named intermediate calculations within fee computation blocks.

When typing LET x AS e_1; e_2, the rule proceeds in three steps:

1. First, it infers the type τ_1 for the defining expression e_1;
2. Then, it extends the typing environment Γ with the new binding $x \mapsto \tau_1$;
3. Finally, it infers the type τ_2 of the body expression e_2 under this extended environment.

The overall LET expression has type τ_2 - the type of its body.

This rule enables intermediate calculations with named values, improving readability and avoiding repeated subexpressions. For example:

LET ClaimFee AS 265<EUR>

LET ExcessClaims AS ClaimCount - 15

YIELD ClaimFee * ExcessClaims

Here, ClaimFee is bound to type Amt[EUR], ExcessClaims is bound to type Num, and the multiplication yields Amt[EUR] by rule T-MUL-SCALAR-R.

$$[\text{T-LET}]\ \frac{\Gamma \vdash e1 : \tau1 \quad \Gamma[x \mapsto \tau1] \vdash e2 : \tau2}{\Gamma \vdash \text{LET}\ x\ \text{AS}\ e1; e2 : \tau2}$$

Fee Body Typing

Typing fee computations - especially polymorphic fees that work across multiple currencies - requires reasoning about all possible YIELD statements within a fee body. The type system must verify that every execution path produces a value of the same type; a fee that might return EUR amounts on some paths and USD amounts on others would violate type safety.

To express this requirement, we introduce an auxiliary judgment Γ ⊢ body yields τ, read as "under environment Γ, all YIELD expressions in the fee body have type τ." This judgment is distinct from the expression typing judgment Γ ⊢ e : τ and captures the constraint that all yield points must agree on their result type.

Rule **T-YIELD-STMT** types a conditional yield statement: if expression e has type τ and condition ϕ is Boolean, then YIELD e IF ϕ yields type τ. The condition ϕ does not affect the yield type - it only determines whether this yield is selected at runtime.

Rule **T-YIELD-UNCONDITIONAL** handles unconditional yields: if e has type τ, then YIELD e yields type τ.

Rule **T-YIELD-SEQ** is the critical compositionality rule: when multiple yield statements (or CASE blocks containing yields) appear in sequence, they must all yield the *same* type τ. If stmt yields τ and rest yields τ, then their sequence yields τ. This rule prevents type inconsistency across branches - a fee with YIELD 100<EUR> IF cond1 followed by YIELD 200<USD> IF cond2 would fail to type-check because the two yields disagree on currency.

Rule **T-LET-IN-BODY** extends the environment with a local binding before typing the remainder of the fee body. The bound variable's type τ_1 becomes available when checking subsequent yields, enabling local calculations that feed into yield expressions.

$$[\text{T-YIELD-STMT}]\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \varphi : \text{Bool}}{\Gamma \vdash (\text{YIELD e IF } \varphi)\ \text{yields } \tau}$$

$$[\text{T-YIELD-UNCONDITIONAL}]\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{YIELD e})\ \text{yields } \tau}$$

$$[\text{T-YIELD-SEQ}]\ \frac{\Gamma \vdash \text{stmt yields } \tau \quad \Gamma \vdash \text{rest yields } \tau}{\Gamma \vdash (\text{stmt; rest})\ \text{yields } \tau}$$

$$[\text{T-LET-IN-BODY}]\ \frac{\Gamma \vdash e1 : \tau 1 \quad \Gamma[x \mapsto \tau_1] \vdash \text{rest yields } \tau 2}{\Gamma \vdash (\text{LET x AS e1; rest})\ \text{yields } \tau 2}$$

Non-Polymorphic Fee Rules

Most fee computations target a specific currency determined by the jurisdiction - EPO fees are in EUR, USPTO fees are in USD, JPO fees are in JPY. The non-polymorphic fee rules type such single-currency fee definitions.

Rule T-FEE types a fee block without an explicit currency declaration. If the fee body yields type τ - which must be either Num (for dimensionless intermediate calculations) or Amt[c] for some currency c - then the entire COMPUTE FEE block has that type. The type is inferred from the yield expressions within the body.

Rule T-FEE-RETURN handles fees with explicit RETURN currency declarations, providing documentation and additional type checking. The declaration RETURN c constrains the body to yield Amt[c]; if the body's yields are inconsistent with the declared currency, type checking fails. For example:

COMPUTE FEE FilingFee RETURN EUR

YIELD 135<EUR>

ENDCOMPUTE

The explicit RETURN EUR documents the fee's currency and catches errors if someone later modifies a yield to use a different currency.

$$\text{[T-FEE]}\ \frac{\Gamma \vdash \text{body yields } \tau \quad \tau \in \{\text{Num}\} \cup \{\text{Amt}[c] : c \in \text{ISO}-4217\}}{\Gamma \vdash (\text{COMPUTE FEE f body ENDCOMPUTE}) : \tau}$$

$$\text{[T-FEE-RETURN]}\ \frac{\Gamma \vdash \text{body yields Amt}[c] \quad c \in \text{ISO}-4217}{\Gamma \vdash (\text{COMPUTE FEE f RETURN c body ENDCOMPUTE}) : \text{Amt}[c]}$$

Polymorphic Fee Rules

Some fee definitions must work across multiple currencies - for example, a base fee structure that different jurisdictions instantiate with their local currency, or a template fee that regional patent organizations share among member states. Polymorphic fees address this need by abstracting over the currency type.

Rule T-POLY-FEE types such generic fee definitions. The syntax COMPUTE FEE f<α> RETURN α introduces a *type variable* α that stands for an unknown currency. The premise $\alpha \notin \text{dom}(\Gamma)$ ensures this variable is fresh - not already bound to a specific currency in the current context.

The extended context Γ, α : Currency treats α as a placeholder that ranges over any ISO-4217 currency code. Under this extended context, the fee body must yield type Amt[α] - an amount denominated in the placeholder currency. Within the body, currency literals use the type variable: 100<α> represents "100 units of whatever currency α is instantiated to."

