Catala: A Programming Language for the Law

Denis Merigoux, Nicolas Chataing, Jonathan Protzenko

Catala: A Programming Language for the Law

DENIS MERIGOUX, Inria, France
NICOLAS CHATAING, Inria ENS Paris, France
JONATHAN PROTZENKO, Microsoft Research, USA

Law at large underpins modern society, codifying and governing many aspects of citizens' daily lives. Oftentimes, law is subject to interpretation, debate and challenges throughout various courts and jurisdictions. But in some other areas, law leaves little room for interpretation, and essentially aims to rigorously describe a computation, a decision procedure or, simply said, an algorithm.

Unfortunately, prose remains a woefully inadequate tool for the job. The lack of formalism leaves room for ambiguities; the structure of legal statutes, with many paragraphs and sub-sections spread across multiple pages, makes it hard to compute the intended outcome of the algorithm underlying a given text; and, as with any other piece of poorly-specified critical software, the use of informal, natural language leaves corner cases unaddressed.

We introduce **Catala**, a new programming language that we specifically designed to allow a straightforward and systematic translation of statutory law into an executable implementation. **Catala** aims to bring together lawyers and programmers through a shared medium, which together they can understand, edit and evolve, bridging a gap that too often results in dramatically incorrect implementations of the law. We have implemented a compiler for **Catala**, and have proven the correctness of its core compilation steps using the F* proof assistant.

We evaluate **Catala** on several legal texts that are algorithms in disguise, notably section 121 of the US federal income tax and the byzantine French family benefits; in doing so, we uncover a bug in the official implementation of the French benefits. We observe as a consequence of the formalization process that using **Catala** enables rich interactions between lawyers and programmers, leading to a greater understanding of the original legislative intent, while producing a correct-by-construction executable specification reusable by the greater software ecosystem. Doing so, **Catala** increases trust in legal institutions, and mitigates the risk of societal damage due to incorrect implementations of the law.

1 INTRODUCTION

We now know that since at least 2000 B.C.E. and the Code of Ur-Nammu [48], various societies have attempted to edict, codify and record their governing principles, customs and rules in a set of legal texts – the law. Nowadays, most aspects of one's daily life are regulated by a set of laws or another, ranging from family law, tax law, criminal law, to maritime laws, business laws or civil rights law. No law is set in stone; laws are, over time, amended, revoked and modified by legislative bodies. The resulting legal texts eventually reflect the complexity of the process and embody the centuries of practice, debates, power struggles and political compromises between various parties.

The practice of law thus oftentimes requires substantial human input. First, to navigate the patchwork of exceptions, amendments, statutes and jurisprudence relevant to a given case. Second, to fully appreciate and identify the situation at play; understand whether one party falls in a given category or another; and generally classify and categorize, in order to interpret a real-world situation into something the law can talk about.

This latter aspect is perhaps the greatest challenge for a computer scientist: a general classification system remains an elusive prospect when so much human judgement and appreciation is involved. Fortunately, a subset of the law, called *computational law* or sometimes *rules as code*, concerns itself with situations where entities are well-defined, and where human appreciation, judgement

Authors' addresses: Denis Merigoux, denis.merigoux@inria.fr, Inria, France; Nicolas Chataing, nicolas.chataing@ens.fr, Inria, ENS Paris, France; Jonathan Protzenko, Microsoft Research, USA, protz@microsoft.com.

2021. XXXX-XXXX/2021/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

or interpretation are not generally expected. Examples of computational law include, but are not limited to: tax law, family benefits, pension computations, monetary penalties and private billing contracts. All of these are algorithms in disguise: the law (roughly) defines a function that produces outcomes based on a set of inputs.

As such, one might think computational law would be easily translatable into a computer program. Unfortunately, formidable challenges remain. First, as mentioned above, the law is the result of a centuries-long process: its convoluted structure demands tremendous expertise and training to successfully navigate and understand, something that a computer programmer may not have. Second, the language in which legal statutes are drafted is so different from existing programming languages that a tremendous gap remains between the legal text and its implementation, leaving the door open for discrepancies, divergence and eventual bugs, all with dramatic societal consequences.

Examples abound. In France, the military's payroll computation involves 174 different bonuses and supplemental compensations. Three successive attempts were made to rewrite and modernize the military paycheck infrastructure; but with a complete disconnect between the military statutes and the implementation teams that were contracted, the system had to be scrapped [34]. Software engineering failures are perhaps a fact of life in the IT world; but in this particular case, actual humans bore the consequences of the failure, with military spouses receiving a 3-cent paycheck, or learning years later that they owe astronomical amounts to the French state. Perhaps more relevant to the current news, the US government issued a decree intended to provide financial relief to US taxpayers whose personal economic situation had been affected by the Covid-19 pandemic. Owing to an incorrect implementation by the Internal Revenue Service (IRS), nearly one million Americans received an incorrect Economic Impact Payment (EIP), or none at all [15].

Both examples are similar, in that a seemingly pure engineering failure turns out to have dramatic consequences in terms of human livelihoods. When a family is at the mercy of the next paycheck or EIP, a bug in those systems could mean bankruptcy. In our view, these is no doubt that these systems are yet another flavor of critical software.

A natural thought is perhaps to try to simplify the law itself. Unfortunately, this recurrent theme of the political discourse often conflicts with the political reality that requires compromise and fined-grained distinctions. Hence, the authors do not anticipate a drastic improvement around the world concerning legal complexity. Therefore, our only hope for improvement lies on the side of programming languages and tools.

Tax law provides a quintessential example. While many of the implementations around the world are shrouded in secrecy, the public occasionally gets a glimpse of the underlying infrastructure. Recently, Merigoux *et al.* [31] reverse-engineered the computation of the French income tax, only to discover that the tax returns of an entire nation were processed using an antiquated system designed in 1990, relying on 80,000 lines of code written in a custom, in-house language, along with 6,000 lines of hand-written C directly manipulating tens of thousands of global variables. This particular situation highlights the perils of using the wrong tool for the job: inability to evolve, resulting in hand-written C patch-ups; exotic semantics which make reproducing the computation extremely challenging; and a lack of accountability, as the cryptic in-house language cannot be audited by anyone, except by the handful of experts who maintain it. This is by no means a "French exception": owing to an infrastructure designed while Kennedy was still president, the IRS recently handed over $300,000,000's worth of fraudulent refunds to taxpayers [3]. The rewrite, decades in planning, keeps being pushed further away in the future [12].

In this work, we propose a new language, **Catala**, tailored specifically for the purpose of faithfully, crisply translating computational law into executable specifications. **Catala** is designed to follow the existing structure of legal statutes, enabling a one-to-one correspondence between a legal paragraph and its corresponding translation in **Catala**. Under the hood, **Catala** uses prioritized

default logic [5]; to the best of our knowledge, **Catala** is the first instance of a programming language designed with this logic as its core system. **Catala** has clear semantics, and compiles to a generic lambda-calculus that can then be translated to any existing language. We formalize the compilation scheme of **Catala** with F* and show that it is correct. In doing so, we bridge the gap between legal statutes and their implementation; we avoid the in-house language trap; and we provide a solid theoretical foundation to audit, reproduce, evaluate and reuse computational parts of the law. Our evaluation, which includes user studies, shows that **Catala** can successfully express complex sections of the US Internal Revenue Code and the entirety of the French family benefits computation.

The benefits of using **Catala** are many: lawmakers and lawyers are given a formal language that accurately captures their intent and faithfully mirrors the prose; programmers can derive and distribute a canonical implementation, compiled to the programming language of their choice; citizens can audit, understand and evaluate computational parts of the law; and advocacy groups can shed more light on oftentimes obscure, yet essential, parts of civil society.

2 BACKGROUND ON LEGAL TEXTS & THEIR FORMALIZATION

Legal statutes are written in a style that can be confounding for a computer scientist. While a program's control-flow (as a first approximation) makes forward progress, statutes frequently back-patch previous definitions and re-interpret earlier paragraphs within different contexts. The result more closely resembles assembly with arbitrary jumps and code rewriting, rather than a structured language.

To illustrate how the law works, we focus on Section 121 of the US Internal Revenue Code [19], our running example throughout this paper. Section 121 is written in English, making it accessible to an international audience without awkward translations; it features every single difficulty we wish to tackle with **Catala**; and it is a well-studied and well-understood part of the tax law. We go through the first few paragraphs of the section; for each of them, we informally describe the intended meaning, then highlight the surprising semantics of the excerpt.

2.1 Overview of Section 121

Section 121 is concerned with the "Exclusion of gain from sale of principal residence". In common parlance, if the taxpayer makes a profit from the sale of their residence, they are not required to report the profits as income, hence making such profits non-taxable. Paragraph (a) defines the exclusion itself.

> **(a) Exclusion**
> Gross income shall not include gain from the sale or exchange of property if, during the 5-year period ending on the date of the sale or exchange, such property has been owned and used by the taxpayer as the taxpayer's principal residence for periods aggregating 2 years or more.

The part of the sentence that follows the "if" enumerates conditions under which this tax exclusion can be applied. This whole paragraph is valid *unless specified otherwise*, as we shall see shortly.

2.2 Out-of-order definitions

Paragraph (b) immediately proceeds to list *limitations*, that is, situations in which (a) does not apply, or needs to be refined. Section 121 thus consists of a general case, (a), followed by a long enumeration of limitations ranging from (b) to (g). We focus only on (b). The first limitation (b)(1) sets a maximum for the exclusion, "generally" $250,000. Left implicit is the fact that any proceeds of the sale beyond that are taxed as regular income.

> **(b) Limitations**
> **(1) In general**
> The amount of gain excluded from gross income under subsection (a) with respect to any sale or exchange shall not exceed $250,000.

We remark that even though (b)(1) is a key piece of information for the application of Section 121, the reader will find it only if they keep going after (a). This is a general feature of legal texts: relevant information is scattered throughout, and (a) alone is nowhere near enough information to make a determination of whether the exclusion applies to a taxpayer.

2.3 Backpatching; exceptions

Entering (b)(2), paragraph (A) *modifies* (b)(1) *in place*, stating that under certain conditions, the maximum exclusion can be $500,000.

> **(A) $500,000 Limitation for certain joint returns**
> Paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if—
> (i) either spouse meets the ownership requirements of subsection (a) with respect to such property;
> (ii) both spouses meet the use requirements of subsection (a) with respect to such property; and
> (iii) neither spouse is ineligible for the benefits of subsection (a) with respect to such property by reason of paragraph (3).

Several key aspects of paragraph (A) are worth mentioning. First, (A) *backpatches* paragraph (b)(1); the law essentially encodes a search-and-replace in its semantics.

Second, (A) *overrides* a previous general case under specific conditions. In a functional programming language, a pattern-match first lists the most specific matching cases, and catches all remaining cases with a final wildcard. A text of law proceeds in the exact opposite way: the general case in (a) above is listed first, then followed by limitations that modify the general case under certain conditions. This is by design: legal statutes routinely follow a "general case first, special cases later" approach which mirrors the legislator's intentions.

Third, conditions (i) through (iii) are a conjunction, as indicated by the "and" at the end of (ii). We note that (iii) contains a forward-reference to (3) which we have not seen yet. (Through our work, we fortunately have never encountered a circular reference.)

2.4 Re-interpreting

If limitation (A) doesn't apply, we move on to (B), which essentially stipulates that the exclusion in (b)(1) should be re-interpreted for each spouse separately *as if they were not married*; the final exclusion is then the sum of the two sub-computations.

> **(B) Other joint returns**
> If such spouses do not meet the requirements of subparagraph (A), the limitation under paragraph (1) shall be the sum of the limitations under paragraph (1) to which each spouse would be entitled if such spouses had not been married. For purposes of the preceding sentence, each spouse shall be treated as owning the property during the period that either spouse owned the property.

We thus observe that the law is re-entrant and can call itself recursively under different conditions. This is indicated here by the use of the conditional tense, i.e. "would".

2.5 Out-of-order backpatching

In another striking example, (3) cancels the whole exclusion (a) under certain conditions.

> **(3) Application to only 1 sale or exchange every 2 years**
> Subsection (a) shall not apply to any sale or exchange by the taxpayer if, during the 2-year period ending on the date of such sale or exchange, there was any other sale or exchange by the taxpayer to which subsection (a) applied.

Paragraph (3) comes a little further down; a key takeaway is that, for a piece of law, one must process the *entire* document; barring that, the reader might be missing a crucial limitation that only surfaces much later in the text.

