IPFLang Syntax Manual

Welcome to IPFLang, a domain-specific language designed specifically for expressing intellectual property fee calculations. Whether you're an IP practitioner creating fee schedules or a developer integrating fee logic into larger systems, this manual will guide you through the language's syntax and capabilities.

IPFLang brings precision and clarity to the complex world of IP fees. It features a currency-aware type system that prevents mixing currencies accidentally, static verification to ensure your fee schedules cover all cases, and powerful temporal operations for handling deadlines and renewals. The language is designed to read almost like natural language while maintaining the rigor needed for accurate financial calculations.

Table of Contents

Core Concepts

At its heart, an IPFLang script is a recipe for calculating fees. You begin by declaring what information you need from the user—these are your **inputs**. Perhaps you need to know the entity type (large, small, or micro), the number of claims in a patent application, or whether examination is requested.

Next, you define **fees** that compute monetary amounts based on those inputs. A filing fee might be $500 for small entities and $1000 for large entities. An excess claims fee might charge $50 for each claim beyond the first 20.

Finally, you specify what **returns** to report back—typically totals of mandatory fees, optional fees, and grand totals.

What makes IPFLang special are its advanced features:

Currency-Aware Type System: The language prevents you from accidentally adding euros to dollars. Each monetary value carries its currency, and the type system enforces consistency at compile time.

Completeness Verification: You can ask IPFLang to verify that your fee definitions cover every possible combination of inputs. No more discovering gaps in production.

Provenance Tracking: The system records exactly which rules contributed to each fee, creating an audit trail for transparency and compliance.

Version Management: Track fee schedule changes over time with effective dates and regulatory references, enabling historical queries and impact analysis.

Writing Your First Script

Let's start with something simple—a basic filing fee that depends on entity type:

```
DEFINE LIST EntityType AS 'Entity type'
CHOICE Large AS 'Large entity'
CHOICE Small AS 'Small entity'
DEFAULT Large
ENDDEFINE

COMPUTE FEE FilingFee
YIELD 1000 IF EntityType EQ Large
YIELD 500 IF EntityType EQ Small
ENDCOMPUTE

RETURN Total AS 'Total Filing Fee'
```

This script defines one input (EntityType), computes one fee (FilingFee), and returns one value (Total). Notice how readable it is—the `YIELD` statements read almost like English sentences.

Comments

As your scripts grow, you'll want to annotate them. Use the # character to start a comment that extends to the end of the line:

```
# This is a full-line comment
DEFINE NUMBER ClaimCount AS 'Number of claims'  # Inline comment after code
BETWEEN 0 AND 100
DEFAULT 1
ENDDEFINE
```

Comments can appear anywhere except inside quoted strings. They're invaluable for documenting the reasoning behind complex fee structures or noting regulatory references.

Version Declaration

Fee schedules change over time. Patent offices adjust their prices annually, regulators issue new rules, and courts clarify interpretations. IPFLang's version system helps you track these changes systematically.

At the top of your script, you can declare a version with an effective date and optional metadata:

```
VERSION '<VersionId>' EFFECTIVE yyyy-MM-dd [DESCRIPTION '<description>']
[REFERENCE '<reference>']
```

The version identifier should be unique—many organizations use year-based schemes like '2024.1' or semantic versioning like '1.2.0'. The effective date tells you when this version goes into force. Descriptions and regulatory references are optional but highly recommended for auditability.

Here's the simplest form:

```
VERSION '2024.1' EFFECTIVE 2024-01-15
```

Adding a description makes it easier to understand what changed:

```
VERSION '2024.1' EFFECTIVE 2024-01-15 DESCRIPTION 'Annual fee increase'
```

For compliance and auditing, include a regulatory reference:

```
VERSION '2024.1' EFFECTIVE 2024-01-15 REFERENCE 'Federal Register Vol. 89, No.
123'
```

You can combine all elements for maximum clarity:

```
VERSION '2024.1' EFFECTIVE 2024-01-15 DESCRIPTION 'USPTO fee increase' REFERENCE
'Federal Register Vol. 89, No. 123'

DEFINE NUMBER ClaimCount AS 'Number of claims'
BETWEEN 1 AND 100
DEFAULT 1
ENDDEFINE

COMPUTE FEE FilingFee RETURN USD
  YIELD 100<USD>
ENDCOMPUTE
```

Version declarations are optional but highly recommended, especially for fee schedules that change regularly. Place them at the very top of your script, before any other definitions. Each script can have only one version directive.