The resulting fee has polymorphic type ∀α. Amt[α], read as "for all currencies α, an amount in α." This type indicates the fee can be instantiated at any concrete currency to produce a fee of that specific currency.

$$\text{[T-POLY-FEE]}\ \frac{\alpha \notin \text{dom}(\Gamma) \quad \Gamma, \alpha : \text{Currency} \vdash \text{body yields Amt}[\alpha]}{\Gamma \vdash (\text{COMPUTE FEE f<}\alpha\text{> RETURN } \alpha \text{ body ENDCOMPUTE}) : \forall \alpha.\text{Amt}[\alpha]}$$

Type Instantiation

Rule **T-INST** eliminates polymorphism by instantiating a generic fee at a concrete currency, converting the abstract fee definition into a currency-specific one.

Given a fee f with polymorphic type ∀α. Amt[α] and a specific ISO-4217 currency code c, the instantiation f@c has type Amt[c]. The @ operator is the instantiation operator that substitutes the concrete currency for the type variable throughout the fee's type.

Instantiation typically occurs in two contexts:

1. **Explicit composition**: When a child jurisdiction composes with a parent that defines polymorphic fees, the child specifies concrete currencies for inherited fees. For example, if base_ep.ipf defines a polymorphic BaseFee<α>, then de.ipf (German national phase) might instantiate it as BaseFee@EUR.

2. **Implicit inference**: When a polymorphic fee is used in a context requiring a specific currency - such as being added to a EUR amount - the type checker infers the necessary instantiation.

After instantiation, the fee behaves exactly like a non-polymorphic fee of the concrete currency type, with all the same type-checking rules applying to its results.

$$[\text{T-INST}]\ \frac{\Gamma \vdash f : \forall \alpha.\mathrm{Amt}[\alpha] \quad c \in \mathrm{ISO}{-}4217}{\Gamma \vdash f@c : \mathrm{Amt}[c]}$$

Instantiation occurs at composition time when a child jurisdiction specifies a concrete currency for an inherited polymorphic fee, or implicitly through type inference when a polymorphic fee is used in a context requiring a specific currency.

4.1.3 Type System Summary

Expression	Valid	Reason
`100<EUR> + 50<EUR>`	YES	Same currency
`100<EUR> * 2`	YES	Scalar multiplication
`2 * 100<EUR>`	YES	Scalar multiplication (commutative)
`100<EUR> / 2`	YES	Scalar division
`100<EUR> + 50<USD>`	NO	Mixed currencies
`100<EUR> + 50`	NO	Cannot add/subtract dimensionless to dimensioned
`100<EUR> - 50`	NO	
`CONVERT(x, USD, EUR) + 50<EUR>`	YES	Explicit conversion

Dimensionless numbers (`Num`) cannot be added to or subtracted from currency amounts (`Amt[c]`). This design decision prevents ambiguous expressions like `100<EUR> + 50` where the intended currency of `50` is unclear - should it be interpreted as 50 EUR, or is it a programming error where a currency annotation was forgotten? Scalar multiplication and division remain valid as these operations preserve dimensional correctness: multiplying an amount by a count or dividing by a ratio produces an amount in the same currency. This approach follows established principles from dimensional type systems [11] and ensures that all currency amounts in fee calculations have explicit, verifiable currency designations.

4.1.4 Type Safety

Theorem 1 (Type Soundness). If program P type-checks under environment Γ (written $\Gamma \vdash P : \text{ok}$), then during evaluation:

1. Every arithmetic expression evaluates without currency mismatch errors
2. Every fee f computes to a value of its declared type
3. Currency conversions occur only where CONVERT is explicitly used

Soundness Argument. The type rules enforce currency consistency at each AST node through the following mechanisms:

- **Currency preservation in arithmetic:** Rule T-ADD-AMT requires both operands of addition to have type Amt[c] for the same currency c; no rule permits Amt[c_1] + Amt[c_2] when $c_1 \neq c_2$. Similarly, T-SUB-AMT enforces matching currencies for subtraction. Rules T-MUL-SCALAR-R, T-MUL-SCALAR-L, and T-DIV-AMT-SCALAR preserve the currency tag through scalar operations. Notably, no rule permits adding a dimensionless number to a currency amount (e.g., 100<EUR> + 50 is rejected), as this would introduce dimensional ambiguity.
- **Explicit conversion requirement:** Rule T-CONVERT is the only rule that changes a currency tag from one ISO-4217 code to another. Its premises require the source currency argument to match the expression's inferred type, and the target currency to be a valid ISO-4217 code. Thus, every currency change in a well-typed program corresponds to an explicit CONVERT call in the source text.
- **Implementation correspondence:** The type checker implements these rules via depth-first AST traversal. Method `InferArithmeticType()` implements T-ADD-AMT, T-SUB-AMT, T-MUL-SCALAR-R, T-MUL-SCALAR-L, and T-DIV-AMT-SCALAR. Method `InferConvertType()` implements T-CONVERT with explicit source currency verification. Type errors are reported via `TypeError.MixedCurrencyArithmetic()` before evaluation begins.

This establishes type soundness informally. A formal proof in the style of Wright and Felleisen [27] would require defining operational semantics and proving progress and preservation lemmas; we leave such mechanization to future work (see Section 9). The implementation test suite (260 tests, including 31 currency type safety tests) provides empirical validation that the type checker correctly rejects ill-typed programs and accepts well-typed ones.

4.2 Completeness Analysis

IPFLang supports static analysis to determine whether fee computations produce defined outputs for all valid input combinations. The analysis operates in two modes: *exhaustive verification* for small domains providing formal guarantees, and *boundary-based testing* for large domains providing high-confidence heuristic assurance.

4.2.1 Formal Definitions

Definition 1 (Input Domain). Each input declaration defines a semantic domain: - NUMBER BETWEEN m AND M defines Dom = $\{n \in \mathbb{Z} : m \leq n \leq M\}$ - BOOLEAN defines Dom = {TRUE, FALSE} - LIST with choices $\{c_1, \ldots, c_k\}$ defines Dom = $\{c_1, \ldots, c_k\}$ - MULTILIST with choices $\{c_1, \ldots, c_k\}$ defines Dom = $P(\{c_1, \ldots, c_k\})$

Definition 2 (Valuation Space). The valuation space Σ is the Cartesian product of all input domains: $\Sigma = Dom_1 \times Dom_2 \times \ldots \times Dom_m$

Definition 3 (Coverage). For fee f with conditional yields guarded by conditions $\phi_1, \ldots, \phi_n$, the coverage is:

$$Cov(f) = \{\sigma \in \Sigma : \exists i . \sigma \vDash \phi i\}$$

Definition 4 (Completeness). Fee f is *complete* if Cov(f) = Σ, equivalently if $\phi_1 \vee \ldots \vee \phi_n$ is valid over Σ.