2.6 A final example

Paragraph (4) concerns the specific case of a surviving spouse that sells the residence within two years of the death of their spouse, knowing that the conditions from (A) applied (i.e. "returned true") *right before the date of the death.*

> **(4) Special rule for certain sales by surviving spouses**
> In the case of a sale or exchange of property by an unmarried individual whose spouse is deceased on the date of such sale, paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if such sale occurs not later than 2 years after the date of death of such spouse and the requirements of paragraph (2)(A) were met immediately before such date of death.

Paragraph (4) combines several of the complexities we saw above. It not only back-patches (1), but also recursively calls (2)(A) under a different context, namely, executing (2)(A) at a *previous date* in which the situation was different. In functional programming lingo, one might say that there is a hidden lambda in (2)(A), that binds the date of the sale.

2.7 Formal insights on legal logic

We have now seen how legal statutes are written, the thought process they exhibit, and how one is generally supposed to interpret them. We wish to emphasize that the concepts described are by no means specific to tax law or the US legal system: we found the exact same patterns in other parts of US law and non-US legal systems. Section 121 contains many more paragraphs; however, the first few we saw above are sufficient to illustrate the challenges in formally describing the law.

The main issue in modeling legal texts therefore lies in their underlying logic, which relies heavily on the pattern of having a default case followed by exceptions. This nonmonotonic logic is known as *default logic* [40]. Several refinements of default logic have been proposed over time; the one closest to the purposes of the law is known as prioritized default logic [5], wherein default values are guarded by justifications, and defaults can be ordered according to their relative precedence. Lawsky [24] argues that this flavor of default logic is the best suited to expressing the law. We concur, and adopt prioritized default logic as a foundation for **Catala**.

In default logic, formulas include defaults, of the form $a : \vec{b_i}/c$, wherein: if formula a holds; if the $\vec{b_i}$ are consistent with the set of known facts; then c holds. Prioritized logic adds a strict partial order over defaults, to resolve conflicts when multiple defaults may be admissible at the same time.

The main design goal of **Catala** is to provide the design and implementation of a language tailored for the law, using default logic as its core building block, both in its syntax and semantics. **Catala** thus allows lawyers to express the general case / exceptions pattern naturally. We now informally present **Catala**.

3 A CATALA TUTORIAL

Our introduction to legal texts in Section 2 mixes informal, high-level overviews of what each paragraph intends to express, along with excerpts from the law itself. Our English prose is too informal to express anything precisely; but "legalese" requires a high degree of familiarity with the law to successfully grasp all of the limitations and compute what may or may not be applicable to a given taxpayer's situation.

We now introduce **Catala** by example, and show how the subtleties of each paragraph can be handled unambiguously and clearly by **Catala**. Our guiding principle is twofold: we want to formally express the intended meaning without being obscured by the verbosity of legal prose; yet, we wish to remain close to the legal text, so that the formal specification remains in close correspondence with the reference legal text, and can be understood by lawyers. **Catala** achieves this with a dedicated surface language that follows the thought process of legal minds.

3.1 Metadata: turning implicits into explicits

Legal prose is very dense, and uses a number of concepts without explicitly defining them in the text. For instance, in Section 121, the notion of time period is implicit, and so are the various types of tax returns one might file (individual or joint). Furthermore, entities such as the taxpayers (whom we will call "Person 1" and "Person 2") need to be materialized. Finally, for each one of those entities, there are a number of inputs that are implicitly referred to throughout the legal statute, such as: time periods in which each Person was occupying the residence as their primary home; whether there was already a sale in the past two years; and many more, as evidenced by the myriad of variables involved in (i)-(iii).

Our first task when transcribing legal prose into a formal **Catala** description is thus to enumerate all structures, entities and variables relevant to the problem at stake. We provide the definitions and relationships between variables later on. This is a conscious design choice of **Catala**: before even talking about *how* things are computed, we state *what* we are talking about. In doing so, we mimic the behavior of lawyers, who are able to infer what information is relevant based on the legal text. We call this description of entities the *metadata*.

```
declaration structure Period:
  data start content date
  data end content date

declaration structure PersonalData:
  data property_ownership content collection Period
  data property_usage_as_principal_residence content collection Period
  ...

declaration scope Section121SinglePerson:
  context gain_from_sale_or_exchange_of_property content money
  context personal content PersonalData
  context requirements_ownership_met condition
  context requirements_usage_met condition
  context requirements_met condition
  ...

  context amount_excluded_from_gross_income_uncapped content money
  context amount_excluded_from_gross_income content money

```

```
21   context aggregate_periods_from_last_five_years content duration
22     depends on collection Period
```

The snippet above shows an excerpt from Section 121's metadata. The first two **declaration**s declare product types via the **structure** keyword. **Catala** features a number of built-in types, such as **date**, **condition** (i.e. a boolean) and **money**. The higher-kinded type **collection** is also built-in. The type **Period** contains two fields, **start** and **end**.

A word about aesthetics: while programmers generally prize compactness of notation, advocating e.g. point-free-styles or custom operators, non-experts are for the most part puzzled by compact notations. Our surface syntax was designed in collaboration with lawyers, who confirmed that the generous keywords improve readability, thus making **Catala** understandable by legal minds.

Line 10 declares **Section121SinglePerson**, a **scope**. A key technical device and contribution of **Catala**, scopes allow the programmer to follow the law's structure, revealing the implicit modularity in legal texts. Scopes are declared in the metadata section: the **context** keyword indicates that the value of the field might be determined later, *depending on the context*. Anticipating on Section 4, the intuition is that **scope**s are functions and **context**s are their parameters and local variables.

The main purpose of Section 121 is to talk about the gain that a person derived from the sale of their residence (line 11), of type **money**. Paragraph (a) implicitly assumes the existence of time periods of ownership and usage of the residence; we materialize these via the **personal** field which holds two **collection Period**s. These in turn allow us to define whether the ownership and usage requirements are met (of type **condition**, lines 13-14). A further condition captures whether *all* requirements are met (line 15). The outcome of the law is the amount that can be excluded from the gross income, of type **money** (line 19). (The need for an intermediary variable at line 18 becomes apparent in Section 3.3.) A local helper computes the aggregate number of days in a set of time periods; the helper takes a single argument of type **collection Period** (line 21) and, being a local closure, can capture other **context** variables.

3.2 Scopes and contexts: declarative rules and definitions

We now continue with our formalization of (a) and define the context-dependent variables, as well as the relationships between them. **Catala** is declarative: the user relates **context** variables together, via the **definition** keyword, or the **rule** keyword for **condition**s. We offer separate syntax for two reasons. First, for legal minds, conditions and data are different objects and reflecting that in the surface syntax helps with readability. Second, there is a core semantic difference: booleans (conditions) are false by default in the law; however, other types of data have no default value. Internally, **Catala** desugars **rule**s to **definition**s equipped with a default value of **false** (§4.1).

```
1  scope Section121SinglePerson:
2    rule requirements_ownership_met under condition
3      aggregate_periods_from_last_five_years of personal.property_ownership >=^ 730 day
4    consequence fulfilled
5
6    rule requirements_usage_met under condition
7      aggregate_periods_from_last_five_years of
8        personal.property_usage_as_principal_residence >=^ 730 day
9    consequence fulfilled
10
11   rule requirements_met under condition
12     requirements_ownership_met and requirements_usage_met
13   consequence fulfilled
14
15   definition amount_excluded_from_gross_income_uncapped equals
```

```
16      if requirements_met then gain_from_sale_or_exchange_of_property
17      else $0
```

Lines 2-4 capture the ownership requirement, by calling the helper **`aggregate_periods_...`** with argument **`property_ownership`**, a previously-defined context variable. (The full definition of the helper, which involves another context variable for the date of sale, is available in Appendix A.) Paragraph (a) states "for periods aggregating 2 years or more": for the purposes of Section 121, and as defined in Regulation 1.121-1(c)(1), a year is always 365 days. **Catala** resolves the ambiguity by simply not offering any built-in notion of yearly duration, and thus makes the law clearer. The **`^`** suffix of the comparison operator **`>=^`** means that we are comparing durations.

The ownership requirement is "fulfilled" (i.e. defined to **`true`**) under a certain condition. This is our first taste of prioritized default logic expressed through the syntax of **Catala**: the built-in default, set to **`false`**, is overridden with a rule that has higher priority. This is a simple case and more complex priorities appear in later sections. However, this example points to a key insight of **Catala**: rather than having an arbitrary priority order resolved at run-time between various **`rule`**s, we encode priorities statically in the surface syntax of the language, and the pre-order is derived directly from the syntax tree of rules and definitions. We explain this in depth later on (Section 4.1).

Similarly, lines 6-9 define the usage requirement using the **`rule`** keyword to trigger a **`condition`**: the value of **`requirements_usage_met`** is **`false`** unless the boolean expression at lines 7-8 is **`true`**. One legal subtlety, implicit in (a), is that ownership and usage periods do not have to overlap. The **Catala** program makes this explicit by having two collections of time periods.

The requirements are met if both ownership and usage requirements are met (lines 11-13). In that case, the income gain can be excluded from the income tax (lines 15-17). The latter is defined via the **`definition`** keyword, as **`rule`** is reserved for booleans.

We have now formalized Paragraph (a) in **Catala**. At this stage, if the user fills out the remaining inputs, such as the gain obtained from the sale of the property, and the various time periods, the interpreter automatically computes the resulting value for the amount to be excluded from the gross income. The interpreter does so by performing a control-flow analysis and computing a topological sort of assignments. Cycles are rejected, since the law is not supposed to have dependency cycles. (Section 4 describes the full semantics of the language.)

We note that a single sentence required us to explicitly declare otherwise implicit concepts, such as the definition of a year; and to clarify ambiguities, such as whether time periods may overlap. With this concise example, we observe that the benefits of formalizing a piece of law are the same as formalizing any piece of critical software: numerous subtleties are resolved, and non-experts are provided with an explicit, transparent executable specification that obviates the need for an expert legal interpretation of implicit semantics.

3.3 Split scopes: embracing the structure of the law

We now move on to limitations (§2.2). A key feature of **Catala** is that it enables a literate programming style [21], where each paragraph of law is immediately followed by its **Catala** transcription. Now that we're done with (a), we insert a textual copy of the legal prose for (b), then proceed to transcribe it in **Catala**.

```
1  scope Section121SinglePerson:
2    definition gain_cap equals $250,000
3
4    definition amount_excluded_from_gross_income equals
5      if amount_excluded_from_gross_income_uncapped >=$ gain_cap then
6        gain_cap
```

```
7       else
8         amount_excluded_from_gross_income_uncapped
```

In Paragraph (b)(1), the law overwrites the earlier definition from (a) and re-defines it to be capped by $250,000. In line with our earlier design choices, we rule out confusion and rely on the auxiliary variable (the "uncapped" variant), to then compute the final amount excluded from the gross income (lines 4-8). Out-of-order definitions that are provided at a later phase in the source are an idiomatic pattern in **Catala**.

3.4 Complex usage of the default calculus; exceptions

Before making any further progress, we need to introduce new entities to take into account the fact that we may now possibly be dealing with a joint return. We introduce a new abstraction or, in **Catala** lingo, scope: **`Section121Return`**.

```
1  declaration structure CoupleData:
2    data personal1 content PersonalData
3    data personal2 content PersonalData
4
5  declaration enumeration ReturnType:
6    -- SingleReturn content PersonalData
7    -- JointReturn content CoupleData
8
9  declaration scope Section121Return:
10   context return_data content ReturnType
11   context person1 scope Section121SinglePerson
12   context person2 scope Section121SinglePerson
13   context gain_cap content money
14   ...
```

We follow the out-of-order structure of the law; only from here on do we consider the possibility of a joint return. Having introduced a new level of abstraction, we need to relate the **`return_type`** to the persons involved. We do so by introducing new equalities, of which we show the first one.

```
1  scope Section121Return:
2    definition person1.personal equals match return_data with
3    -- SingleReturn of personal1 : personal1
4    -- JointReturn of couple : couple.personal1
5    ...
```

Having set up a proper notion of joint return, we now to turn our attention to (b)(2)(A) (§2.3).

```
1  scope Section121Return:
2    definition gain_cap equals person1.gain_cap
3
4    rule paragraph_A_applies under condition
5      (return_data is JointReturn) and
6      (person1.requirements_ownership_met or person2.requirements_ownership_met) and
7      (person1.requirements_usage_met and person2.requirements_usage_met) and
8      (not (paragraph_3_applies of person1.other_section_121a_sale)) and
9      (not (paragraph_3_applies of person2.other_section_121a_sale))
10   consequence fulfilled
11
12   exception
13   definition gain_cap under condition
```

```
14      paragraph_A_applies
15    consequence equals $500,000
```

Until now, the gain cap was defined to be that of the taxpayer, that is, Person 1 (line 2). We now need to determine whether the conditions from Paragraph (A) apply (line 4). To that end, we introduce an intermediary variable, **paragraph_A_applies**. This variable will be used later on for (B), whose opening sentence is "if such spouses do not meet the requirements of (A)".