When you work with IPFLang programmatically, the version system unlocks powerful capabilities: comparing versions to see what changed, querying historical fees as they were on a specific date, and analyzing the impact of fee changes across different scenarios.

Input Definitions

Inputs are the questions you ask your users. IPFLang provides six input types, each designed for a specific kind of data. Let's explore each one.

LIST Input

The LIST input presents users with a set of choices and allows them to select exactly one. Think of it as a dropdown menu or radio button group.

```
DEFINE LIST <Name> AS '<Display Text>'
CHOICE <Symbol1> AS '<Label1>'
CHOICE <Symbol2> AS '<Label2>'
...
DEFAULT <DefaultSymbol>
GROUP <GroupName>
ENDDEFINE
```

Here's an entity type selector with three choices:

```
DEFINE LIST EntityType AS 'Entity type'
CHOICE NormalEntity AS 'Normal entity'
CHOICE SmallEntity AS 'Small entity'
CHOICE MicroEntity AS 'Micro entity'
DEFAULT NormalEntity
GROUP General
ENDDEFINE
```

Each `CHOICE` has a symbol (like `NormalEntity`) that you'll use in your code, and a label (like `'Normal entity'`) that users will see. The `DEFAULT` specifies which choice is pre-selected. The optional `GROUP` organizes this input in the user interface—more on groups later.

In your fee computations, you test against the symbols:

```
YIELD 100 IF EntityType EQ NormalEntity
YIELD 50 IF EntityType EQ SmallEntity
```

MULTILIST Input

Sometimes users need to select multiple items from a list. MULTILIST handles this—think checkboxes rather than radio buttons.

Consider designation countries for a European patent:

```
DEFINE MULTILIST DesignationCountries AS 'Designation countries'
CHOICE DE AS 'Germany'
CHOICE FR AS 'France'
CHOICE GB AS 'United Kingdom'
CHOICE IT AS 'Italy'
DEFAULT DE,FR
GROUP Designations
ENDDEFINE
```

The `DEFAULT` can specify multiple pre-selected items, separated by commas.

MULTILIST inputs have a special property: `!COUNT` returns how many items were selected. This is perfect for per-country fees:

```
YIELD 100 * DesignationCountries!COUNT
```

This charges 100 units for each designated country.

NUMBER Input

For numeric inputs like claim counts or page counts, use NUMBER. You can specify minimum and maximum bounds to prevent invalid values:

```
DEFINE NUMBER ClaimCount AS 'Number of claims'
BETWEEN 1 AND 1000
DEFAULT 10
GROUP Claims
ENDDEFINE

DEFINE NUMBER PageCount AS 'Number of pages'
BETWEEN 1 AND 500
DEFAULT 30
GROUP Application
ENDDEFINE
```

NUMBER inputs participate in arithmetic naturally. Here's a common pattern—charging for claims beyond a threshold:

```
# Excess claims fee (claims over 10)
LET ExcessClaims AS ClaimCount - 10
YIELD 50 * ExcessClaims IF ExcessClaims GT 0
```

This calculates how many claims exceed 10 and charges $50 for each excess claim.

BOOLEAN Input

For yes/no questions, BOOLEAN provides a simple true/false choice:

```
DEFINE BOOLEAN RequestsExamination AS 'Requests examination'
DEFAULT TRUE
GROUP Examination
ENDDEFINE

DEFINE BOOLEAN ClaimsSequenceListing AS 'Contains sequence listing'
DEFAULT FALSE
GROUP Claims
ENDDEFINE
```

Use it in conditions just like you'd expect:

```
YIELD 500 IF RequestsExamination EQ TRUE
```

This charges a $500 examination fee only if examination was requested.

DATE Input

Many IP fees depend on dates—filing dates, priority dates, renewal dates. DATE inputs handle temporal data with bounds checking.