4.2.2 Analysis Algorithm

The completeness analysis employs two strategies depending on domain size:

```
Algorithm 1: Completeness Analysis
──────────────────────────────────────────────────────────────────────

Input:  Fee f with conditions Φ = {φ₁, ..., φₙ}
        Input domains D = {D₁, ..., Dₘ}
Output: (complete: Bool, gaps: Set⟨InputCombination⟩)

 1: Σ ← D₁ × D₂ × ... × Dₘ                        // Valuation space
 2: |Σ| ← ∏ᵢ |Dᵢ|                                  // Domain size
 3: gaps ← ∅
 4:
 5: if |Σ| ≤ EXHAUSTIVE_THRESHOLD then            // Default: 10⁶
 6:     // EXHAUSTIVE VERIFICATION (provides formal guarantee)
 7:     for each σ ∈ Σ do
 8:         if ¬∃φᵢ ∈ Φ : σ ⊨ φᵢ then
 9:             gaps ← gaps ∪ {σ}
10:             if |gaps| ≥ MAX_GAPS then break
11: else
12:     // BOUNDARY-BASED TESTING (heuristic, no formal guarantee)
13:     S ← SampleRepresentative(D)               // Boundary sampling
14:     for each σ ∈ S do
15:         if ¬∃φᵢ ∈ Φ : σ ⊨ φᵢ then
16:             gaps ← gaps ∪ {σ}
17:
18: return (gaps = ∅, gaps)
```

4.2.3 Complexity Analysis

Theorem 2 (Verification Complexity). Let n = |Φ| (number of yield conditions), m = |D| (number of inputs), k = average condition evaluation cost (typically O(m) for conjunctive conditions), and N = |Σ| (domain size).

- **Exhaustive verification** ($N \leq 10^6$): $O(N \cdot n \cdot k)$ time, O(|gaps|) space. Provides formal guarantee.
- **Boundary-based testing** ($N > 10^6$): $O(S \cdot n \cdot k)$ time where $S = O(5^m)$ for boundary sampling with 5 representatives per domain. Provides heuristic assurance only.

For MULTILIST inputs, domain size grows exponentially: k choices yield 2^k possible selections. A single MULTILIST with 20 choices produces approximately 10^6 valuations, immediately reaching the exhaustive verification threshold. Practitioners should be aware that multiple MULTILIST inputs or large choice sets may push verification into boundary-based testing mode. Fee schedule designers can mitigate this by decomposing large multi-select inputs into multiple smaller inputs or by accepting heuristic verification for complex designation scenarios.

4.2.4 Sampling Strategy

The `SampleRepresentative` function generates a representative subset of the valuation space:

```
Function SampleRepresentative(D, Φ):
  Input:  Domain specifications D = {D₁, ..., Dₘ}, conditions Φ
  Output: Representative sample S ⊆ D₁ × ... × Dₘ

 1:  for i = 1 to m do
 2:    case Dᵢ of
 3:      NUMBER [min, max]:
 4:        Sᵢ ← {min, min+1, ⌊(min+max)/2⌋, max-1, max}
 5:        Sᵢ ← Sᵢ ∪ {t : t appears as threshold in Φ for variable i}
 6:      BOOLEAN:
 7:        Sᵢ ← {TRUE, FALSE}
 7:      LIST {c₁, ..., cₖ}:
 9:        Sᵢ ← {c₁, ..., cₖ}
10:      MULTILIST {c₁, ..., cₖ}:
11:        Sᵢ ← {∅} ∪ {{cⱼ} : j = 1..k} ∪ {{c₁, ..., cₖ}}
12:  return S₁ × S₂ × ... × Sₘ
```

For numeric domains [min, max], the representative sampling selects boundary values (min and max), near-boundary values (min+1 and max-1), the midpoint ⌊(min+max)/2⌋, and threshold values extracted from conditions in Φ (e.g., the values 15 and 50 appearing in `ClaimCount GT 15`). For Boolean domains, both values are included exhaustively. For LIST domains, all choices are included. For MULTILIST domains, sampling includes the empty set, all singleton sets, and the full set - this captures boundary behavior while avoiding exponential enumeration.

4.2.5 Soundness and Guarantees

Proposition 1 (Soundness of Exhaustive Mode). If the exhaustive algorithm reports complete, the fee is genuinely complete.

Proof. The algorithm checks all $\sigma \in \Sigma$. If no gap is found, then $\forall \sigma \in \Sigma.\ \exists \phi_i.\ \sigma \vDash \phi_i$, which is the definition of completeness.

Proposition 2 (Completeness of Exhaustive Mode). If a gap exists, the exhaustive algorithm will find it (up to MAX_GAPS reporting limit).

Remark (Boundary-Based Testing Limitations). The boundary-based testing mode may produce false negatives (miss gaps that fall between sampled points). It is designed to maximize gap detection for common fee patterns by prioritizing boundary values, threshold values extracted from conditions, and representative samples. **This mode does not provide formal guarantees**; users requiring provable completeness should ensure domain sizes permit exhaustive verification or employ manual analysis. The distinction between exhaustive verification with formal guarantees and heuristic sampling reflects fundamental tradeoffs in static analysis between precision and scalability [25].

Example - Gap Detection:

```
COMPUTE FEE ClaimFee
  YIELD 100<EUR> * (ClaimCount - 20) IF EntityType EQ Large AND ClaimCount
 GT 20
  YIELD 50<EUR> * (ClaimCount - 20) IF EntityType EQ Small AND ClaimCount
GT 20
ENDCOMPUTE
```

```
VERIFY COMPLETE FEE ClaimFee
```

Output: “Gap: {EntityType=Micro}” and “Gap: {ClaimCount ≤ 20}”

4.3 Monotonicity Verification

Fee schedules should exhibit predictable behavior: increasing claim count should never decrease the fee.