We now introduce the notion of **exception** (line 12). In **Catala**, if, at run-time, more than a single applicable definition for any **context** variable applies, program execution aborts with a fatal error. In the absence of the **exception** keyword, and in the presence of a joint return that satisfies paragraph (A), the program would be statically accepted by **Catala**, but would be rejected at run-time: there are two definitions for **gain_cap**, both their conditions hold (**true** and **paragraph_A_applies**), and there is no priority ordering indicating how to resolve the conflict. The **exception** keyword allows solving this very issue. The keyword indicates that, in the pre-order of definitions, the definition at line 12 has a higher priority than the one at 2.

Generally, **Catala** allows an arbitrary tree of definitions each refined by exceptions, including exceptions to exceptions (which we have encountered in the law); the rule of thumb remains: only one single definition should be applicable at a time, and any violation of that rule indicates either programmer error, or a fatal flaw in the law.

3.5 Recapping

Modeling the law is labor-intensive, owing to all of the implicit assumptions present in what is seemingly "just legalese". In our experience, this process is best achieved through pair-programming, in which a **Catala** expert transcribes a statute with the help of a legal expert. We thus stop here our **Catala** tutorial and defer the full modelization of Section 2 to Appendix A. Briefly, modeling (B) requires introducing a new scope for a two-pass processing that models the re-entrancy ("if such spouses had not been married"). Modeling the forward-reference to (3) requires introducing a helper **paragraph_3_applies** whose **definition** is provided later on, after Paragraph (3) has been suitably declared (line 8, above).

As this tutorial wraps up, we look back on all of the language features we presented. While **Catala** at first glance resembles a lambda-calculus with heavy syntactic sugar, diving into the subtleties of the law exhibits the need for two features that are not generally found in lambda-calculi. First, we allow the user to define context variables through a combination of an (optional) default case, along with an arbitrary number of special cases, either prioritized or non-overlapping. The theoretical underpinning of this feature is the *prioritized default calculus*. Second, the out-of-order nature of definitions means that **Catala** is entirely declarative, and it is up to the **Catala** compiler to compute a suitable dependency order for all the definitions in a given program.

Fortunately, the law does not have general recursion, meaning that we do not need to compute fixed points, and do not risk running into circular definitions. Hence, our language is not Turing-complete, purposefully. We thus focus on what makes **Catala** special: its default calculus.

4 FORMALIZING CATALA

We now formally introduce the semantics and compilation of **Catala**. To that end, we describe a series of compilation steps: we desugar the concrete syntax to a *scope language*; we define the semantics of scopes via a translation to a *default calculus*; we then finally compile the *default calculus* to a language equipped with exceptions, such as OCaml. This last part is where the most crucial compilation steps occur: we prove its soundness via a mechanization in the F* proof assistant.

Scope name	S			
Sub-scope instance	S_n			
Location	ℓ	::=	x	scope variable
		|	$S_n[x]$	sub-scope variable
Type	τ	::=	`bool` | `unit`	base types
		|	$\tau \to \tau$	function type
Expression	e	::=	x | `true` | `false` | `()`	variable, literals
		|	$\lambda\ (x : \tau).\ e$ | $e\ e$	λ-calculus
		|	ℓ	location
		|	d	default term
Default	d	::=	$\langle e^* \mid e :\!- e \rangle$	default term
		|	$\circledast$	conflict error term
		|	$\varnothing$	empty error term
Atom	a	::=	`def` $\ell : \tau = \langle e^* \mid e :\!- e \rangle$	location definition
		|	`call` S_n	sub-scope call
Scope declaration	σ	::=	`scope` $S : a^*$	
Program	P	::=	σ^*	

Fig. 1. The scope language, our first intermediate representation

4.1 From Catala to the scope language

The scope language resembles **Catala**'s user-facing language: the notion of scope is still present; **`rule`**s and **`definition`**s remain, via a unified **`def`** declaration devoid of any syntactic sugar. Perhaps more importantly, definitions are provided in-order and our usage of default calculus becomes clear.

Figure 1 presents the syntax of the scope language. We focus on the essence of **Catala**, i.e. how to formalize a language with default calculus at its core; to that end, and from this section onwards, we omit auxiliary features, such as data types, in order to focus on a core calculus.

To avoid carrying an environment, a reference to a sub-scope variable, such as **`person1.personal`** earlier, is modeled as a reference to a sub-scope annotated with a unique identifier, such as **`Section121SinglePerson`**$_1$.personal. Therefore, locations are either a local variable x, or a sub-scope variable, of the form $S_n[x]$.

Types and expressions are standard, save for default terms d of the form $\langle \vec{e_i} \mid e' :\!- e'' \rangle$ which informally reduce as follows. Each of the *exceptions* e_i is evaluated; if two or more are valid (i.e. not of the form $\varnothing$), a conflict error $\circledast$ is raised. If exactly one exception e_i is valid, the final result is e_i. If no exception is valid, and e' evaluates to `true` the final result is e''. If no exception is valid, and e' evaluates to `false`, the final result is $\varnothing$. We provide a full formal semantics of default terms in §4.2.

The syntactic form $\langle \vec{e_i} \mid e' :\!- e'' \rangle$ encodes a *static* tree of priorities, baking the pre-order directly in the syntax tree of each definition. We thus offer a restricted form of prioritized default logic, in which each definition is its own world, equipped with a static pre-order.

Atoms a either define a new location, gathering all default cases and exceptions in a single place; or, rules indicate that a sub-scope needs to be called to compute further definitions.

We now explain how to desugar the surface syntax, presented in Section 3, to this scope language.

	Syntactic sugar	**... rewrites to**
(i)	`rule X under cond. Y cons. fulfilled`	`label` $\mathtt{L}_X$ `def. X equals false` (inserted once) `exception` $\mathtt{L}_X$ `def. X under cond. Y cons. equals true`
(ii)	`def. X equals Y`	`def. X under cond. true cons. equals Y`
(iiia)	`def. X . . .` (multiple definitions of `X`, no exceptions)	`label` $\mathtt{L}_X$ `def. X nodefault` (inserted once) `exception` $\mathtt{L}_X$ `def. X . . .`
(iiib)	`def. X . . .` (single definition of `X`)	`label` $\mathtt{L}_X$ `def. X . . .`
(iv)	`exception def. X`	`exception` $\mathtt{L}_X$ `def. X`

Table 1. Desugaring the surface language into an explicit surface syntax

$$\textsc{D-Label}\ \frac{\mathsf{lookup}(X, L) = C, D, \vec{L}_i \qquad X, L_i \rightsquigarrow d_i}{X, L \rightsquigarrow \langle \vec{d}_i \mid C :\!- D \rangle}$$

$$\textsc{D-EntryPoint}\ \frac{X, L \rightsquigarrow \langle \vec{d}_i \mid C :\!- D \rangle}{\mathsf{label}\ L\ \mathsf{definition}\ X\ \ldots \rightsquigarrow \mathsf{def}\ X = \langle \vec{d}_i \mid C :\!- D \rangle}$$

Fig. 2. Building the default tree and translating surface definitions

Syntactic sugar. Table 1 presents rewriting rules, whose transitive closure forms our desugaring. These rules operate within the surface language; Table 1 abbreviates surface-level keywords for easier typesetting.

In its full generality, **Catala** allows exceptions to definitions, followed by an arbitrary nesting of exceptions to exceptions. This is achieved by a label mechanism: all exceptions and definitions are *labeled*, and each exception refers to the definition or exception it overrides. Exceptions to exceptions are actually found in the law, and while we spared the reader in our earlier tutorial, we have found actual use-cases where this complex scenario was needed. Exceptions to exceptions remain rare; the goal of our syntactic sugar is to allow for a more compact notation in common cases, which later gets translated to a series of fully labeled definitions and exceptions.

After desugaring, definitions and exceptions form a forest, with exactly one root **`definition`**s node for each variable X, holding an n-ary tree of **`exception`** nodes.

We start with the desugaring of **`rule`** which, as mentioned earlier, is a boolean definition with a base case of **`false`** (i). Definitions without conditions desugar to the trivial `true` condition (ii).

The formulation of (iiia) allows the user to provide multiple definitions for the same variable X without labeling any of them; thanks to (iiia), these are implicitly understood to be a series of exceptions without a default case. The surface syntax always requires a default to be provided; internally, the **`nodefault`** simply becomes **`condition true consequence equals`** $\varnothing$.

We provide another alternative to the fully labeled form via (iiib); the rule allows the user to provide a single base definition, which may then be overridden via a series of exceptions. To that end, we introduce a unique label L_X which un-annotated exceptions are understood to refer to (iv).

Materializing the default tree. Equipped with our default expressions d, we show how to translate a scattered series of **Catala** definitions into a single **`def`** rule from the scope language. We write $X, L \rightsquigarrow d$, meaning that the definition of X labeled L, along with all the (transitive) exceptions to L, collectively translate to d. We use an auxiliary helper $\mathsf{lookup}(X, L) = C, D, \vec{L}_i$, meaning that at label L, under condition C, X is defined to be D, subject to a series of exceptions labeled L_i.

Type	τ	::=	`bool` | `unit`	boolean and unit types
		|	$\tau \to \tau$	function type
Expression	e	::=	x | s | `true` | `false` | `()`	variable, top-level name, literals
		|	$\lambda\ (x : \tau).\ e$ | $e\ e$	λ-calculus
		|	d	default term
Default	d	::=	$\langle e^* \mid e :\!- e \rangle$	default term
		|	$\circledast$	conflict error term
		|	$\varnothing$	empty error term
Top-level declaration	σ	::=	`let` $s = e$	
Program	P	::=	σ^*	

Fig. 3. The default calculus, our second intermediate representation

ConflictError
$\Gamma \vdash \circledast : \tau$

EmptyError
$\Gamma \vdash \varnothing : \tau$

T-Default
$$\frac{\Gamma \vdash e_i : \tau \qquad \Gamma \vdash e_{\text{just}} : \texttt{bool} \qquad \Gamma \vdash e_{\text{cons}} : \tau}{\Gamma \vdash \langle e_1, \ldots, e_n \mid e_{\text{just}} :\!- e_{\text{cons}} \rangle : \tau}$$

Fig. 4. Typing rules for the default calculus

Rule D-Label performs the bulk of the work, and gathers the exception labels L_i; each of them translates to a default expression d_i, all of which appear on the left-hand side of the resulting translation; if all of the d_i are empty, the expression evaluates to D guarded under condition C. As an illustration, if no exceptions are to be found, the translation is simply $\langle \mid C :\!- D \rangle$. Finally, rule D-EntryPoint states that the translation starts at the root **`definition`** nodes.

Reordering definitions. Our final steps consists in dealing with the fact that `def`s remain unordered. To that end, we perform two topological sorts. First, for each scope S, we collect all definitions and re-order them according to a local dependency relation $\rightarrow$:

$$\begin{cases} y \to x & \text{if def } x = \ldots y \ldots \\ S_n \to x & \text{if def } x = \ldots S_n[y] \ldots \\ y \to S_n & \text{if def } S_n[x] = \ldots y \ldots \end{cases}$$

After re-ordering, we obtain a scope S where definitions can be processed linearly. Sub-scope nodes of the form S_n become `call`s, to indicate the position at which the sub-scope computation can be performed, i.e. once its parameters have been filled and before its outputs are needed.

We then topologically sort the scopes themselves to obtain a linearized order. We thus move from a declarative language to a functional language where programs can be processed in evaluation order. In both cases, we detect the presence of cycles, and error out. General recursion is not found in the law, and is likely to indicate an error in modeling. Bounded recursion, which we saw in Section 2.2, can be manually unrolled to make it apparent.

