Goldman Sachs

Extending DMN with FEEL Libraries

September 2025

Content

- Problem Statement
- Problem Solution
- Q&As

Problem Statement

- The **Decision Model and Notation (DMN)** combined with **FEEL (Friendly Enough Expression Language)** and a **general-purpose programming language** (like Java, C#, or Python) differ significantly in **expressiveness**, **purpose**, and **abstraction level**.

- Here's a comparison based on **intent**, **expressiveness**, and **flexibility**

Purpose and Domain

Feature	**DMN/FEEL**	**Programming Language**
Domain Focus	Business decision modeling. Tailored for expressing decision logic in a structured, visual, and understandable way for business users.	General-purpose computation across any domain: web dev, finance, systems, etc.
Audience	Business analysts, process modelers	Software developers
Clarity of Logic	Highly readable and self-explanatory. Uses decision tables and natural-language-like expressions.	Requires coding knowledge. Logic may not be obvious without reading the code carefully.

Control Flow and Logic Expressiveness

Aspect	**DMN/FEEL**	**Programming Language**
Data Structures	Limited to lists, contexts, and primitive types	Rich support: lists, sets, maps, custom classes, trees, graphs, etc.
Conditional Logic	Expressed via decision tables or basic if-then-else in FEEL	Fully supported: if, switch, ternary, etc.
Loops / Iteration	Limited support (for, filter, some, every in FEEL, but not procedural loops)	Full support: for, while, recursion, etc.
Functions / Methods	Simple function expressions; limited higher-order function capabilities	Full support for defining and invoking complex functions and methods

Turing Completeness

Feature	**FEEL**	**Programming Language**
Fixed-length loops	✅ Supported	✅ Supported
Unbounded loops	❌ Not allowed	✅ Yes
Recursion	❌ Forbidden	✅ Supported
Memory manipulation	❌ Not allowed	✅ Yes
Turing-complete?	❌ No	✅ Yes

Extensibility and Integration

Feature	**DMN/FEEL**	**Programming Language**
Extensibility	Limited – not intended for custom logic beyond predefined operations	Full – can build and import libraries, plugins, frameworks
Integration with Systems	Designed to be embedded in BPMN/workflows or called from applications	Can directly build or interact with full systems, APIs, databases, etc.

Summary: Use Case Fit

Use Case	**DMN/FEEL**	**Programming Language**
Business rules (e.g., loan approval, eligibility criteria)	✅ Ideal	✅ Possible but overkill
Complex data processing or algorithms	❌ Not suitable	✅ Ideal
System automation, web services	❌ Only as part of decision logic	✅ Full control
Collaboration with non-technical stakeholders	✅ Very readable	❌ Often too technical

Summary: Expressiveness Comparison

Criteria	**DMN/FEEL**	**Programming Language**
Clarity for business rules	✅ Excellent	❌ Often obscure
Flexibility and abstraction	❌ Limited	✅ High
Control structures and data handling	❌ Minimal	✅ Rich
Execution and performance	✅ Fast and optimized for decisions	✅ General-purpose, customizable
Total expressiveness (power)	❌ Not Turing complete	✅ Turing complete

Conclusion

- **DMN/FEEL** is **intentionally less expressive** than a programming language to remain simple, readable, and business-focused.
- **Programming languages** are **far more expressive**, flexible, and powerful but require technical skill and introduce complexity.
- In practice, they're **complementary**: use DMN/FEEL for **rules and decisions**; use programming languages for **system logic, orchestration, and infrastructure**.

Problem Solution: FEEL libraries

A library has:

- A **name**
- A **namespace**
- Contains a list of **function declarations**
- Each library is uniquely identified by a pair: **namespace + name**

```
library ::= 'namespace' qualifiedName ';'
        name '{' functions '}'.

functions ::= function (';' function)*.

function ::= 'function' name '(' (formalParameter (','
formalParameter)*)? ')' ':' type.
```

where name, qualifiedName, type and formalParameter are described in the FEEL grammar.

Problem Solution: Design

- The definitions of the FEEL libraries are platform-independent
- They do not contain any information about the execution platform (e.g., Java).
- The discovery mechanism of the definitions of the libraries and the artifacts needed to execute the functions (e.g., Java jars or Python modules) is vendor-specific.
- The mapping of the FEEL types to the native platforms (e.g., Java) is defined in Table 47: Mapping between FEEL and other domains.
- The functions defined in a library become visible in the scope of the evaluation (see 10.3.2.11) once they are imported in a DMN file (see 6.3.3) with the import type equal to the FEEL namespace (e.g., https://www.omg.org/spec/DMN/20240513/FEEL/).
- The functions defined in a library are invoked in the same way as the imported BKMs or Decision Services (e.g., prefix.f(a, b, c)).

Problem Solution: Example

```
namespace org.omg.feel.stringUtil;

stringLib {
   // Checks if str is empty ("") or null.
  function isEmpty(str: string) : boolean ;
  ...
   // The capitalized str, null if str is null
  function capitalize(str: string) : string
}
```

once the library is imported with

```
<import namespace="org.omg.feel.stringUtil“ name="myLib"
    importType="https://www.omg.org/spec/DMN/20240513/FEEL/"
/>
```

the function `isEmpty` can be invoked by `myLib.isEmpty("abc")`.

Problem Solution: Advantages

🔁 Reusability
FEEL libraries allow you to define common logic once (e.g., complex calculations, string parsing, date rules) and reuse it across multiple DMN models or decisions.

🧠 Abstraction of Complex Logic
Hide complicated or technical expressions behind a simple, business-friendly name, making decision tables clearer and easier to maintain.

🧩 Domain-Specific Logic
Libraries often capture specific business domain rules, such as tax rules, insurance risk scores, or eligibility criteria.

🛠️ Maintainability and Consistency
Centralized functions help ensure consistent decision logic across multiple models and are easier to update in one place when rules change.

🧪 Testability
FEEL libraries make it easier to test smaller units of logic independently, especially when using decision modeling tools that support unit testing or simulation.

Questions?