Dates are written in `dd.MM.yyyy` format (day, month, year). The special keyword `TODAY` refers to the current date:

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

DEFINE DATE PriorityDate AS 'Priority date'
BETWEEN 01.01.1990 AND TODAY
DEFAULT 01.01.2024
GROUP Priority
ENDDEFINE
```

DATE inputs come with powerful properties for temporal calculations:

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

These properties enable annuity calculations, late fee assessments, and deadline tracking:

```
# Annuity calculation based on years since filing
LET YearsSinceFiling AS FilingDate!YEARSTONOW
YIELD 100 * FLOOR(YearsSinceFiling)
```

This charges $100 per complete year since the filing date.

AMOUNT Input

Sometimes users need to provide monetary amounts as inputs—translation costs, search fees, or other variable expenses. AMOUNT handles this, but unlike plain numbers, it enforces currency awareness.

When you define an AMOUNT, you specify which currency it uses:

```
DEFINE AMOUNT PriorArtSearchFee AS 'Prior art search fee'
CURRENCY EUR
DEFAULT 1500
GROUP Search
ENDDEFINE

DEFINE AMOUNT TranslationCost AS 'Translation cost'
```

```
CURRENCY USD
DEFAULT 2000
GROUP Translation
ENDDEFINE
```

The currency code (like EUR or USD) must be a valid ISO 4217 code. Once defined, IPFLang's type system ensures you don't accidentally mix currencies:

```
# Same-currency arithmetic is type-safe
YIELD PriorArtSearchFee + 500<EUR>

# Mixed currencies cause compile-time type error!
# YIELD PriorArtSearchFee + TranslationCost  # ERROR: EUR + USD
```

This catches errors before they reach production. If you need to combine different currencies, you must use explicit conversion (covered later).

Organizing with Groups

As your scripts grow, you'll accumulate many inputs. Groups help organize them into logical sections for better user experience.

Define groups with a name, display label, and weight:

```
DEFINE GROUP General AS 'General Information' WITH WEIGHT 1
DEFINE GROUP Claims AS 'Claims Information' WITH WEIGHT 2
DEFINE GROUP Fees AS 'Fee Options' WITH WEIGHT 3
```

The weight determines display order—lower numbers appear first. This lets you control how inputs are presented to users. When you define an input, you can assign it to a group:

```
DEFINE NUMBER ClaimCount AS 'Number of claims'
BETWEEN 1 AND 1000
DEFAULT 10
GROUP Claims
ENDDEFINE
```

This groups all claim-related inputs together in the interface.

Fee Computations

Now we reach the heart of IPFLang—computing fees. Every fee calculation begins with `COMPUTE FEE` and ends with `ENDCOMPUTE`. Inside, you use `YIELD` statements to produce values.

Basic Fees

The simplest fee just yields a constant:

```
COMPUTE FEE FilingFee
YIELD 500
ENDCOMPUTE
```

This filing fee is always $500, regardless of inputs. Of course, real fees are rarely this simple.

Optional Fees

Some fees are optional—expedited processing, for example. Mark these with the `OPTIONAL` keyword:

```
COMPUTE FEE ExpeditedProcessingFee OPTIONAL
YIELD 1000
ENDCOMPUTE
```

Optional fees are computed and displayed, but reported separately from mandatory fees in totals. This helps users understand which costs they can avoid.

Cases and Conditions

Most fees vary based on inputs. You have two ways to express this: simple conditional yields, or structured CASE blocks.

For straightforward conditions, put `IF` directly on the yield:

```
COMPUTE FEE SimpleFee
YIELD 100 IF EntityType EQ MicroEntity
YIELD 200 IF EntityType EQ SmallEntity
YIELD 400 IF EntityType EQ NormalEntity
ENDCOMPUTE
```

For more complex logic, especially when one input affects multiple calculations, use CASE blocks:

```
COMPUTE FEE EntityBasedFee
CASE EntityType EQ NormalEntity AS
  YIELD 1000 IF ClaimCount LTE 20
  YIELD 1000 + (50 * (ClaimCount - 20)) IF ClaimCount GT 20
ENDCASE
CASE EntityType EQ SmallEntity AS
  YIELD 500 IF ClaimCount LTE 20
  YIELD 500 + (25 * (ClaimCount - 20)) IF ClaimCount GT 20
```