4.2 From the scope language to a default calculus

For the next step of our translation, we remove the scope mechanism, replacing `def`s and `call`s with regular λ-abstractions and applications. The resulting language, a core lambda calculus equipped only with default terms, is the *default calculus* (Figure 3). The typing rules of the default calculus are standard (Figure 4); we note that the error terms from the default calculus are polymorphic.

Values	v	::=	$\lambda\ (x : \tau)\ .\ e$	functions
		|	$\texttt{true} \mid \texttt{false}$	booleans
		|	$\circledast \mid \varnothing$	errors
Evaluation contexts	C_λ	::=	$\cdot\ e \mid v\ \cdot$	function application evaluation
		|	$\langle v^* \mid \cdot :- e \rangle$	default justification evaluation
		|	$\langle v^* \mid \texttt{true} :- \cdot \rangle$	default consequence evaluation
	C	::=	C_λ	regular contexts
		|	$\langle v^*, \cdot, e^* \mid e :- e \rangle$	default exceptions evaluation

Fig. 5. Evaluation contexts for the default calculus

$$\textsc{D-Context}\ \frac{e \longrightarrow e' \qquad e' \notin \{\circledast, \varnothing\}}{C[e] \longrightarrow C[e']} \qquad \textsc{D-Beta}\ (\lambda\ (x : \tau)\ .\ e)\ v \longrightarrow e[x \mapsto v] \qquad \textsc{D-ContextConflictError}\ \frac{e \longrightarrow \circledast}{C[e] \longrightarrow \circledast}$$

$$\textsc{D-ContextEmptyError}\ \frac{e \longrightarrow \varnothing}{C_\lambda[e] \longrightarrow \varnothing} \qquad \textsc{D-DefaultTrueNoExceptions}\ \langle \varnothing, \ldots, \varnothing \mid \texttt{true} :- v \rangle \longrightarrow v \qquad \textsc{D-DefaultFalseNoExceptions}\ \langle \varnothing, \ldots, \varnothing \mid \texttt{false} :- e \rangle \longrightarrow \varnothing$$

$$\textsc{D-DefaultOneException}\ \frac{v \neq \varnothing}{\langle \varnothing, \ldots, \varnothing, v, \varnothing, \ldots, \varnothing \mid e_1 :- e_2 \rangle \longrightarrow v} \qquad \textsc{D-DefaultExceptionsConflict}\ \frac{v_i \neq \varnothing \qquad v_j \neq \varnothing}{\langle \ldots, v_i, \ldots, v_j, \ldots \mid e_1 :- e_2 \rangle \longrightarrow \circledast}$$

Fig. 6. Reduction rules for the default calculus

Reduction rules. We present small-step operational semantics, of the form $\boxed{e \longrightarrow e'}$. For efficiency, we describe reduction under a context, using a standard notion of value (Figure 5), which includes our two types of errors, $\circledast$ and $\varnothing$. We intentionally distinguish regular contexts C_λ from general contexts C.

Figure 6 presents the reduction rules for the default calculus. Rule D-Context follows standard call-by-value reduction rules for non-error terms; D-Beta needs no further comment. $\circledast$ is made fatal by D-ContextConflictError: the reduction aborts, under *any context* C. The behavior of $\varnothing$ is different: such an error propagates only up to its enclosing "regular" context C_λ; this means that such an $\varnothing$-error can be caught, as long as it appears in the exception list of an enclosing default expression. Therefore, we now turn our attention to the rules that govern the evaluation of default expressions.

If no exception is valid, i.e. if the left-hand side contains only $\varnothing$s; and if after further evaluation, the justification is `true` for the consequence v, then the whole default reduces to v (D-DefaultTrueNoExceptions). If no exception is valid, and if the justification is `false`, then we do not need to evaluate the consequence, and the default is empty, i.e. the expression reduces to $\varnothing$. If *exactly* one exception is a non-empty value v, then the default reduces to v. In that case, we evaluate neither the justification or the consequence (D-DefaultOneException). Finally, if two or more exceptions are non-empty, we cannot determine the priority order between them, and abort program execution (D-DefaultExceptionsConflict).

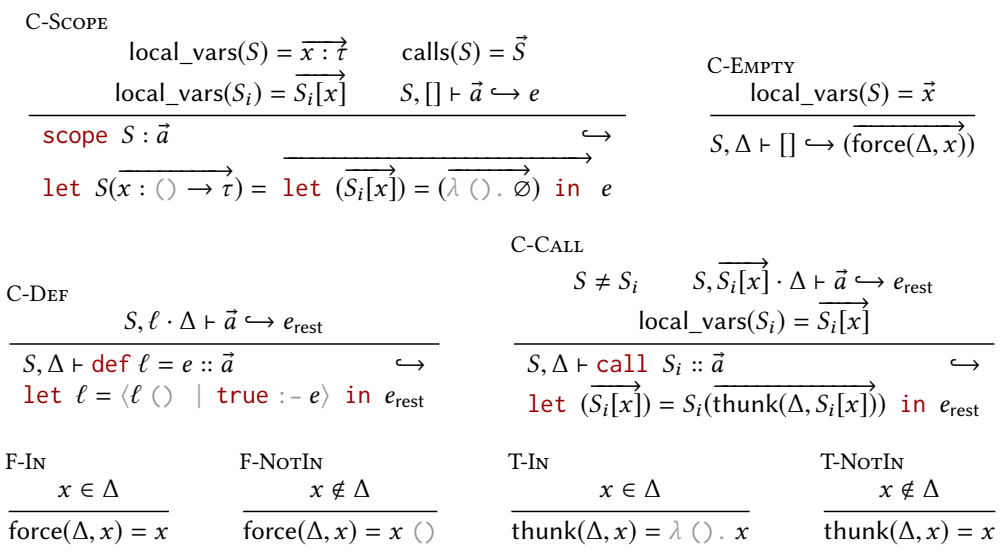

Fig. 7. Compiling the scope language to a default calculus

Compiling the scope language. We succinctly describe the compilation of the scope language to the default calculus in Figure 7. Our goal is to get rid of scopes in favor of regular lambda-abstractions, all the while preserving the evaluation semantics; incorrect order of evaluation might lead to propagating premature errors (i.e. throwing exceptions too early).

We assume for simplicity of presentation that we are equipped with tuples, where $(x_1, \ldots, x_n)$ is concisely written $(\vec{x})$. We also assume that we are equipped with let-bindings, of the form `let` $(x_1, \ldots, x_n) = e$, for which we adopt the same concise notation. Given a scope S made up of atoms $\vec{a}$, local_vars(S) returns all variables x for which `def` $x \in \vec{a}$. Note that this does not include rules that override sub-scope variables, which are of the form `def` $S_n[x]$. We now turn to our judgement, of the form $\boxed{S, \Delta \vdash a \hookrightarrow e}$, to be read as "in the translation of scope S, knowing that variables in Δ have been forced, scope rule r translates to default calculus expression e".

A scope S with local variables $\vec{x}$ compiles to a function that takes an n-uple $(\vec{x})$ containing *potential overrides* for all of its context variables (C-Scope). In the translation, each x therefore becomes a thunk, so as to preserve reduction semantics: should the caller decide to leave a local variable x_i to be $\varnothing$, having a thunk prevents D-ContextEmptyError from triggering and returning prematurely. Rule C-Scope performs additional duties. For each one of the sub-scopes S_i used by S, we set all of the arguments to S_i, denoted $\overrightarrow{S_i[x]}$, to be a thunked $\varnothing$ to start with.

Advancing through the scope S, we may encounter definitions or calls. For definitions (C-Def), we simply insert a shadowing let-binding, and record that ℓ has been forced by extending Δ. Whether ℓ is of the form x or $S_n[x]$, we know that the previous binding was thunked, since our previous desugaring guarantees that any variable ℓ now has a single definition. The rewritten default expression gives the caller-provided argument higher precedence; barring any useful information provided by the caller, we fall back on the existing definition e. This key step explains how our law-centric syntax, which allows caller scopes to override variables local to a callee scope, translates to the default calculus.

Type	τ	::=	`bool` | `unit`	boolean and unit types
		|	$\tau \to \tau$	function type
		|	`list` τ	list type
		|	`option` τ	option type
Expression	e	::=	x | s | `true` | `false` | `()`	variable, top-level name, literals
		|	$\lambda\ (x : \tau).\ e$ | $e\ e$	λ-calculus
		|	`None` | `Some` e	option constructors
		|	`match` e `with`	option destructuring
			| `None` $\to e$ | `Some` $x \to e$	
		|	`[`e^*`]` | `fold_left` $e\ e\ e$	list introduction and fold
		|	`raise` ε | `try` e `with` $\varepsilon \to e$	exceptions
Exception	ε	::=	$\varnothing$	empty exception
		|	$\circledast$	conflict exception
Top-level declaration	σ	::=	`let` $s = e$	
Program	P	::=	σ^*	

Fig. 8. The enriched lambda calculus, our final translation target

For calls (C-Call), we ensure all of the arguments are thunked before calling the sub-scope; the return tuple contains *forced* values, which we record by extending Δ with all $\overrightarrow{S_i[x]}$. The premise $S \neq S_i$ captures the fact that recursion is not allowed.

Finally, after all rules have been translated and we are left with nothing but the empty list [] (C-Empty), we simply force all scope-local variables $\vec{x}$ and return them as a tuple.

4.3 From the default calculus to a lambda calculus

While sufficient to power the **Catala** surface language, the relatively simple semantics of our default calculus are non-standard. We now wish to compile to more standard constructs found in existing programming languages. We remark that the reduction semantics for default terms resembles that of exceptions: empty-default errors propagate ("are thrown") only up to the enclosing default term ("the try-catch"). Confirming this intuition and providing a safe path from **Catala** to existing programming languages, we now present a compilation scheme from the default calculus to a lambda calculus enriched with a few standard additions: lists, options and exceptions.

Figure 8 shows the syntax of the target lambda calculus. In order to focus on the gist of the translation, we introduce `list` and `option` as special, built-in datatypes, rather than a full facility for user-defined inductive types. For those reasons, we offer the minimum set of operations we need: constructors and destructors for `option`, and a left fold for `lists`. We omit typing and reduction rules, which are standard. The only source term that does not belong to the target lambda calculus is the default term $\langle \vec{e} \mid e_{\text{just}} :\!- e_{\text{cons}} \rangle$. Hence, translating this term is the crux of our translation.

Our translation is of the form $\boxed{e \rightrightarrows e'}$, where e is a term of the default calculus and e' is a term of the target lambda calculus. Figure 9 presents our translation scheme. The semantics of default terms are intertwined with those of $\varnothing$ and $\circledast$. The translation of $\varnothing$ and $\circledast$ is simple: both compile to exceptions in the target language. We now focus on C-Default, which deals with default terms. As a warm-up, we start with a special case: $\langle \mid e_{\text{just}} :\!- e_{\text{cons}} \rangle$. We translate this term to `if` e_{just} `then` e_{cons} `else raise` $\varnothing$, which obeys the evaluation semantics of both D-DefaultTrueNoExceptions and D-DefaultFalseNoExceptions. This simple example serves as a

$$\textsc{C-Default}\quad \frac{e_1 \rightrightarrows e_1' \quad \cdots \quad e_n \rightrightarrows e_n' \quad e_{\text{just}} \rightrightarrows e_{\text{just}}' \quad e_{\text{cons}} \rightrightarrows e_{\text{cons}}'}{\begin{array}{l}\langle e_1, \ldots, e_n \mid e_{\text{just}} \text{ :- } e_{\text{cons}} \rangle \rightrightarrows \\ \texttt{let } r_{\text{exceptions}} = \texttt{process\_exceptions } [\lambda\, \_ \to e_1'; \ldots; \lambda\, \_ \to e_n'] \texttt{ in} \\ \texttt{match } r_{\text{exceptions}} \texttt{ with Some } e' \to e' \mid \texttt{None} \to \texttt{if } e_{\text{just}}' \texttt{ then } e_{\text{cons}}' \texttt{ else raise } \varnothing\end{array}}$$

$$\textsc{C-EmptyError}\quad \varnothing \rightrightarrows \texttt{raise } \varnothing \qquad \textsc{C-ConflictError}\quad \circledast \rightrightarrows \texttt{raise } \circledast \qquad \textsc{C-Var}\quad x \rightrightarrows x$$

$$\textsc{C-Literal}\quad \frac{e \in \{(), \texttt{true}, \texttt{false}\}}{e \rightrightarrows e}$$

$$\textsc{C-Abs}\quad \frac{e \rightrightarrows e'}{\lambda\,(x : \tau).\ e \rightrightarrows \lambda\,(x : \tau).\ e'} \qquad \textsc{C-App}\quad \frac{e_1 \rightrightarrows e_1' \quad e_2 \rightrightarrows e_2'}{e_1\ e_2 \rightrightarrows e_1'\ e_2'}$$

Fig. 9. Translation rules from default calculus to lambda calculus

```
process_exceptions : list (unit → τ) → option τ
process_exceptions ≜ fold_left (λ (a : option τ) (e' : unit → τ).
    let e' : τ = try Some (e'()) with ∅ → None in
    match (a, e') with
      | (None, e') → e'
      | (Some a, None) → Some a
      | (Some a, Some e') → raise ⊛ ) None
```

Fig. 10. `process_exceptions` translation helper

blueprint for the more general case, which has to take into account the list of exceptions $\vec{e}$, and specifically count how many of them are $\varnothing$.