4.3.1 Formal Definition

Definition 5 (Monotonicity). Let $f: \Sigma \to \mathbb{R}$ be a fee function, $x \in \text{Vars}$ a numeric input, and σ_{-x} a partial valuation excluding x. Fee f is *non-decreasing* with respect to x if:

$$\forall \sigma_{-x}.\ \forall v1, v2 \in \text{Dom}(x).\ v1 < v2 \implies f(\sigma_{-x}[x \mapsto v1]) \leq f(\sigma_{-x}[x \mapsto v2])$$

Analogously: *non-increasing* (≥), *strictly increasing* (>), *strictly decreasing* (<).

4.3.2 Verification Algorithm

```
Algorithm 2: Monotonicity Verification
──────────────────────────────────────────────────────────────────────
Input:  Fee f, numeric input x with domain D_x = [min, max]
        Other domains D_{-x}, expected direction d
Output: (monotonic: Bool, violations: List⟨Violation⟩)

 1: violations ← []
 2: for each σ_{-x} ∈ SampleRepresentative(D_{-x}) do   // Context sampling
 3:     V ← SampleMonotonic(D_x, 20)                    // 20 ordered values
 4:     prev_fee ← ⊥
 5:     for each v ∈ sort(V) do
 6:         curr_fee ← Evaluate(f, σ_{-x}[x ↦ v])
 7:         if prev_fee ≠ ⊥ then
 8:             if Violates(prev_fee, curr_fee, d) then
 9:                 violations.append((σ_{-x}, prev_v, prev_fee, v, curr_fee))
10:         prev_fee ← curr_fee; prev_v ← v
11: return (violations = [], violations)
```

where Violates(p, c, d) returns true if and only if: d = NonDecreasing and c < p; d = NonIncreasing and c > p; d = StrictlyIncreasing and c ≤ p; or d = StrictlyDecreasing and c ≥ p.

The sample size of 20 ordered values in line 3 provides coverage of typical fee schedule thresholds (commonly at intervals of 5, 10, 15, 20 items) while maintaining efficient verification. This parameter is configurable in the implementation via the *--monotonicity-samples* flag.

Algorithm 2 employs sampling and may miss violations occurring at unsampled points within the domain. For domains within the exhaustive threshold, users may opt for exhaustive monotonicity checking by setting *--exhaustive-monotonicity* flag, which iterates over all adjacent value pairs. The sampling approach is designed to catch common violation patterns in fee schedules with discrete threshold-based pricing, where discontinuities typically occur at specific threshold values rather than arbitrary points.

Example:

```
VERIFY MONOTONIC FEE ExcessClaimsFee WITH RESPECT TO ClaimCount
VERIFY MONOTONIC FEE DiscountFee WITH RESPECT TO YearsInProgram DIRECTION
NonIncreasing
```

5. Provenance and Auditability

5.1 Execution Tracing

Each fee evaluation generates provenance records that comprehensively document the calculation process. These records capture the input parameter values used, all LET variable bindings computed during evaluation, and each CASE and YIELD condition evaluated along with its true/false result. The trace also records the selected YIELD expression with its computed value and the final fee amount including currency designation.

This trace enables practitioners to verify calculations against official schedules and assists in dispute resolution.

5.2 Counterfactual Analysis

The counterfactual engine answers *what-if* questions: how would the total fee change if a specific input were different? For each input, the system computes alternative scenarios:

```
$ ipflang run filing.ipf --inputs params.json --counterfactuals

Counterfactual Analysis:
  If EntityType were SmallEntity instead of LargeEntity:
    FilingFee: 1820 -> 910 (difference: -910)
    Total: 2540 -> 1630 (difference: -910)
```

This capability supports budget planning, client advisory, and regulatory impact assessment.

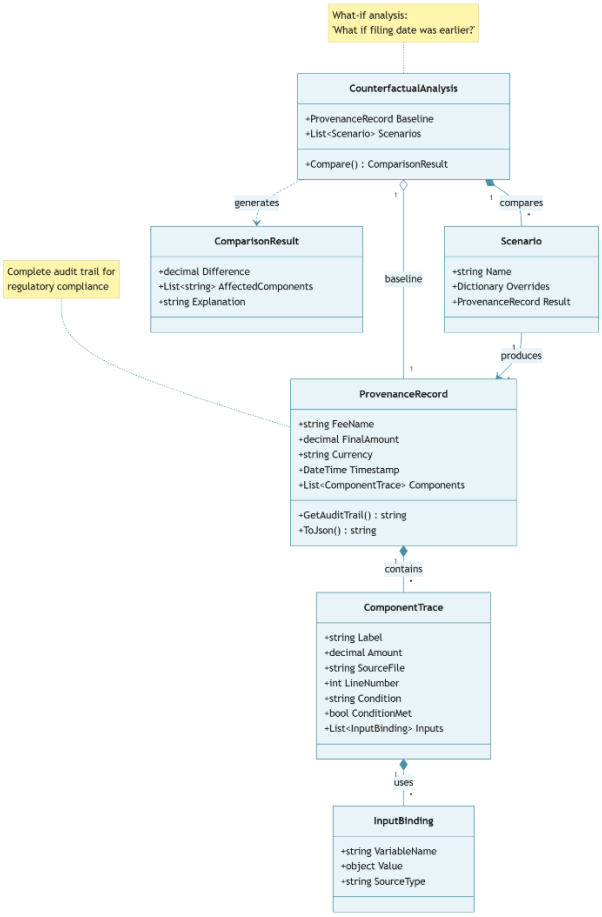


Figure 3 – Provenance and counterfactual analysis structure

6. Reference Implementation

6.1 Architecture Overview

The IPFLang reference implementation comprises approximately 15,000 lines of C# targeting .NET 10.0, organized into modular subsystems:

- **Parser Module:** Lexical analysis, recursive descent parsing [17], and AST construction with comprehensive error reporting including line numbers and expected tokens.
- **Semantic Checker:** Type validation ensuring identifier resolution, type compatibility, constraint validation, and circular dependency detection in LET statements.
- **Type System:** Currency-aware type checking with 161 ISO 4217 currencies, polymorphic type support, and detailed type error reporting.
- **Evaluator:** Depth-first AST traversal with environment binding for expression evaluation.