```
ENDCASE
CASE EntityType EQ MicroEntity AS
  YIELD 250 IF ClaimCount LTE 20
  YIELD 250 + (12 * (ClaimCount - 20)) IF ClaimCount GT 20
ENDCASE
ENDCOMPUTE
```

Each CASE groups related logic together. This reads more naturally and makes completeness verification more effective: IPFLang can check that each entity type has a complete set of rules.

Local Variables

Fee calculations often involve intermediate values. The `LET` keyword lets you name these values, making your logic clearer and avoiding repetition:

```
COMPUTE FEE ExcessClaimsFee
LET ExcessClaims AS ClaimCount - 10
LET ExcessPages AS PageCount - 30
YIELD 50 * ExcessClaims IF ExcessClaims GT 0
YIELD 20 * ExcessPages IF ExcessPages GT 0
ENDCOMPUTE
```

Here, `ExcessClaims` and `ExcessPages` are computed once and reused. This is clearer than writing `ClaimCount - 10` multiple times, and it ensures consistency.

Polymorphic Fees

Advanced users sometimes need fees that work with multiple currencies. Polymorphic fees provide this flexibility through type parameters:

```
# A fee that returns the same currency type it receives
COMPUTE FEE ScaledFee<C> RETURN C
YIELD 100<C> * Multiplier
ENDCOMPUTE
```

The `<C>` is a type variable—a placeholder for whatever currency you use when invoking the fee. This fee can work with EUR, USD, or any other currency, maintaining type safety throughout.

Expressions

Expressions are the building blocks of your fee calculations. IPFLang provides familiar arithmetic, comparison, and logical operators.

Arithmetic Operators

IPFLang supports the standard arithmetic operations you'd expect:

Operator	Description	Example
`+`	Addition	`ClaimCount + 5`
`-`	Subtraction	`ClaimCount - 10`
`*`	Multiplication	`50 * ExcessClaims`
`/`	Division	`TotalFee / 2`
`( )`	Grouping	`(ClaimCount - 10) * 50`

Combine them naturally:

```
LET ExcessClaims AS ClaimCount - 10
LET BaseFee AS 500
YIELD BaseFee + (ExcessClaims * 50)
```

Parentheses work as you'd expect, controlling order of operations.

Comparison Operators

For conditional logic, IPFLang provides readable comparison operators. Rather than symbols like `==` or `<=`, it uses words:

Operator	Description	Example
`EQ`	Equal	`EntityType EQ NormalEntity`
`NEQ`	Not equal	`EntityType NEQ MicroEntity`
`LT`	Less than	`ClaimCount LT 10`
`LTE`	Less than or equal	`ClaimCount LTE 20`
`GT`	Greater than	`ClaimCount GT 10`
`GTE`	Greater than or equal	`ClaimCount GTE 5`
`IN`	Is in list	`Country IN DesignatedCountries`
`NIN`	Not in list	`Country NIN ExcludedCountries`

The `IN` and `NIN` operators test membership in MULTILIST selections, perfect for per-country calculations.

Logical Operators

Combine conditions with `AND` and `OR`:

Operator	Description	Example

Operator	Description	Example
`AND`	Logical AND	`ClaimCount GT 10 AND EntityType EQ NormalEntity`
`OR`	Logical OR	`EntityType EQ SmallEntity OR EntityType EQ MicroEntity`

Use parentheses to group complex conditions:

```
YIELD 1000 IF ClaimCount GT 20 AND EntityType EQ NormalEntity
YIELD 500 IF ClaimCount GT 20 AND (EntityType EQ SmallEntity OR EntityType EQ
MicroEntity)
```

This charges different fees based on both claim count and entity type.

Built-in Functions

IPFLang includes functions for rounding and other mathematical operations:

Function	Description	Example
`ROUND(x)`	Round to nearest integer	`ROUND(Fee / 10) * 10`
`FLOOR(x)`	Round down	`FLOOR(YearsSinceFiling)`
`CEIL(x)`	Round up	`CEIL(MonthsRemaining / 12)`

These are essential for calculations involving dates and partial values:

```
LET Years AS FLOOR(FilingDate!YEARSTONOW)
YIELD 100 * Years
```

`FLOOR` ensures you charge only for complete years, not partial ones.

Currency Literals

One of IPFLang's most powerful features is its currency-aware type system. To create a monetary value with a specific currency, annotate a number with an ISO 4217 currency code:

```
100<EUR>      # 100 Euros
50.50<USD>    # 50.50 US Dollars
1000<GBP>     # 1000 British Pounds
250<JPY>      # 250 Japanese Yen
```

The syntax is straightforward: `100<EUR>`, `50.50<USD>`, etc.

What makes this powerful is type safety. You can add or subtract same-currency values freely:

```
YIELD 100<EUR> + 50<EUR>           # Valid: 150 EUR
YIELD 100<EUR> * 2                 # Valid: 200 EUR (scalar multiplication)
```

But mixing currencies triggers a compile-time error:

```
YIELD 100<EUR> + 50<USD>           # TYPE ERROR: Cannot mix EUR and USD
```

This catches mistakes that could otherwise lead to serious financial errors. If you need to combine different currencies, use explicit conversion.

Currency Conversion

When you need to combine amounts in different currencies, use the `CONVERT` function:

```
CONVERT(<Amount>, <SourceCurrency>, <TargetCurrency>)
```

This performs runtime conversion using exchange rates:

```
# Convert USD amount to EUR
YIELD CONVERT(100<USD>, USD, EUR) + 50<EUR>

# Convert an amount variable
YIELD CONVERT(TranslationCost, USD, EUR) + PriorArtSearchFee
```

The conversion is explicit and visible in your code, making it clear where currency mixing occurs. This transparency helps with auditing and prevents accidental conversions.

How Exchange Rates Work:

IPFLang fetches exchange rates from exchangerate-api.com, which provides ECB-based (European Central Bank) data. All rates are relative to EUR as the base currency. When you convert between two currencies, the system calculates: `amount / sourceRate * targetRate`.

The exchange rate data:

- Requires an API key for live rates (obtained from exchangerate-api.com)
- Is cached and refreshed periodically, not fetched on every conversion
- Includes timestamps showing when rates were last updated
- Covers all 161 ISO 4217 currency codes

For testing and development, the system can use mock exchange rates with fixed values. This ensures your scripts work even without network connectivity or API keys.

Returns

After computing all your fees, you'll want to report results. The `RETURN` directive defines named outputs:

```
RETURN TotalMandatory AS 'Total mandatory fees'
RETURN TotalOptional AS 'Total optional fees'
RETURN GrandTotal AS 'Grand total'
```

Each return has a symbol (used internally) and a display label (shown to users). The system automatically calculates totals based on your fee definitions—mandatory fees, optional fees, and combined totals.

Verification Directives

IPFLang's verification system helps you catch errors before they affect real calculations. These static analyses run at compile time, not runtime.

VERIFY COMPLETE

Completeness verification ensures your fee covers all possible input combinations. No gaps, no missed cases:

```
DEFINE LIST EntityType AS 'Entity type'
CHOICE Normal AS 'Normal'
CHOICE Small AS 'Small'
DEFAULT Normal
ENDDEFINE

COMPUTE FEE BasicFee
YIELD 100 IF EntityType EQ Normal
YIELD 50 IF EntityType EQ Small
ENDCOMPUTE

VERIFY COMPLETE FEE BasicFee
```

The verification analyzes your fee logic and confirms that every EntityType value has a corresponding yield. If you forget a case, IPFLang reports exactly which combination is missing. This is invaluable for complex fee schedules with multiple inputs—manually checking all combinations becomes impractical, but automated verification catches every gap.

VERIFY MONOTONIC

Monotonicity verification checks that a fee behaves sensibly with respect to numeric inputs. In most cases, more of something (more claims, more pages) should never decrease the fee:

```
DEFINE NUMBER ClaimCount AS 'Number of claims'
BETWEEN 1 AND 100
```

```
DEFAULT 10
ENDDEFINE

COMPUTE FEE ClaimFee
LET ExcessClaims AS ClaimCount - 10
YIELD 50 * ExcessClaims IF ExcessClaims GT 0
YIELD 0 IF ExcessClaims LTE 0
ENDCOMPUTE