In the general case, C-Default relies on a helper, `process_exceptions`; each exception is translated, thunked, then passed to the helper; if the helper returns `Some` , exactly one exception did not evaluate to $\varnothing$; we return it. If the helper returns `None`, no exception applied, and we fall back to the simple case we previously described.

We now review **`process_exceptions`** defined in Figure 10. It folds over the list of exceptions, with the accumulator initially set to `None`, meaning no applicable exception was found. Each exception is forced in order, thus implementing the reduction semantics of the default calculus. The accumulator transitions from `None` to `Some` if a non-empty exception is found, thus implementing a simple automaton that counts the number of non-empty exceptions. If two non-$\varnothing$ exceptions are found, the automaton detects an invalid transition and aborts with a non-catchable $\circledast$.

4.4 Certifying the translation

The translation from scope language to default calculus focuses on turning scopes into the lambda-abstractions that they truly are underneath the concrete syntax. This is a mundane transformation, concerned mostly with syntax. The final step from default calculus to lambda calculus with exceptions is much more delicate, as it involves compiling custom evaluation semantics. To rule out

Fig. 11. Translation correctness theorems. A shows a regular simulation; B shows our variant of the theorem.

any errors in the most sensitive compilation step of **Catala**, we formally prove our translation correct, using F⋆ [1, 30, 45], a proof assistant based on dependent types, featuring support for semi-automated reasoning via the SMT-solver Z3 [9].

Correctness statement. We wish to state two theorems about our translation scheme. First, typing is preserved: if $e \rightrightarrows e'$ and $\varnothing \vdash e : \tau$, then $\varnothing \vdash e' : \tau$ in the target lambda calculus. Second, we want to establish a simulation result, i.e. the compiled program e_λ faithfully simulates a reduction step from the source language, using n steps in the target language.

The usual simulation result is shown in Figure 11, A. If e_d is a term of the default calculus and if $e_d \longrightarrow e'_d$, and $e_d \rightrightarrows e_\lambda$, then there exists a term e'_λ of the lambda calculus such that $e_\lambda \longrightarrow^* e'_\lambda$ and $e'_d \rightrightarrows e'_\lambda$. This specific theorem does not apply in our case, because of the thunking we introduce in our translation. As a counter-example, consider the reduction of e_1 within default term $\langle v_0 , e_1 \mid e_{just} :- e_{cons} \rangle$. If e_1 steps to e'_1 in the default calculus, then the whole term steps to $\langle v_0 , e'_1 \mid e_{just} :- e_{cons} \rangle$. However, we translate exceptions to thunks; and our target lambda calculus does not support strong reduction, meaning $\lambda \_ \rightarrow e_{\lambda,1}$ does not step into $\lambda \_ \rightarrow e'_{\lambda,1}$. Diagram A is therefore not applicable in our case.

The theorem that actually holds in our case is shown as diagram B (Figure 11). The two translated terms e_λ and e'_λ eventually reduce to a common form e_{target}. Taking the transitive closure of form B, we obtain that if e_d reduces to a value v, then its translation e_λ reduces to a value v_λ that is the translation of v, a familiar result.

Overview of the proof. We have mechanically formalized the semantics of both the default calculus and target lambda calculus, as well as the translation scheme itself, inside the F⋆ proof assistant. Figure 12 shows the exact theorem we prove, using concrete F⋆ syntax; the theorem as stated establishes both type preservation and variant B, via the **`take_1_steps`** predicate and the existentially quantified **`n1`** and **`n2`**.

Proof effort and engineering. Including the proof of type safety for the source and target language, our F⋆ mechanization amounts to approximately 3,500 lines of code and required 1 person-month. We rely on partial automation via Z3, and the total verification time for the entire development is of the order of a few minutes. The choice of F⋆ was not motivated by any of its advanced features, such as its effect system: the mechanization fits inside the pure fragment of F⋆. Our main motivation was the usage of the SMT solver which can typically perform a fair amount of symbolic reasoning and definition unrolling, thus decreasing the amount of manual effort involved.

To focus the proof effort on the constructs that truly matter (i.e. default expressions), the semantics of lists, folds and options are baked into the target calculus. That is, our target calculus does not support user-defined algebraic data types. We believe this is not a limitation, and instead allows the proof to focus on the key parts of the translation. We use De Bruijn indices for our binder representation, since the unrolling of `process_exceptions` results in variable shadowing. Given those simplifications, we were surprised to find that our proof still required 3,500 lines of F⋆. A lot

```
module D = DefaultCalculus
module L = LambdaCalculus
val translation_correctness (de: D.exp) (dtau: D.ty) : Lemma
  (requires (D.typing D.empty de dtau))
  (ensures (
    let le = translate_exp de in let ltau = translate_ty dtau in
    L.typing L.empty le ltau ∧ begin
      if D.is_value de then L.is_value le else begin
        D.progress de dtau; D.preservation de dtau;
        let de' = Some?.v (D.step de) in
        translation_preserves_empty_typ de dtau; translation_preserves_empty_typ de' dtau;
        let le' : typed_l_exp ltau = translate_exp de' in
        exists (n1 n2:ℕ) (target: typed_l_exp ltau).
          (take_l_steps ltau le n1 == Some target ∧
           take_l_steps ltau le' n2 == Some target) end end))
```

Fig. 12. Translation certification theorem, in F⋆

of the complexity budget was spent on the deep embedding of the `process_exceptions` helper. It is during the mechanization effort that we found out that theorem A does not hold, and that we need to establish B instead. Our mechanized proof thus significantly increases our confidence in the **Catala** compilation toolchain; the proof is evidence that even for a small calculus and a simple translation, a lot of subtleties still remain.

While F⋆ extracts to OCaml, we chose *not* to use the extracted F⋆ code within the **Catala** compiler. First, the proof does not take into account all language features. Second, the actual translation occupies about 100 lines of code in both the production **Catala** compiler and the proof; we are content with comparing both side-by-side. Third, the **Catala** compiler features advanced engineering for locations, error messages, and propagating those to the proof would be difficult.

5 THE CATALA COMPILER

Based on this formalization, we implement **Catala** in a standalone compiler and interpreter. The architecture of the compiler is based on a series of intermediate representations, in the tradition of CompCert [28] or Nanopass [20]. Figure 13 provides a high-level overview of the architecture, with links to relevant sections alongside each intermediate representation.

The compiler is written in OCaml and features approximately 13,000 lines of code. This implementation, available as open-source software on GitHub, makes good use of the rich and state-of-the art library ecosystem for compiler writing, including **`ocamlgraph`** [8] for the e.g. the two topological sorts we saw (Section 4.1), **`bindlib`** [27] for efficient and safe manipulation of variables and terms, and the **`menhir`** parser generator [38]. Thanks to these libraries, we estimate that the development effort was 5 person-months.

5.1 Usability

We devoted a great of attention towards the usability of the compiler. Indeed, while we don't expect lawyers to use **Catala** unaccompanied, we would not want to restrict its usage to λ-savvy functional programmers. To improve the programmer experience, we use the special parser error reporting scheme of **`menhir`** [37], to provide customized and context-aware syntax error messages that depend on the set of tokens acceptable by the grammar at the site of the erroneous token (see Figure 14). The shape of the error messages is heavily inspired by the Rust compiler design [46].

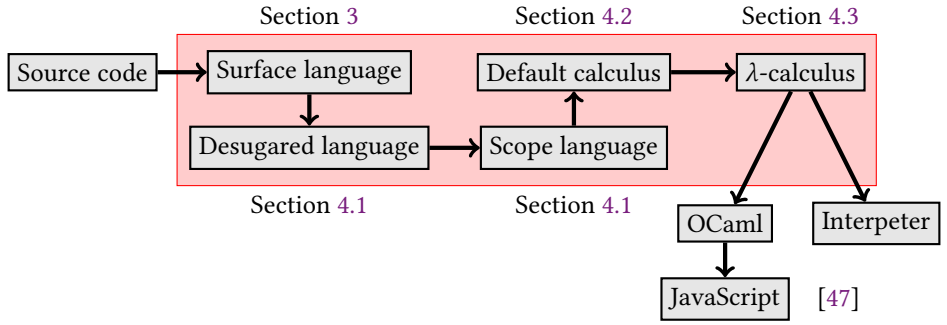

Fig. 13. High-level architecture of the **Catala** compiler (red box)

```
[ERROR] Syntax error at token "years"
[ERROR] Message: expected a unit for this literal, or a valid operator
[ERROR]          to complete the expression
[ERROR] Autosuggestion: did you mean "year", or maybe "or", or maybe "and",
[ERROR]                 or maybe "day", or maybe ".", or maybe ">", [...]
[ERROR]
[ERROR] Error token:
[ERROR]    --> section_121.catala_en
[ERROR]     |
[ERROR] 180 |       if date_of_sale_or_exchange <=@ period.begin +@ 5 years then
[ERROR]     |                                                       ^^^^^
[ERROR]
[ERROR] Last good token:
[ERROR]    --> section_121.catala_en
[ERROR]     |
[ERROR] 180 |       if date_of_sale_or_exchange <=@ period.begin +@ 5 years then
[ERROR]     |                                                     ^
```

Fig. 14. Example of **Catala** syntax error message

Error messages flow through the compiler code via a unique exception containing the structured data of the error message:

```
exception StructuredError of (string * (string option * Pos.t) list)
```

This structure enables on-the-fly rewriting of error messages as they propagate up the call stack, which is useful for e.g. adding a new piece of context linking to a code position of a surrounding AST node. In this spirit, error messages for scope variable cycle detection display the precise location for where the variables in the cycle are used; error messages for default logic conflict errors ⊛ show the location of the multiple definitions that apply at the same time for a unique variable definition.

Finally, we have instrumented the **Catala** interpreter with helpful debugging features. Indeed, when pair programming with a lawyer and a programmer over the formalization of a piece of law, it is helpful to see what the execution of the program would look like on carefully crafted test cases. While test cases can be directly written in **Catala** using a top-level scope that simply defines the arguments of the sub-scope to test, the compilation chain inserts special log hooks at critical code points. When executing a test case, the interpreter then displays a meaningful log (shown in

Appendix B) featuring code positions coupled with the position inside the legal statute for each default logic definition taken.

We believe that this latter feature can easily be extended to provide a comprehensive and legal-oriented explanation of the result of a **Catala** program over a particular input. Such an explanation would help increase trust of the system by its users, e.g. citizens subject to a complex taxation regime; thereby constituting a concrete instance of a much sought-after "explainable AI" [11, 14].

5.2 Performance

Based on a preliminary set of benchmarks, we estimate that a typical program as complex as Section 121 of the US Internal Revenue Code but featuring approximately 1500 lines of Catala code (literate programming included), interprets in approximately 150ms, meaning that the performance of the interpreter remains acceptable even for a production environment.

When the code is compiled to OCaml, execution time drops to 0.5ms. Therefore, we conclude that performance problems are, at this stage of the project, nonexistent.

5.3 Extensible compiler backend

A crucial consideration in designing a DSL is the interoperability story within existing environments. While some DSLs operate in isolation, we envision **Catala** programs being exposed as reusable libraries that can be called from any development platform, following the needs of our adopters. In the context of legal systems, this is a very strong requirement: such environments oftentimes include legacy, mainframe-based systems operating in large private organizations or government agencies [29]. Furthermore, since the algorithms that **Catala** is designed to model are at the very heart of e.g. tax collection systems, proposing a classic interoperability scheme based on APIs or inter-language FFIs might create an undesirable barrier to adoption; a system designed in the 1960s probably has no notion of API or FFI whatsoever!