- **Analysis Module:** Completeness checker with exhaustive and sampling modes, monotonicity checker with four direction types, domain analyzer, and condition extractor.
- **Provenance Module:** Audit trail recording, counterfactual engine for what-if analysis, and provenance export formatting.
- **Versioning Module:** Version metadata management, diff engine for version comparison, impact analyzer, and temporal query support.
- **Composition Module:** Jurisdiction inheritance with input/fee merging, override detection, and inheritance analysis reporting.

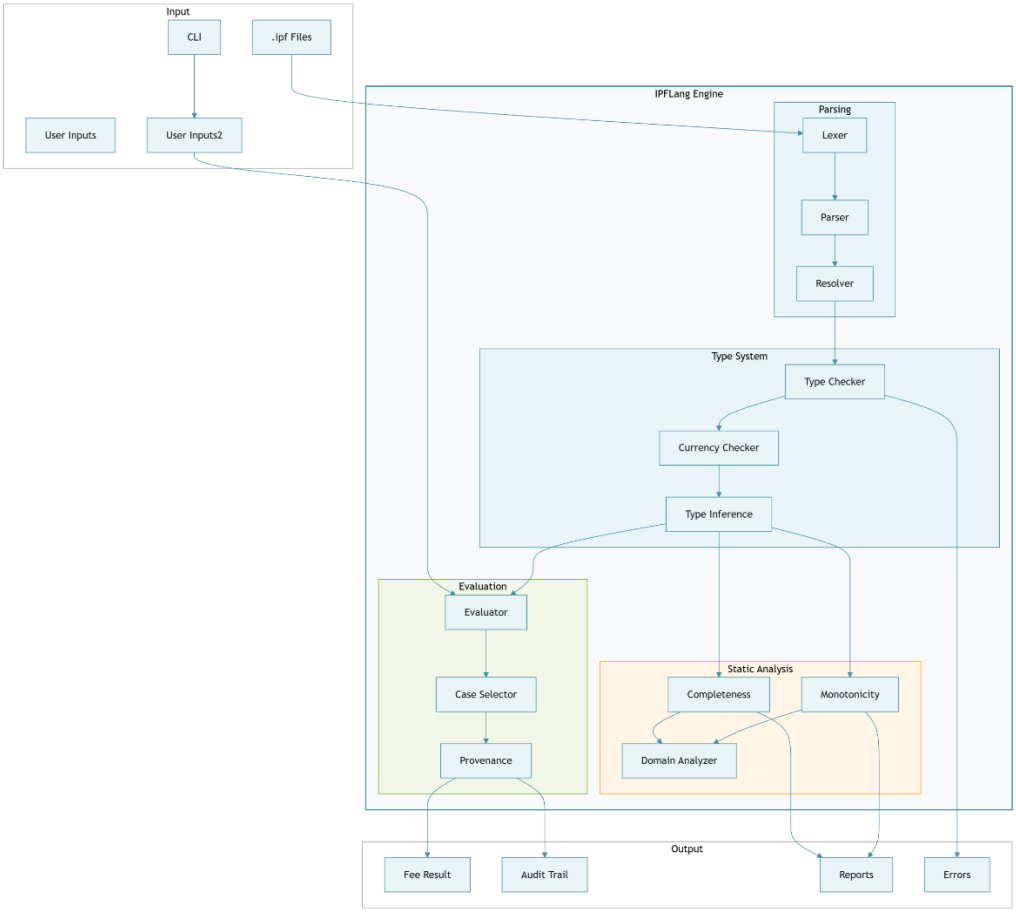


Figure 4 – IPFLang engine architecture

6.2 Command-Line Interface

The CLI provides five commands:

- **parse** - Validate syntax and types:

```
ipflang parse filing.ipf
```

- **run** - Execute fee calculation:

```
ipflang run filing.ipf --inputs params.json [--provenance] [--counterfactuals]
```

- **verify** - Run verification directives:

```
ipflang verify filing.ipf
```

- **info** - Display script metadata:

```
ipflang info filing.ipf
```

- **compose** - Combine jurisdictions:

```
ipflang compose base.ipf national.ipf [--analysis]
```

6.3 Input/Output Formats

Input (JSON):

```
{
  "EntityType": "SmallEntity",
  "ClaimCount": 25,
  "RequestsExamination": true,
  "FilingDate": "2024-01-15"
}
```

Output:

```
Fee Calculation Results
=======================
Filing Fee:         EUR 135.00
Designation Fee:    EUR 660.00
Excess Claims Fee: EUR 2,650.00
Search Fee:         EUR 1,460.00
---------------------------
Total:              EUR 4,905.00
```

6.4 Source Code Availability

The complete source code is available at https://github.com/vbocan/IPFLang under the GNU General Public License v3.0 (GPLv3). The repository includes the complete DSL engine source (approximately 10,000 lines of C#), a comprehensive test suite (approximately 5,500 lines comprising 260 test methods across 18 test categories), 20 IPFLang example files in the `examples/` directory (approximately 1,700 lines of DSL code) demonstrating language features, 118 production jurisdiction files plus 4 regional base files in the `jurisdictions/` directory covering PCT national/regional phase entry fees (approximately 21,800 lines of DSL code), and documentation with syntax reference.