VERIFY MONOTONIC FEE ClaimFee WITH RESPECT TO ClaimCount
```

This verification confirms that increasing ClaimCount never decreases ClaimFee. If you accidentally wrote logic that reduced fees at higher claim counts, IPFLang would catch it and show you the exact values where the violation occurs.

You can specify different monotonicity directions:

- `NonDecreasing` (default) - Fee never decreases as input increases
- `NonIncreasing` - Fee never increases as input increases
- `StrictlyIncreasing` - Fee always increases (no flat sections)
- `StrictlyDecreasing` - Fee always decreases

Most IP fees are non-decreasing, but the flexibility is there when you need it.

Advanced Features

Temporal Operations

IP fees often involve time—filing deadlines, renewal dates, late fees, priority periods. IPFLang includes temporal operators for these scenarios, available through the programmatic API.

The system supports:

- **Business day calculations**: Add or subtract business days, automatically excluding weekends
- **Late fee calculations**: Apply multipliers that increase daily, or use stepped tiers
- **Renewal date calculations**: Compute patent renewal cycles from filing dates
- **Priority period validation**: Check Paris Convention 12-month priority windows
- **Grace period logic**: Model payment deadline extensions with fee adjustments

These operations integrate with DATE inputs and their temporal properties (`!YEARSTONOW`, `!MONTHSTONOW`, etc.), enabling sophisticated time-based fee logic.

Jurisdiction Composition

Real-world IP fee schedules often follow hierarchical patterns. The European Patent Office defines base fees, then Germany, France, and other countries add national-phase specifics. The USPTO defines federal fees, then state-level filings add local requirements.

IPFLang's composition system lets you model these relationships explicitly. Create a base jurisdiction file with common inputs and fees, then create child jurisdiction files that inherit from it:

Base Jurisdiction (epo_base.ipf):

```
VERSION '2024.1' EFFECTIVE 2024-01-01 DESCRIPTION 'EPO Base Fees'

DEFINE LIST ApplicantType AS 'Applicant type'
CHOICE LARGE AS 'Large entity'
CHOICE SME AS 'SME (30% reduction)'
DEFAULT LARGE
ENDDEFINE

COMPUTE FEE FilingFee
CASE ApplicantType EQ LARGE AS
  YIELD 140<EUR>
ENDCASE
CASE ApplicantType EQ SME AS
  YIELD 98<EUR>
ENDCASE
ENDCOMPUTE

VERIFY COMPLETE FEE FilingFee
```

Child Jurisdiction (epo_germany.ipf):

```
VERSION '2024.2' EFFECTIVE 2024-01-01 DESCRIPTION 'EPO German National Phase'

# Inherits ApplicantType and FilingFee from parent

# Add Germany-specific input
DEFINE BOOLEAN NeedsTranslation AS 'Translation to German required'
DEFAULT TRUE
ENDDEFINE

# Add Germany-specific fee
COMPUTE FEE GermanTranslationFee
CASE NeedsTranslation EQ TRUE AS
  YIELD 1050<EUR>
ENDCASE
CASE NeedsTranslation EQ FALSE AS
  YIELD 0
ENDCASE
ENDCOMPUTE

VERIFY COMPLETE FEE GermanTranslationFee
```

The child inherits everything from the parent—inputs, fees, verifications—and adds its own elements. You can also override inherited fees by redefining them with the same name.

This composition approach offers substantial benefits:

- **Code Reuse**: Share 60-80% of definitions across related jurisdictions
- **Maintenance**: Update base fees once, changes flow to all children automatically
- **Consistency**: Ensure related jurisdictions use compatible structures
- **Auditability**: Track exactly which jurisdiction contributed each fee

---

Complete Example

Let's bring everything together with a complete European Patent Office fee calculator:

```
# ============================================
# European Patent Application Fee Calculator
# ============================================

# Version declaration
VERSION '2024.1' EFFECTIVE 2024-01-15 DESCRIPTION 'Updated EPO fees for 2024'
REFERENCE 'EPO Official Journal 2023/12'

# --- Groups ---
DEFINE GROUP General AS 'General Information' WITH WEIGHT 1
DEFINE GROUP Claims AS 'Claims' WITH WEIGHT 2
DEFINE GROUP Pages AS 'Application Pages' WITH WEIGHT 3
DEFINE GROUP Options AS 'Options' WITH WEIGHT 4