Instead, we choose for **Catala** an unusually simple and straightforward interoperability scheme: direct source code generation to virtually any programming language. This solution is generally impractical, requiring considerable workarounds to reconcile semantic mismatches between target and source, along with a significant runtime support library. In the case of **Catala**, however, our intentional simplicity makes this "transpiling" scheme possible.

Indeed, the final intermediate representation of the **Catala** compiler is a pure and generic lambda calculus operating over simply-typed values. By re-using standard functional compilation techniques such as closure conversion [32], we claim that it is possible to compile **Catala** to any programming language that has functions, arrays, structures, unions, and support for exceptions. We also believe that a more complex version of the compilation scheme presented in Section 4.3 would remove the need for exceptions (in favor of option types), but leave this as future work.

The runtime required for generated programs only has to include an infinite precision arithmetic library (or can default to fixed-sized arithmetic and floats) and a calendar library to compute the days difference between two dates, taking leap years into account. We demonstrate this with the OCaml backend of the **Catala** compiler, which amounts to 350 lines of compiler code and 150 lines of runtime code (excluding the `zarith` [33] and `calendar` [44] libraries). Merely compiling to OCaml already unlocks multiple target environments, such as the Web, via the `js_of_ocaml` compiler [47]. We thus effortlessly bring **Catala** to the Web.

6 PUTTING CATALA TO WORK

The solid formal and technical foundations of **Catala** would be quite useless if the language was not fit for its target audience: lawyers and legal expert systems programmers. We claim that the design process of **Catala** as well as our comprehensive code co-production process proposal maximizes

the potential for adoption by professionals. To support this claim, we report early user study results and demonstrate an end-to-end use case with the computation of an important French social benefit.

6.1 Interacting with lawyers

Catala's design has been supervized and influenced by lawyers since its inception. Indeed, the project started out of Sarah Lawsky's insight on the logical structure of legal statutes [22–25]. As well as providing the formal base building block of **Catala**, lawyers also reviewed the syntax of **Catala**, choosing the keywords and providing insights counter-intuitive to programmers, such as the `rule`/`definition` distinction of Section 3.2.

We also conducted a careful analysis of the production process of legal expert systems. We found that in France, administrative agencies always use a V-shaped development cycle for their legal expert systems. In practice, lawyers of the legal department take the input set of legal statutes and write a detailed natural language specification, that is supposed to make explicit the different legal interpretations required to turn the legal statute into an algorithm. Then, legal expert systems programmers from the IT department take the natural specification and turn it into code, often never referring back to the original statute text.

Exclusive interviews conducted by the authors with legal expert systems programmers and lawyers inside a high-profile French administration reveal that this theoretical division of labor is artificial. Indeed, the natural language specification often proves insufficient or ambiguous to programmers, which leads to programmers having to spend hours on the phone with the lawyers to clarify and get the algorithm right. Furthermore, the validation of the implementation depends on lawyer-written test cases, whose number and quality suffer from budget restrictions.

This insight suggests that a more agile development process associating lawyers and programmers from the beginning would be more efficient. We claim the **Catala** is the right tool for the job, since it allows lawyers and programmers to perform pair programming on a shared medium that locally combines the legal text as well as the executable code.

We do not expect lawyers to write **Catala** code by themselves. A number of frameworks such as Drools [39] are built on this promise. For our part, we believe that software engineering expertise is needed to produce maintainable, performant, high-quality code. Hence, we envision for lawyers to act as observers and reviewers of the code production process, safeguarding the correctness with respect to the legal specification.

We don't expect adoption difficulties from the programmers' side, since **Catala** is morally a pure functional language with a few oddities that makes it well-suited to legal specifications. To assess our claim of readability by lawyers, we conducted a small user study with $N = 15$ law graduate students enrolled in the Masters program "Droit de la création et numérique" (Intellectual Property and Digital Law) at Université Panthéon-Sorbonne. The methodology of the study is the following: the participants were given a 30 min. presentation of **Catala**'s context and goals, but were not exposed to any program or syntax. Then, participants were briefed during 15 min. about Section 121 and its first paragraph (Section 2.1) and received a copy of the corresponding **Catala** code (Section 3.1 and Section 3.2). After 10 min. of observation, they were asked to answer a series of questions. Then, 15 min. were spent answering participants' questions about the **Catala** code. Finally, the participants filled the questionnaire again.

Table 2 presents the questions asked to the participants, while Figure 15 shows the results for the second and final filling of the questionnaire by the participants. The results for the first round are similar, albeit featuring less positive answers.

These early results, while lacking the rigor of a scientific user study, indicate a relatively good reception of the literate programming paradigm by lawyers. After investigation, we believe that mixed results for question (5) could be explained by a lack of familiarity with the US Internal

#	Exact text of the question
(2)	Can you read the code without getting a headache?
(4)	Can you link the code to the meaning of the law it codifies?
(5)	Can you certify that the code does exactly what the law says and nothing more? If not, are there any mistakes in the code?

Table 2. Selected questions of the user study

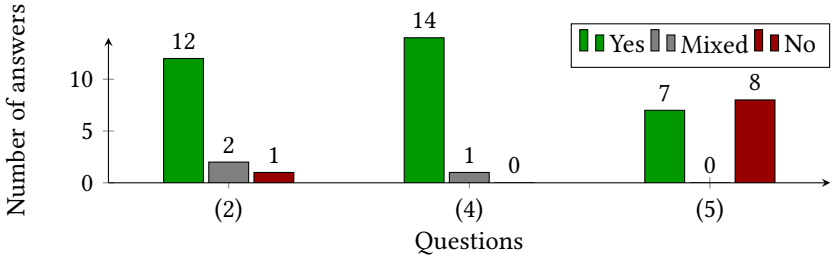

Fig. 15. Results of the second round of questions in the user study

Revenue Code from the French lawyers. Indeed, the wording of the question ("certify") implies that the lawyer would be confident enough to defend their opinion in court. We believe, from deeper interactions with lawyers closer to the project, that familiarity with the formalized law combined with basic **Catala** training could bring the lawyers' confidence to this level.

6.2 A look back to Section 121

We have used Section 121 of the US Internal Revenue Code as an support medium for introducing **Catala** in Section 3. But more interestingly, this piece of law is also our complexity benchmark for legal statutes, as it was deemed (by a lawyer collaborator) to be one of the most difficult sections of the tax code. This reputation comes from its dense wording featuring various layers of exceptions to every parameter of the gross income deduction.

We have so far formalized it up to paragraph (b)(4), which is approximately 15% of the whole section and around 350 lines of code (including the text of the law), but contain its core and most used exceptions. We include the result of this effort in Appendix A. The current formalization was done in four 2-hour sessions of pair programming between the authors and lawyers specialized in the US Internal Revenue Code. Progress is relatively slow because we consider in the process every possible situation or input that could happen, as in a real formalization process. However, this early estimate indicates that formalizing the whole US Internal Revenue Code is a completely reachable target for a small interdisciplinary team given a few years' time.

While finishing the formalization of Section 121 is left as future work, we are confident that the rest of the section can be successfully expressed in **Catala**: the maze of exceptions is localized to (a) and (b), and the rest of the limitations are just a long tail of special cases; with our general design that supports arbitrary trees of exceptions in default logic, this should pose no problem.

6.3 Case study: French Family Benefits

Section 6.1 argues that **Catala** is received positively by lawyers. This is only half of the journey: we need to make sure **Catala** is also successfully adopted by the large private or public organization

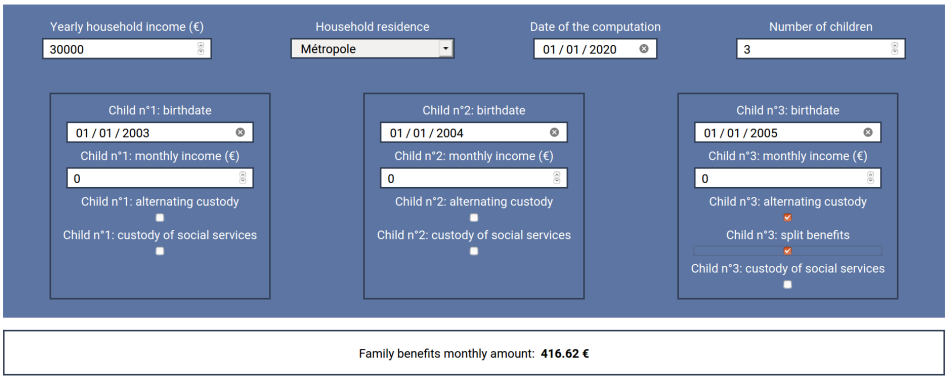

Fig. 16. Screenshot of the Web family benefits simulator powered by **Catala**

where legacy systems are ripe for a principled rewrite. To support our claims concerning the toolchain and interoperability scheme in a real-world setting, we formalized the entire French family benefits computation in **Catala** and exposed the compiled program as an OCaml library and JavaScript Web simulator. The full code of this example can be found in the supplementary material of this article, although it is written in French.

A crucial part of the French welfare state, family benefits are distributed to households on the basis of the number of their dependent children. Created in the early 1930's, this benefit was designed to boost French natality by offsetting the additional costs incurred by child custody to families. Family benefits are a good example of a turbulent historical construction, as the conditions to access the benefits have been regularly amended over the quasi-century of existence of the program. For instance, while family benefits were distributed to any family without an income cap, a 2015 reform lowered the amount of the benefit for wealthy households [4, 6].

The computation can be summarized with the following steps. First, determine how many dependent children are relevant for the family benefits (depending on their age and personal income). Second, compute the base amount, which depends on the household income, the location (there are special rules for overseas territories) and a coefficient updated each year by the government to track inflation. Third, modulate this amount in the case of alternating custody or social services custody. Fourth, apply special rules for when a child is exactly at the age limit for eligibility, or when the household income is right above a threshold. All of these rules are specified by 27 articles of the French Social Security Code, as well as various executive orders.

The **Catala** formalization of this computation amounts to approximately 1,500 lines of code, including the text of the law. The code is split between 6 different **`scope`**s featuring 63 **`context`** variables and 83 **`definition`**s and **`rule`**s. We believe these numbers to be fairly representative. Distributed as an OCaml library, our code for the computation of the French family benefits is also used to power an online simulator (see Figure 16).

After writing the code as well as some test cases, we compared the results of our program with the official state-sponsored simulator mes-droits-sociaux.gouv.fr, and found no issue. However, the case where a child is in the custody of social services was absent from the official simulator, meaning we could not compare results for this case. Fortunately, the source code of the simulator is available as part of the OpenFisca software project [43]. The OpenFisca source file corresponding to the family benefits, amounts to 365 lines of Python. After close inspection of the OpenFisca code, a discrepancy was located with the **Catala** implementation. Indeed, according to article L755-12 of the Social Security Code, the income cap for the family benefits does not apply in overseas

territories with single-child families. This subtlety was not taken into account by OpenFisca, and was fixed after its disclosure by the authors.

Formalizing a piece of law is thus no different from formalizing a piece of software with a proof assistant. In both cases, bugs are found in existing software, and with the aid of a formal mechanization, can be reported and fixed.

7 CONCLUSION & RELATED WORK

Catala follows a long tradition of scholarly works that aim to extract the logical essence of legal statutes, starting as early as 1914 [10]. To provide some context, we compare our work with two seminal articles in the field.

In his visionary 1956 article, Allen [2] notes that symbolic logic can be used to remove ambiguity in the law, and proposes its use for a wide range of applications: legal drafting, interpretation, simplification and comparison. Using custom notations that map transparently to first-order logic, Allen does not provide an operational tool to translate law into formalism but rather points out the challenges such as law ambiguity and rightfully sets the limits of his approach, stating for instance that in generality, "filling of gaps in legislation by courts cannot and should not be entirely eliminated". Interestingly, he also manually computes a table of truth to prove that two sections of the US Internal Revenue Code are equivalent.

The vision laid out by Allen is refined in 1986 by Sergot *et al.* [42]. This article narrows the range of its formalism to statutory law (as opposed to case law), and focuses on the British Nationality Act, a statute used to determine whether a person can qualify for the British nationality based on various criteria. Co-authored by Robert Kowalski, this works features the use of Prolog [7] as the target programming language, showing the usefulness of declarative logic programming for the formalization task. However, the work acknowledges a major limitation concerning the expression of negation in the legal text, and points out that "the type of default reasoning that the act prescribes for dealing with abandoned infants is nonmonotonic", confirming the later insights of Lawsky [24]. A major difference with **Catala** is the absence of literate programming; instead, Sergot *et al.* derived a synthetic and/or diagram as the specification for their Prolog program.