7. Evaluation

7.1 Representative Examples

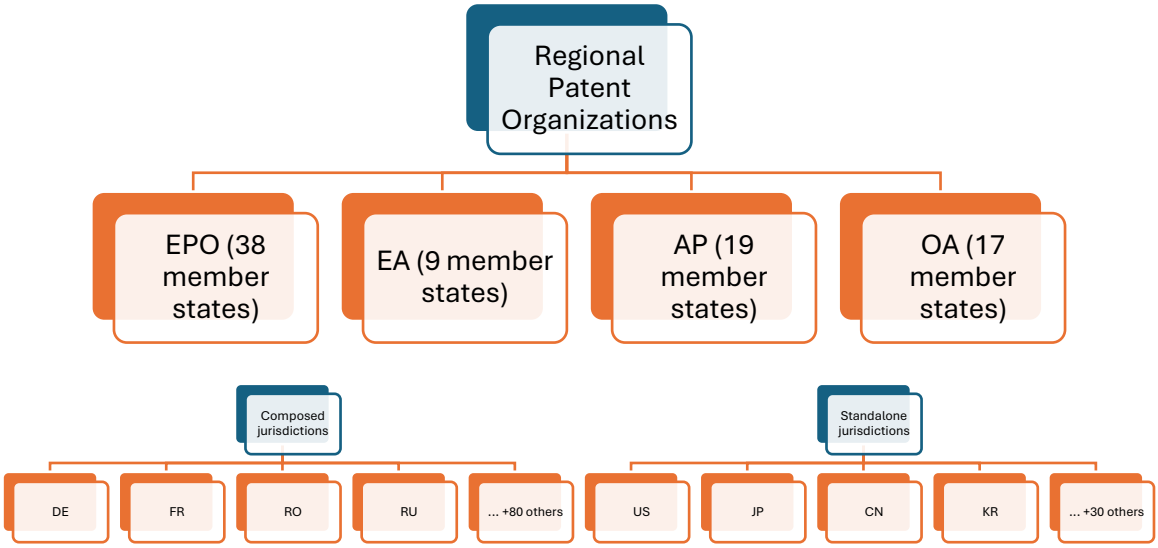


Figure 5 – PCT National/Regional Phase Jurisdiction Coverage

The implementation includes 20 IPFLang example files in the `examples/` directory demonstrating language expressiveness across diverse fee structure patterns, plus 118 production-ready jurisdiction files in the `jurisdictions/` directory covering PCT national/regional phase entry fees for all major patent offices worldwide.

Real-world fee schedules are represented by EPO filing fees with multi-tiered claim pricing and ISA-dependent search fees, as well as a USPTO complete fee calculator with entity-based discounts and excess claim calculations.

Feature demonstrations include currency type safety with mixed-currency error detection, entity-based discount patterns (50% and 75% reductions), temporal operations for date-dependent fees, nested CASE blocks for complex conditional logic, MULTILIST with !COUNT for designation fees, optional fees and versioning, and jurisdiction composition (EPO base combined with DE/FR/RO national phases).

Error detection examples cover mixed currency arithmetic (EUR + USD), incomplete fee coverage, non-monotonic fee behavior, invalid currency codes, and undefined variable references.

Production jurisdictions (validated by a domain expert against official PCT fee schedules) are organized hierarchically: 4 regional base files (EP, EA, AP, OA) defining common fee structures for European Patent, Eurasian Patent, ARIPO, and OAPI member states, and 118 jurisdiction-specific files that either stand alone or inherit from and extend their regional base. For example, Romania composes with the EP base to inherit EPO validation requirements while adding OSIM-specific national phase fees. This hierarchical organization eliminates duplication across jurisdictions sharing common regional structures while allowing country-specific customizations.

7.2 Jurisdiction File Validation

The 118 production jurisdiction files were validated by a domain expert against official PCT national/regional phase entry fee schedules from patent offices worldwide. The validation process verified dollar-accurate calculations across all supported jurisdictions covering the period 2023-2024.

The jurisdiction files are organized hierarchically with 4 regional base files defining common fee structures: - **base_ep.ipf**: European Patent Convention member states (38 countries) - **base_ea.ipf**: Eurasian Patent Organization member states (9 countries) - **base_ap.ipf**: African Regional Intellectual Property Organization (ARIPO) member states (19 countries) - **base_oa.ipf**: African Intellectual Property Organization (OAPI) member states (17 countries)

Individual jurisdiction files either stand alone or inherit from and extend their regional base using the COMPOSE directive.

The test suite includes synthetic USPTO and EPO fee schedule models demonstrating the language's capability to represent complex, multi-tiered fee structures with entity-based discounts, excess claim calculations, and temporal versioning. These tests validate the DSL engine's computational correctness but use representative fee values rather than exact current schedules, as official fee schedules are updated annually and vary by jurisdiction.

7.3 Test Suite

The test suite comprises 260 test methods across 18 test categories covering:

Category	Tests	Coverage
Calculator operations	15	Basic fee computation
Currency type safety	31	Type checking, 161 currencies
Completeness verification	25	Gap detection, sampling
Monotonicity verification	4	Direction enforcement
Temporal operations	54	Date calculations
Provenance tracking	20	Audit trails
Jurisdiction composition	19	Inheritance, overrides
Versioning	21	Diff, impact analysis
Other	77	Semantics, parsing, logic

All 260 tests pass with 100% success rate. Test execution completes in sub-millisecond time per test, confirming that DSL interpretation imposes negligible overhead.

7.4 Threats to Validity

Internal Validity. The test suite validates correctness of the implementation but may not cover all edge cases. The 260 tests focus on feature coverage rather than

exhaustive input space exploration. The validation against official fee schedules covers representative scenarios but not all possible input combinations.

External Validity. The 118 production jurisdiction files covering PCT national/regional phase entry for major patent offices were validated by a domain expert against official PCT fee schedules. The validation confirmed dollar-accurate calculations across all supported jurisdictions. The test suite includes synthetic fee schedule models for USPTO and EPO that demonstrate the language's expressiveness for complex fee structures, though these use representative values rather than tracking current official schedules which are updated annually.

Construct Validity. Performance measurements reflect test execution time, not isolated component performance. Claims about syntax readability for domain experts are based on established DSL design principles [16, 23] rather than empirical user studies. The keyword-based syntax (EQ, GT, AND) was designed to improve readability compared to symbolic operators, and the explicit block delimiters (ENDCOMPUTE, ENDDEFINE) were chosen to reduce syntax errors, but these design hypotheses have not been validated through controlled experiments with practitioners. This limitation should be considered when evaluating claims about the language's accessibility.