# --- Inputs ---

DEFINE LIST EntityType AS 'Applicant type'
CHOICE NormalEntity AS 'Large entity'
CHOICE SmallEntity AS 'SME (small/medium enterprise)'
CHOICE MicroEntity AS 'Micro entity / Natural person'
DEFAULT NormalEntity
GROUP General
ENDDEFINE

DEFINE NUMBER ClaimCount AS 'Total number of claims'
BETWEEN 1 AND 500
DEFAULT 10
GROUP Claims
ENDDEFINE

DEFINE NUMBER PageCount AS 'Number of application pages'
BETWEEN 1 AND 1000
DEFAULT 35
GROUP Pages
ENDDEFINE

DEFINE BOOLEAN RequestsExamination AS 'Request examination'
DEFAULT TRUE
GROUP Options
ENDDEFINE

DEFINE AMOUNT AdditionalServiceFee AS 'Additional service fee'
```

```
CURRENCY EUR
DEFAULT 0
GROUP Options
ENDDEFINE

# --- Fee Computations ---

# Filing fee with entity-based reduction
COMPUTE FEE FilingFee
CASE EntityType EQ NormalEntity AS
  YIELD 1500<EUR>
ENDCASE
CASE EntityType EQ SmallEntity AS
  YIELD 750<EUR>
ENDCASE
CASE EntityType EQ MicroEntity AS
  YIELD 375<EUR>
ENDCASE
ENDCOMPUTE

# Excess claims fee (over 15 claims)
COMPUTE FEE ExcessClaimsFee
LET ExcessClaims AS ClaimCount - 15
YIELD 250<EUR> * ExcessClaims IF ExcessClaims GT 0
ENDCOMPUTE

# Excess pages fee (over 35 pages)
COMPUTE FEE ExcessPagesFee
LET ExcessPages AS PageCount - 35
YIELD 17<EUR> * ExcessPages IF ExcessPages GT 0
ENDCOMPUTE

# Examination fee
COMPUTE FEE ExaminationFee
CASE RequestsExamination EQ TRUE AS
  YIELD 1950<EUR> IF EntityType EQ NormalEntity
  YIELD 975<EUR> IF EntityType EQ SmallEntity
  YIELD 488<EUR> IF EntityType EQ MicroEntity
ENDCASE
ENDCOMPUTE

# Additional services
COMPUTE FEE AdditionalServices
YIELD AdditionalServiceFee
ENDCOMPUTE

# --- Returns ---
RETURN EPOFiling AS 'EPO Filing Fees'
RETURN Examination AS 'Examination Fees'
RETURN Total AS 'Total Fees'
```

This example demonstrates most of IPFLang's core features: versioning, groups for organization, multiple input types, conditional fee logic with CASE blocks, local variables, currency-aware amounts, and named returns. It's production-ready code that clearly expresses complex fee calculations.

ISO 4217 Currency Codes

IPFLang supports all 161 ISO 4217 currency codes. Here are the most commonly used ones:

Code	Currency
EUR	Euro
USD	United States Dollar
GBP	British Pound Sterling
JPY	Japanese Yen
CHF	Swiss Franc
CNY	Chinese Yuan
CAD	Canadian Dollar
AUD	Australian Dollar
KRW	South Korean Won
INR	Indian Rupee
RON	Romanian Leu
PLN	Polish Zloty
SEK	Swedish Krona
DKK	Danish Krone
NOK	Norwegian Krone

Type System Summary

The currency-aware type system prevents common errors at compile time:

Expression	Valid?	Reason
`100<EUR> + 50<EUR>`	✓	Same currency
`100<EUR> * 2`	✓	Scalar multiplication
`2 * 100<EUR>`	✓	Scalar multiplication (commutative)
`100<EUR> / 2`	✓	Scalar division
`100<EUR> + 50<USD>`	✗	Mixed currencies

Expression	Valid?	Reason
`100<EUR> + 50`	✗	Cannot add dimensionless to dimensioned
`100<EUR> - 50`	✗	Cannot subtract dimensionless to dimensioned
`CONVERT(x, USD, EUR) + 50<EUR>`	✓	Explicit conversion

Type errors are caught at parse time, long before any calculation runs. This makes IPFLang particularly robust for financial applications where currency mistakes can have serious consequences.