However, the line of work around logic programming never took hold in the industry and the large organizations managing legal expert systems. The reasons, various and diagnosed by Leigh [26], mix the inherent difficulty of translating law to code, with the social gap between the legal and computer world. As a reaction, several and so far unsuccessful attempts were made to automate the translation using natural language processing techniques [17, 36]. Others claim that the solution is to lower the barriers to the programming world using low-code/no-code tools, so that lawyers can effectively code their own legal expert systems [35].

The main recent research direction around the formalization of law is spearheaded by optimistic proponents of computational law [13], promising a future based on Web-based, automated legal reasoning by autonomous agents negotiating smart contracts on a blockchain-powered network [16, 18, 41, 49].

By contrast, we focus on the challenges related to maintaining existing legal expert systems in large public or private organizations, providing essential services to millions of citizens and customers. **Catala** aims to provide an industrial-grade tool that enables close collaboration of legal and IT professionals towards the construction of correct, comprehensive and performant implementations of algorithmic statutory law.

The wide range of applications imagined by Layman in 1956 has yet to be accomplished in practice. With its clear and simple semantics, we hope for **Catala** formalizations of statutes to provide ideal starting point for future formal analyses of the law, enabling legal drafting, interpretation, simplification and comparison using the full arsenal of modern formal methods.

REFERENCES

[1] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 515–529.

[2] Layman E Allen. 1956. Symbolic logic: A razor-edged tool for drafting and interpreting legal documents. *Yale LJ* 66 (1956), 833.

[3] Anne Broache. 2008. IRS trudges on with aging computers. https://www.cnet.com/news/irs-trudges-on-with-aging-computers/

[4] Marie-Andrée Blanc. 2016. La modulation des allocations familiales : une erreur historique. *Travail, genre et sociétés* 35, 1 (2016), 157–161. https://doi.org/10.3917/tgs.035.0157

[5] Gerhard Brewka and Thomas Eiter. 2000. *Prioritizing Default Logic*. Springer Netherlands, Dordrecht, 27–45.

[6] Marie-Françoise Clergeau. 2016. Plaidoyer pour la modulation. *Travail, genre et sociétés* 35, 1 (2016), 173–177. https://doi.org/10.3917/tgs.035.0173

[7] Alain Colmerauer and Philippe Roussel. 1996. *The Birth of Prolog*. Association for Computing Machinery, New York, NY, USA, 331–367.

[8] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. 2007. Designing a Generic Graph Library Using ML Functors. *Trends in functional programming* 8 (2007), 124–140.

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[10] John Dewey. 1914. Logical method and law. *Cornell LQ* 10 (1914), 17.

[11] Derek Doran, Sarah Schulz, and Tarek R Besold. 2017. What does explainable AI really mean? A new conceptualization of perspectives. *arXiv preprint arXiv:1710.00794* (2017).

[12] Frank R. Konkel. 2018. The IRS system processing your taxes is almost 60 years old. https://www.nextgov.com/it-modernization/2018/03/irs-system-processing-your-taxes-almost-60-years-old/146770/

[13] Michael Genesereth. 2015. Computational Law. *The Cop in the Backseat* (2015).

[14] Randy Goebel, Ajay Chander, Katharina Holzinger, Freddy Lecue, Zeynep Akata, Simone Stumpf, Peter Kieseberg, and Andreas Holzinger. 2018. Explainable ai: the new 42?. In *International cross-domain conference for machine learning and knowledge extraction*. Springer, 295–303.

[15] Government Accountability Office (GAO). 2021. COVID-19: Urgent Actions Needed to Better Ensure an Effective Federal Response – Report to Congressional Committees. https://www.gao.gov/reports/GAO-21-191/#appendix24 Appendix 24, first table.

[16] Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. 2018. Spesc: A specification language for smart contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 132–137.

[17] Nils Holzenberger, Andrew Blair-Stanek, and Benjamin Van Durme. 2020. A Dataset for Statutory Reasoning in Tax Law Entailment and Question Answering. *arXiv preprint arXiv:2005.05257* (2020).

[18] Tom Hvitved. 2011. *Contract formalisation and modular implementation of domain-specific languages*. Ph.D. Dissertation. Citeseer.

[19] Internal Revenue Service. [n.d.]. Exclusion of gain from sale of principal residence. https://www.law.cornell.edu/uscode/text/26/121

[20] Andrew W Keep and R Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 343–350.

[21] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (01 1984), 97–111.

[22] Sarah Lawsky. 2018. Formal Methods and the Law. https://popl18.sigplan.org/details/POPL-2018-papers/3/Formal-Methods-and-the-Law

[23] Sarah B. Lawsky. 2017. Formalizing the Code. *Tax Law Review* 70, 377 (2017).

[24] Sarah B. Lawsky. 2018. A Logic for Statutes. *Florida Tax Review* (2018).

[25] Sarah B Lawsky. 2020. Form as Formalization. *Ohio State Technology Law Journal* (2020).

[26] Philip Leith. 2016. The rise and fall of the legal expert system. *International Review of Law, Computers & Technology* 30, 3 (2016), 94–106.

[27] Rodolphe Lepigre and Christophe Raffalli. 2018. Abstract representation of binders in ocaml using the bindlib library. *arXiv preprint arXiv:1807.01872* (2018).

[28] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. *SIGPLAN Not.* 41, 1 (Jan. 2006), 42–54.

[29] Kif Leswing. 2020. New Jersey needs volunteers who know COBOL, a 60-year-old programming language. https://www.cnbc.com/2020/04/06/new-jersey-seeks-cobol-programmers-to-fix-unemployment-system.html

[30] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin HriŢcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 30–59.
[31] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021. A Modern Compiler for the French Tax Code. *Compiler Construction* (2021).
[32] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 271–283.
[33] Antoine Miné, Xavier Leroy, Pascal Cuoq, and Christophe Troestler. 2011. *The Zarith OCaml library*. https://github.com/ocaml/Zarith
[34] Jacques Monin. 2018. Louvois, le logiciel qui a mis l'armée à terre. https://www.franceinter.fr/emissions/secrets-d-info/secrets-d-info-27-janvier-2018
[35] Jason Morris. 2020. Spreadsheets for Legal Reasoning: The Continued Promise of Declarative Logic Programming in Law. *Available at SSRN 3577239* (2020).
[36] Marcos A Pertierra, Sarah Lawsky, Erik Hemberg, and Una-May O'Reilly. 2017. Towards Formalizing Statute Law as Default Logic through Automatic Semantic Parsing.. In *ASAIL@ ICAIL*.
[37] François Pottier. 2016. Reachability and error diagnosis in LR (1) parsers. In *Proceedings of the 25th International Conference on Compiler Construction*. 88–98.
[38] François Pottier and Yann Régis-Gianat. [n.d.]. *The Menhir Parser Generator*. http://cambium.inria.fr/~fpottier/menhir/
[39] Mark Proctor. 2012. Drools: A Rule Engine for Complex Event Processing. In *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance* (Budapest, Hungary) *(AGTIVE'11)*. Springer-Verlag, Berlin, Heidelberg, 2–2. https://doi.org/10.1007/978-3-642-34176-2_2
[40] R. Reiter. 1980. A logic for default reasoning. *Artificial Intelligence* 13, 1 (1980), 81 – 132. Special Issue on Non-Monotonic Logic.
[41] Vincenzo Scoca, Rafael Brundo Uriarte, and Rocco De Nicola. 2017. Smart contract negotiation in cloud computing. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 592–599.
[42] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. 1986. The British Nationality Act As a Logic Program. *Commun. ACM* 29, 5 (May 1986), 370–386.
[43] Sébastien Shulz. 2019. Un logiciel libre pour lutter contre l'opacité du système sociofiscal. *Revue francaise de science politique* 69, 5 (2019), 845–868.
[44] Julien Signoles. 2011. *The Calendar OCaml library*. https://github.com/ocaml-community/calendar
[45] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/
[46] Jonathan Turner. 2016. Shape of errors to come. https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html.
[47] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972.
[48] Wikipedia contributors. 2021. Code of Ur-Nammu — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Code_of_Ur-Nammu&oldid=998720276 [Online; accessed 22-Feb-2021].
[49] Jakub Zakrzewski. 2018. Towards verification of Ethereum smart contracts: a formalization of core of Solidity. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 229–247.

ACKNOWLEDGMENTS

We want to thank first the lawyers at the heart of the **Catala** project: Sarah Lawsky and Liane Huttner. Theirs insights and continued collaboration were invaluable for the success of this endeavor. We also thank Pierre-Évariste Dagand for its useful advice of encoding the default logic partial order into a syntactic tree; this trick helped simplify a lot the formalization, without loss of generality with respect to legislative texts.

This work is partially supported by the European Research Council under the CIRCUS (683032) Consolidator Grant Agreement.

A FULL CODE OF THE SECTION 121 EXAMPLE

This document presents some notable differences with the code presented in Section 3. Indeed, we adapted some of the excerpts to better serve the introductory nature of the tutorial, as Section 121 of the US Tax Code was not written to be a tutorial for a domain specific language. For instance, the scope `Section121Return` in Section 3.4 is named `Section121TwoPersons` here, and covers slightly more features.

This document is to be read linearly, as would be a literate programming document. The implementation begins with a lot of metadata (Section 3.1), then features the **Catala** code intertwined with the legal prose of Section 2. Note that this LaTeX document was generated automatically from the **Catala** source code of the Section 121, using the compiler of Section 5. We believe this PDF-friendly output to be more palatable to lawyers, that do not yet know the joys of version control systems.

THE US TAX CODE

Metadata

```
declaration structure Person:
  data id content integer
```

SECTION 121

Metadata

```
declaration structure Period:
  data begin content date
  data end content date

declaration scope PeriodMerge:
  context periods1 content collection Period
  context periods2 content collection Period
  context output content collection Period

scope PeriodMerge:
  # Placeholders, overwritten by caller
  definition periods1 equals []
  definition periods2 equals []

  # TODO: This requires implementing the merging of two collections of date
  # periods into a single non-overlapping collection of date periods such
  # that the output covers both input date ranges. This algorithm involves
  # sorting the collections, something we can't express in Catala. This is
  # the classical DSL fallacy, and its classical solution is simply to assume
  # the scope PeriodMerge as external and implement it in the target programming
  # language.
  definition output equals []

declaration scope Section121SinglePerson:
  context requirements_met condition
```

Metadata

```
    context requirements_ownership_met condition
    context requirements_usage_met condition
    context date_of_sale_or_exchange content date
    context property_ownership content collection Period
    # Invariant: the periods in the collection are disjoint
    context property_usage_as_principal_residence
      content collection Period
    # Invariant: the periods in the collection are disjoint
    context aggregate_periods_from_last_five_years content duration
      depends on collection Period
    context gain_cap content money
    context gain_from_sale_or_exchange_of_property content money
    context income_excluded_from_gross_income_uncapped content money
    context income_excluded_from_gross_income content money

declaration structure PreviousSaleWhereSection121aApplied:
  data date_of_sale_or_exchange content date

declaration enumeration OtherSection121aSale:
   NoOtherSaleWhereSection121aApplied
   MostRecentSaleWhereSection121aApplied content
    PreviousSaleWhereSection121aApplied

declaration structure PersonalData:
  data property_ownership content collection Period
  data property_usage_as_principal_residence
    content collection Period
  data other_section_121a_sale content OtherSection121aSale

declaration structure JointReturn:
  data person1 content PersonalData
  data person2 content PersonalData

declaration structure DeadSpouseInfo:
  data return content PersonalData
  data date_of_spouse_death content date
  data death_spouse_info_at_time_of_death content PersonalData

declaration enumeration ReturnType:
   SingleReturn content PersonalData
   JointReturn content JointReturn
   SingleReturnSurvivingSpouse content DeadSpouseInfo

declaration scope Section121TwoPersons:
  context person1 content PersonalData
```