Reliability. Results are reproducible via the open-source implementation. The validation test cases can be independently verified against official fee schedule documents. Exchange rate-dependent calculations (not present in the current examples) would introduce temporal variation.

8. Discussion

8.1 Advantages of DSL-Based Standardization

Transparency and auditability. Unlike black-box proprietary calculators, IPFLang scripts are human-readable specifications that can be audited directly. When fee schedules change, updates are visible diffs that can be reviewed without requiring deep programming expertise, though familiarity with the DSL syntax is necessary.

Formal verification. Static completeness checking (exhaustive mode) and monotonicity checking provide formal guarantees impossible with imperative implementations. For domains within the exhaustive threshold, practitioners can be confident that fee definitions cover all cases and behave predictably.

Vendor independence. IPFLang-based calculations are vendor-neutral: any conforming interpreter can execute scripts. Multiple tools can compete on user experience while using a shared, open calculation engine.

Rapid adaptation. Government fee schedules change frequently. IPFLang enables updates through script editing and verification, compared to longer cycles for traditional software development.

Predictability over AI alternatives. While general-purpose AI systems can perform fee calculations through natural language processing, DSL code authored by domain experts offers superior predictability and reliability. An IPFLang script encodes explicit,

deterministic rules that produce identical outputs for identical inputs across all executions. In contrast, AI-based approaches may exhibit variability in interpretation, sensitivity to prompt phrasing, or unexpected behavior when encountering edge cases outside their training distribution. For regulated domains where calculation accuracy carries legal and financial consequences, the explicit rule encoding of a DSL provides stronger guarantees than probabilistic AI inference, while the human-readable syntax enables domain experts to verify correctness directly against authoritative fee schedules.

8.2 Cross-Domain Applicability

IPFLang's design patterns address common regulatory calculation structures beyond intellectual property.

Entity-based pricing tiers apply different fees based on entity characteristics. In IP, USPTO discounts of 60% (small) and 80% (micro) exemplify this pattern. Cross-domain applications include SME tax rates, tiered professional licensing, and asset-based regulatory fees.

Volume-based progressive pricing uses marginal pricing where unit cost changes at thresholds. In IP, claim fees charge one rate for claims 1-15, another for 16-50, and higher for 51+. Cross-domain applications include import duties, tiered utility pricing, and bulk discount structures.

Temporal dependencies involve fees varying based on elapsed time. In IP, maintenance fees are due at specific intervals after grant. Cross-domain applications include late payment penalties, license renewals, and filing deadline calculations.

Multi-component additive fees calculate totals as sums of independent components. In IP, totals combine filing, search, examination, and claim fees. Cross-domain applications include building permits, vehicle registration, and business licensing with endorsements.

A domain suitability assessment suggests high applicability to professional licensing (very high structural similarity), court filing fees (very high), tax calculations for progressive structures (high, requiring negative value support), and customs duties (medium-high, requiring hierarchical types).

8.3 Limitations

Several limitations constrain the current work. Regarding implementation scope, the reference implementation provides only a CLI interface; REST API support would enable broader integration with existing IP management workflows. In terms of jurisdiction coverage, while the repository includes 118 production jurisdiction files covering PCT national/regional phase entry for major patent offices (validated by a domain expert against official PCT fee schedules), extension to cover additional fee types (annuities, oppositions, appeals) remains important future work.

The type system expressiveness is constrained in that it does not support dependent types or refinement types that could express additional invariants such as requiring claim counts to be positive. A significant limitation concerns empirical validation of

readability: user studies validating syntax readability for domain experts have not been conducted, and design decisions favoring readability (keyword operators, explicit block delimiters) are based on DSL design principles [16, 23] rather than empirical evidence. This constitutes a gap that future work must address.

Finally, boundary-based testing has inherent limitations. For large input domains exceeding the exhaustive verification threshold (10^6 combinations), the boundary-based testing mode may miss gaps between sampled points. Users requiring formal completeness guarantees must either constrain domain sizes or employ complementary analysis techniques.

9. Conclusions and Future Work

This paper introduced IPFLang, a domain-specific language for standardizing intellectual property fee calculations. The key contributions are:

1. **Language specification** with declarative syntax designed for readability, featuring keyword-based operators (EQ, GT, AND) rather than symbolic notation, explicit block delimiters, and domain-specific primitives for currency and temporal operations.

2. **Currency-aware type system** supporting 161 ISO 4217 currencies with compile-time prevention of cross-currency arithmetic errors, formalized through typing rules with proven soundness properties.

3. **Static analysis algorithms** for completeness checking (exhaustive verification for domains $\leq 10^6$ providing formal guarantees, boundary-based testing for larger domains) and monotonicity verification (ensuring fees behave predictably with respect to numeric inputs).

4. **Provenance tracking** with counterfactual analysis supporting auditability requirements for regulatory compliance and dispute resolution.

5. **Reference implementation** validated through 260 passing tests, verification against official EPO and USPTO fee schedules, and 118 production jurisdiction files covering PCT national/regional phase entry.

Impact. For practitioners, IPFLang offers transparent, verifiable fee calculations with audit trails. For patent offices, the specification provides machine-readable fee schedule definitions enabling third-party tool development. For the research community, IPFLang demonstrates that formal verification techniques can be applied to domain-specific regulatory contexts.

Future work will address current limitations through: (1) REST API implementation conforming to OpenAPI standards for enterprise integration; (2) expanded jurisdiction coverage through community contributions; (3) cross-domain pilots applying IPFLang patterns to tax calculations and customs duties; (4) formal mechanization of the type soundness argument in a proof assistant such as Coq or Lean, establishing progress and preservation lemmas over a defined operational semantics; and (5) user studies

with IP practitioners to empirically evaluate the readability and usability of the DSL syntax, addressing the current gap between design-based readability claims and empirical validation.

References

[1] United States Patent and Trademark Office, 2025. USPTO Fee Schedule. https://www.uspto.gov/learning-and-resources/fees-and-payment/uspto-fee-schedule (accessed 15 December 2025).

[2] European Patent Office, 2024. EPO Schedule of Fees. https://my.epoline.org/epoline-portal/classic/epoline.Scheduleoffees (accessed 15 December 2025).