Metadata

```
    context section121Person1 scope Section121SinglePerson
    context person2 content PersonalData
    context section121Person2 scope Section121SinglePerson
    context section121a_requirements_met condition
    context section_121_b_3_applies content boolean
      depends on OtherSection121aSale
    context section_121_b_2_A_condition condition
    context gain_cap_person_1 content money
    context gain_cap_person_2 content money
    context gain_cap content money
    context return_type content ReturnType
    context return_date content date
    context date_of_sale_or_exchange content date
    context gain_from_sale_or_exchange_of_property content money
    context income_excluded_from_gross_income_uncapped content money
    context income_excluded_from_gross_income content money
    context period_merge scope PeriodMerge

  # Defining sub-scopes arguments
  scope Section121TwoPersons:
    definition section121Person2.date_of_sale_or_exchange equals
      date_of_sale_or_exchange
    definition section121Person1.date_of_sale_or_exchange equals
      date_of_sale_or_exchange

    definition person1 equals match return_type with pattern
       SingleReturn of data_person1 : data_person1
       JointReturn of data_couple : data_couple.person1
       SingleReturnSurvivingSpouse of data_single: data_single.return

    definition person2 equals match return_type with pattern
       SingleReturn of data_person2 : data_person2
       JointReturn of data_couple : data_couple.person2
       SingleReturnSurvivingSpouse of data_single: data_single.return

    definition section121Person1.property_ownership equals
       person1.property_ownership

    definition section121Person1.property_usage_as_principal_residence equals
      person1.property_usage_as_principal_residence

    definition section121Person2.property_ownership equals
      person2.property_ownership

    definition section121Person2.property_usage_as_principal_residence equals
```

Metadata

```
    person1.property_usage_as_principal_residence

  definition section121Person1.gain_from_sale_or_exchange_of_property equals
    gain_from_sale_or_exchange_of_property
  definition section121Person2.gain_from_sale_or_exchange_of_property equals
    gain_from_sale_or_exchange_of_property

  definition gain_cap_person_1 equals section121Person1.gain_cap
  definition gain_cap_person_2 equals section121Person2.gain_cap

declaration scope Section121TwoPasses:
  context first_pass scope Section121TwoPersons
  context second_pass scope Section121TwoPersons
  context return_type content ReturnType
  context return_date content date
  context date_of_sale_or_exchange content date
  context gain_from_sale_or_exchange_of_property content money
  context period_merge scope PeriodMerge
  context income_excluded_from_gross_income content money

# Defining sub-scopes arguments
scope Section121TwoPasses:
  definition first_pass.return_type equals return_type
  definition second_pass.return_type equals return_type

  definition first_pass.return_date equals return_date
  definition second_pass.return_date equals return_date

  definition first_pass.gain_from_sale_or_exchange_of_property equals
    gain_from_sale_or_exchange_of_property
  definition second_pass.gain_from_sale_or_exchange_of_property equals
    gain_from_sale_or_exchange_of_property

  definition first_pass.date_of_sale_or_exchange equals
    date_of_sale_or_exchange
  definition second_pass.date_of_sale_or_exchange equals
    date_of_sale_or_exchange

  definition income_excluded_from_gross_income equals
    second_pass.income_excluded_from_gross_income
```

(a) Exclusion. Gross income shall not include gain from the sale or exchange of property if, during the 5-year period ending on the date of the sale or exchange, such property has been owned and used by the taxpayer as the taxpayer's principal residence for periods aggregating 2 years or more.

```
scope Section121SinglePerson:
  # Here we aggregate over all the periods of the collection. For
  # each period, three cases:
  # - either the period began less that 5 years before the
  #   date_of_sale_or_exchange in which case we count if full
  # - either the period ended more that 5 years before the
  #   date_of_sale_or_exchange in which case we don't count it
  # - either the 5 years mark is inside the period and we only
  #   cound the half after 5 years
  definition aggregate_periods_from_last_five_years of periods equals
    sum duration for period in periods of (
      if date_of_sale_or_exchange ⩽@ period.begin +@ 5 year then
        period.end -@ period.begin
      else (if date_of_sale_or_exchange ⩾@ period.end +@ 5 year then
        0 day
      else ((period.end +@ 5 year) -@ date_of_sale_or_exchange))
    )

  # Regulation 1.121-1(c)(1): 2 years = 730 days
  # Regulation 1.121-1(c)(1): the periods of ownership and usage
  # don't have to overlap
  rule requirements_ownership_met under condition
    aggregate_periods_from_last_five_years of property_ownership ⩾^ 730 day
  consequence fulfilled

  rule requirements_usage_met under condition
    aggregate_periods_from_last_five_years of
       property_usage_as_principal_residence ⩾^ 730 day
  consequence fulfilled

  rule requirements_met under condition
    requirements_ownership_met and requirements_usage_met
  consequence fulfilled

  definition income_excluded_from_gross_income_uncapped equals
    if requirements_met then gain_from_sale_or_exchange_of_property
    else $0

scope Section121TwoPersons:
  definition section121a_requirements_met equals section121Person1.requirements_met

  definition income_excluded_from_gross_income_uncapped equals
    section121Person1.income_excluded_from_gross_income_uncapped
```

(B) LIMITATIONS

(1) In general. The amount of gain excluded from gross income under subsection (a) with respect to any sale or exchange shall not exceed $250,000.

```
scope Section121SinglePerson:
  definition gain_cap equals $250,000

  definition income_excluded_from_gross_income equals
    if income_excluded_from_gross_income_uncapped ⩾$ gain_cap then
      gain_cap
    else
      income_excluded_from_gross_income_uncapped

scope Section121TwoPersons:
    definition gain_cap equals section121Person1.gain_cap

    definition income_excluded_from_gross_income equals
    if income_excluded_from_gross_income_uncapped ⩾$ gain_cap then
      gain_cap
    else
      income_excluded_from_gross_income_uncapped
```

(2) Special rules for joint returns. In the case of a husband and wife who make a joint return for the taxable year of the sale or exchange of the property—

(A) $500,000 Limitation for certain joint returns. Paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if—

(i) either spouse meets the ownership requirements of subsection (a) with respect to such property;

(ii) both spouses meet the use requirements of subsection (a) with respect to such property; and

(iii) neither spouse is ineligible for the benefits of subsection (a) with respect to such property by reason of paragraph (3).

```
scope Section121TwoPersons:
  rule section_121_b_2_A_condition under condition
  (return_type with pattern JointReturn of data_couple)
  and
  # i)
  (section121Person1.requirements_ownership_met or
     section121Person2.requirements_ownership_met)
  and
  # ii)
  (section121Person1.requirements_usage_met and
     section121Person2.requirements_usage_met)
  # iii)
  and
  (not (
      section_121_b_3_applies of data_couple.person1.other_section_121a_sale))
  and
  (not (
      section_121_b_3_applies of data_couple.person2.other_section_121a_sale))
  consequence fulfilled
```

```
  exception
  rule section121a_requirements_met under condition
    section_121_b_2_A_condition
  consequence fulfilled

  exception
  definition gain_cap under condition
    section_121_b_2_A_condition
  consequence equals $500,000
```

(B) Other joint returns. If such spouses do not meet the requirements of subparagraph (A), the limitation under paragraph (1) shall be the sum of the limitations under paragraph (1) to which each spouse would be entitled if such spouses had not been married. For purposes of the preceding sentence, each spouse shall be treated as owning the property during the period that either spouse owned the property.

```
scope Section121TwoPasses under condition
  (return_type with pattern JointReturn) and
  not (first_pass.section_121_b_2_A_condition):

  definition second_pass.gain_cap equals
    first_pass.gain_cap_person_1 +$
    first_pass.gain_cap_person_2

  definition period_merge.periods1 equals match return_type with pattern
     JointReturn of joint_return: joint_return.person1.property_ownership
     SingleReturnSurvivingSpouse of dead_spouse_info : [] # does not happen
     SingleReturn of return : [] # does not happen
  definition period_merge.periods2 equals match return_type with pattern
     JointReturn of joint_return: joint_return.person2.property_ownership
     SingleReturnSurvivingSpouse of dead_spouse_info : [] # does not happen
     SingleReturn of return : [] # does not happen

  definition second_pass.person1 equals PersonalData {
     property_ownership: period_merge.output
     property_usage_as_principal_residence:
      first_pass.person1.property_usage_as_principal_residence
     other_section_121a_sale: first_pass.person1.other_section_121a_sale
  }
  definition second_pass.person2 equals PersonalData {
     property_ownership: period_merge.output
     property_usage_as_principal_residence:
      first_pass.person2.property_usage_as_principal_residence
     other_section_121a_sale: first_pass.person2.other_section_121a_sale
  }
```

(3) Application to only 1 sale or exchange every 2 years. Subsection (a) shall not apply to any sale or exchange by the taxpayer if, during the 2-year period ending on the date of such sale or exchange, there was any other sale or exchange by the taxpayer to which subsection (a) applied.

```
scope Section121TwoPersons:
  definition section_121_b_3_applies of other_section_121a_sale equals
    (other_section_121a_sale with pattern
       MostRecentSaleWhereSection121aApplied of other_sale) and
    date_of_sale_or_exchange -@ other_sale.date_of_sale_or_exchange ⩽^ 2 year
```

(4) Special rule for certain sales by surviving spouses.

```
# Sarah: the year when your spouse dies, do you file a joint return or
# separate returns?
```

In the case of a sale or exchange of property by an unmarried individual whose spouse is deceased on the date of such sale, paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if such sale occurs not later than 2 years after the date of death of such spouse and the requirements of paragraph (2)(A) were met immediately before such date of death.

```
scope Section121TwoPasses under condition
  return_type with pattern SingleReturnSurvivingSpouse of single_data and
  single_data.date_of_spouse_death <@ date_of_sale_or_exchange and
  date_of_sale_or_exchange ⩽@ single_data.date_of_spouse_death +@ 2 year
  # So here we have to reexecute the scope Section121 using
  # single_data.date_of_spouse_death instead of date_of_sale_or_exchange
  :

  definition second_pass.date_of_sale_or_exchange equals
    match return_type with pattern
     SingleReturnSurvivingSpouse of single_data: single_data.date_of_spouse_death
     SingleReturn of return: date_of_sale_or_exchange # does not happen
     JointReturn of return: date_of_sale_or_exchange # does not happen
  definition second_pass.gain_cap equals $500,000
```

B EXCERPT OF THE INTERPRETER LOG OF A SECTION121 RUN

```
[LOG]     →  Section121SinglePerson.aggregate_periods_from_last_five_years
[LOG]       :=  Section121SinglePerson.aggregate_periods_from_last_five_years.input:
[LOG]            [Period {"begin": 2017-01-01, "end": 2021-01-01}]
[LOG]       > Definition applied:
[LOG]          --> tests/../section_121.catala_en
[LOG]           |
[LOG]       178 |   definition aggregate_periods_from_last_five_years of periods equals
[LOG]           |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[LOG]           + Section 121
[LOG]           +-+ (a) Exclusion
[LOG]       :=  Section121SinglePerson.aggregate_periods_from_last_five_years.output: 1461 days
[LOG]     ←  Section121SinglePerson.aggregate_periods_from_last_five_years
[LOG]     > Definition applied:
[LOG]        --> tests/../section_121.catala_en
[LOG]         |
[LOG]     191 |     aggregate_periods_from_last_five_years of property_ownage >=^ 730 day
[LOG]         |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[LOG]         + Section 121
[LOG]         +-+ (a) Exclusion
[LOG]     :=  Section121SinglePerson.requirements_ownership_met: true
```

```
[LOG]     > Definition applied:
[LOG]        --> tests/../section_121.catala_en
[LOG]         |
[LOG]     200 |     requirements_ownership_met and requirements_usage_met
[LOG]         |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[LOG]         + Section 121
[LOG]         +-+ (a) Exclusion
[LOG]     :=  Section121SinglePerson.requirements_met: true
[LOG]     > Definition applied:
[LOG]        --> tests/../section_121.catala_en
[LOG]         |
[LOG]     203 |   definition income_excluded_from_gross_income_uncapped equals
[LOG]         |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[LOG]         + Section 121
[LOG]         +-+ (a) Exclusion
[LOG]     :=  Section121SinglePerson.income_excluded_from_gross_income_uncapped: $350000.00
[LOG]     > Definition applied:
[LOG]        --> tests/../section_121.catala_en
[LOG]         |
[LOG]     228 |   definition income_excluded_from_gross_income equals
[LOG]         |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[LOG]         + Section 121
[LOG]         +-+ (b) Limitations
[LOG]           +-+ (1) In general
[LOG]     :=  Section121SinglePerson.income_excluded_from_gross_income: $250000.00
[LOG]    ←  Section121TwoPersons.section121Person2.Section121SinglePerson
```