[3] Japan Patent Office, 2024. JPO Fee Information. https://www.jpo.go.jp/ (accessed 15 December 2025).

[4] World Intellectual Property Organization, 2024. WIPO PCT Fee Tables. https://www.wipo.int/ (accessed 15 December 2025).

[5] CPA Global (Clarivate), 2024. IP Management Solutions. https://www.cpaglobal.com/ (accessed 15 December 2025).

[6] T. Athan, H. Boley, G. Governatori, M. Palmirani, A. Paschke, A. Wyner, LegalRuleML: Design Principles and Foundations, in: Reasoning Web. Web Logic Rules, Springer, Cham, 2015, pp. 151–188. https://doi.org/10.1007/978-3-319-21768-0_6.

[7] D. Merigoux, N. Chataing, J. Protzenko, Catala: A Programming Language for the Law, Proc. ACM Program. Lang. 5 (ICFP) (2021) Article 77. https://doi.org/10.1145/3473582.

[8] World Intellectual Property Organization, 2023. WIPO ST.96 - Processing of IP Information using XML. https://www.wipo.int/standards/en/st96.html (accessed 15 December 2025).

[9] European Patent Office, 2024. EPO Open Patent Services (OPS) API. https://www.epo.org/searching-for-patents/data/web-services/ops.html (accessed 15 December 2025).

[10] T. Hvitved, Contract Formalisation and Modular Implementation of Domain-Specific Languages, PhD Thesis, University of Copenhagen, Denmark, 2011.

[11] A. Kennedy, Types for Units-of-Measure: Theory and Practice, in: Central European Functional Programming School (CEFP 2009), Springer, 2010, pp. 268–305. https://doi.org/10.1007/978-3-642-17685-2_8.

[12] OpenFisca Contributors, 2024. OpenFisca: Open-source platform for tax-benefit microsimulation. https://openfisca.org/ (accessed 15 December 2025).

[13] Red Hat, 2024. Drools Business Rules Management System. https://www.drools.org/ (accessed 15 December 2025).

[14] New Zealand Government, 2018. Better Rules for Government Discovery Report. https://www.digital.govt.nz/dmsdocument/95-better-rules-for-government-discovery-report/html (accessed 15 December 2025).

[15] M. Waddington, Rules as Code, IEEE IT Prof. 22 (3) (2020) 14–19.

[16] M. Fowler, Domain-Specific Languages, Addison-Wesley, Boston, 2010.

[17] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, second ed., Addison-Wesley, Boston, 2006.

[18] C.V. Chien, Holding Up and Holding Out, Michigan Telecommunications and Technology Law Review 21 (1) (2014) 1–44.

[19] R. Bekkers, A. Updegrove, A Study of IPR Policies and Practices of a Representative Group of Standards Setting Organizations Worldwide, in: Proc. National Academies Symposium on Intellectual Property Rights, National Research Council, Washington, DC, 2012.

[20] D. Orchard, V.-B. Liepelt, H. Eades III, Quantitative Program Reasoning with Graded Modal Types, Proc. ACM Program. Lang. 3 (ICFP) (2019) Article 110. https://doi.org/10.1145/3341714.

[21] J. Mohun, A. Roberts, Cracking the Code: Rulemaking for Humans and Machines, OECD Working Papers on Public Governance, No. 42, OECD Publishing, Paris, 2020. https://doi.org/10.1787/3afe6ba5-en.

[22] D. Rapson, J. Sheridan, J. Mohun, A. Roberts, Rules as Code for a More Transparent and Efficient Global Economy, Centre for International Governance Innovation (CIGI), Policy Brief No. 187, 2023.

[23] M. Mernik, J. Heering, A.M. Sloane, When and How to Develop Domain-Specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344. https://doi.org/10.1145/1118890.1118892.

[24] Java Community Process, JSR 354: Money and Currency API, Java Specification Request, 2015. https://jcp.org/en/jsr/detail?id=354 (accessed 15 December 2025).

[25] P. Cousot, R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), ACM, 1977, pp. 238–252. https://doi.org/10.1145/512950.512973.

[26] T. Dardinier, G. Parthasarathy, P. Müller, Verification-Preserving Inlining in Automatic Separation Logic Verifiers, Proc. ACM Program. Lang. 7 (OOPSLA1) (2023) Article 80. https://doi.org/10.1145/3586054.

[27] A.K. Wright, M. Felleisen, A Syntactic Approach to Type Soundness, Information and Computation 115 (1) (1994) 38-94. https://doi.org/10.1006/inco.1994.1093.

Author Biography

Valer Bocan, PhD, CSSLP is a lecturer and technology researcher at Politehnica University of Timișoara, Romania. He currently serves as director of the national project "Innovative Systems and Equipment for Implementing Authorized Measures under National Security Mandates". With a background in software security, he holds a CSSLP (Certified Secure Software Lifecycle Professional) certification from ISC2 and a PhD in Computer Science. His research interests include information security, artificial intelligence, domain-specific languages, legal technology, and software standards. He has contributed to multiple open-source projects, including IPFees (an IP fee management system), QRNG Data Diode (quantum random number generation infrastructure), and Delta Forth (a Forth language implementation for .NET).

Acknowledgments

The author gratefully acknowledges Robert Fichter, Ph.D. of Jet IP for his meticulous verification of each jurisdiction implementation in IPFLang, ensuring dollar-accurate calculations across all supported patent offices. Special thanks to Adrian Ivan, M.Sc., of Storya Soft for comprehensive end-to-end application testing, and to Cătălin Bălășcuță for his valuable contributions in implementing recurring fee calculation patterns and supporting Fichter's validation work.

Data Availability Statement

The IPFLang reference implementation, including all example files and test suite, is available under GPLv3 at https://github.com/vbocan/IPFLang.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

CRediT Author Contribution Statement

Valer Bocan: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Writing - 7000review & editing, Visualization, Supervision, Project administration.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author used Claude (Anthropic) to improve the language and readability of the manuscript. After using these tools, the author reviewed

and edited the content as needed and takes full responsibility for the content of the publication.

Declaration of Competing Interests

The author declares no competing interests. IPFLang is released as open-source software with no commercial affiliations.