L4 Manual

Release 0.9

Meng Weng Wong

Jan 12, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This manual is:

- A conceptual reference explaining why the L4 language is important.
- A comprehensive language syntax reference for the L4 language.
- A collection of explanations of advanced topics, such as how the Natural Language Generation works in L4. (Work in progress)
- A collection of FAQs and How-Tos. (Work in progress)

For installation instructions, a quick overview of L4, and a brief explanation as to what the L4 language is good for, go to the Quickstart manual. It is available in the LegalSS Google Sheets itself, or as a Read the Docs document.

This manual was originally written in a tab in the LegalSS spreadsheet.

In the LegalSS spreadsheet, for both the Quickstart guide and the L4 Manual, you should see some "+" icons on the left of the spreadsheet. Clicking those icons will expand and collapse particular sections of the spreadsheet for more or less detail.

CHAPTER

TWO

CONCEPTUAL REFERENCE

The following discusses law as a problem in computer science.

The law encompasses a rich and majestic history of monarchs and parliaments, human rights and natural justice, torts and equity, judgements and appeals, death sentences and stays of execution. Movies and TV shows from "A Few Good Men" to "She-Hulk" glamourize the profession.

By comparison, computational law, a new field whose history is measured in decades, not centuries, is bloodless and dry; there are no movies, no TV shows. But the growing overlap between law and software portends a quiet digital transformation of the way people conduct business and organize their lives.

The overarching theme of computational law, consistent with the aphorism "software is eating the world", is the analysis of legal problems from the perspective of computer science.

2.1 The L4 approach to the problem of law

The L4 approach considers contracts and legislation to be *attempts at specification*.

The behaviour of actors is regulated by rules which describe state transition systems. The transitions between states are themselves governed by decision rules.

Both kinds of rules are expressible ultimately in first-order logic and are recursively subject to meta-rules like priority ordering.

The evaluation at run-time of decision elements to binary truth values is dependent on human input.

Legacy legal drafting is frequently marred by undesirable ambiguities which are attributable to the lack of a formal language.

2.1.1 Advantages of modelling using a formal system

Modelling the above ideas in a formal system makes it possible to apply software engineering practices to the legal sphere. New technologies and tools can help both lawyers and non-lawyers perform legally-oriented work in different and better ways.

For example: ambiguities in complicated sentences, commonly laid at the door of the "Oxford comma", and sometimes the cause of lawsuits, can be considered syntax errors.

Misunderstandings between parties can be considered failures of requirements elicitation, and can be discovered prior to contract signing using methodologies like behaviour-driven development and unit testing.

Under-specification can be mechanically detected by methods borrowed from functional programming such as type checking and totality analysis.

Finally, more mundanely, version control and structured data embedded in documents provide basic levels of machine-readability lacking today.

2.2 Limits to Computational Law

Not every problem in the legal world will be solved by software. Most commonly, terms which "ground out" in some English phrase not further defined will need human interpretation, typically via resort to case law.

Computational law has little to contribute in situations where contracting parties are less concerned with the details of their agreement than on agreeing that they have agreed, or where at least one party is aware that their counterparties may understand the agreement quite differently or not at all, but is prepared to resolve those differences at a later time, after the contract is signed.

2.3 Expanding on technical terms

Let us return to the technical terms mentioned in *The L4 approach to the problem of law*.

2.3.1 On Specification Languages

Specification languages are distinct from programming languages. They are considered "higher-level", and are used primarily for analyzing the properties of systems even before the systems are implemented or run.

Examples of specification languages include Alloy and TLA+.

Both specification and programming languages are *formal languages*, which we expand upon below.

2.3.2 On Formalisation

Computer scientists are familiar with state machines, which represent discrete change over time.

One of the earliest papers in computational law identified Petri Nets as a useful formalism for representing state.

The notion of contract as state machine can be found at the heart of the Business Process Modeling Notation (BPMN) standard for business processes.

2.3.3 On Logic

Computer scientists are also familiar with propositional predicate, and Boolean logics.

They deal with True and False values, combined using AND / OR / NOT connectors into potentially large trees of conditions. These logics lie at the heart of "rule systems", which calculate Boolean outputs based on inputs that are either natively Boolean or are reducible to Booleans, like x == 42. First-order predicate logic helps reduce a complex universe of many types, to Boolean values.

The values of terms are often debatable: reducing a messy, real-world, real-valued term to a Boolean is not always straightforward. "No Vehicles In The Park" provides the classic example of such difficulties. What is a vehicle? Where does the park end? If the park is flooded, does the rule still apply?

Different logics may provide different methods of performing such reduction. Logic programming offers ""negation as failure"" but that is not the only choice. And vacuous truths may lead to explosion.

This reduction problem lies at the heart of many legal conflicts: parties may disagree about the values of terms, and further disagree about the choice of logic.

In the real world it is often necessary to take unknown and undefined values into account: hence the need for a ternary logic.

2.3.4 On Rewrites

The primary specifications which attempt to establish a rule system are themselves subject to rewriting according to further meta-rule systems.

Some of these rewrites may be within the primary specification itself. In this section, any reference to dollars shall mean United States Dollars.

Other rewrites may occur "beyond the awareness" of the primary specification: "any clause of any contract which attempts to establish a non-compete shall be unenforceable."

These transformations are familiar to computer science. Given the text of a program, a compiler may perform transformations and optimizations and dead-code elimination through tree-shaking.

An operating system may choose to block certain system calls depending on access control privileges, or attach a debugger to an executing instance.

A microprocessor may perform speculative execution and out-of-order instruction pipelining.

When multiple rules collide, they can be resolved using a ordering mechanism: firewall rules, for example, include priorities.

2.3.5 On Evaluation

The "evaluation" of a specification depends on its run-time environment and often on *human input*.

Computer science is familiar with the notion of "static analysis", which attempts to show that a program, or specification, satisfies or violates certain properties.

In other words, it should be possible to identify, at the time of drafting, if a law or contract contains undesirable loopholes by which parties may escape intended consequences.

Static analysis methods include SAT solving, which can be said to attempt to anticipate every eventuality.

However, such methods cannot anticipate meta-rules operating outside the bounds of the system. A war of foreign occupation, for instance, may invalidate existing laws and contracts in unpredictable ways.

In any case, it is frequently impossible to determine in advance if a particular event will be considered to have met a certain standard.

Some degree of vagueness is inevitable, and, frequently, desirable: when a thing cannot be defined in any more detail, or it depends on which way the wind is blowing at the time, we need a human to step in and decide.

Did a party apply "reasonable efforts" to a particular action? It depends … on a decision tree which, sooner or later, bottoms out and needs to call an external decider for input.

2.3.6 On Natural Language

Because laws and contracts have, to date, been written in natural languages like English, drafters sometimes introduced ambiguities into their text.

Sometimes, it is up to a judge to make sense of manifestly ungrammatical sentences. Interpretive doctrines like purposive intent help them do their job.

The logical conjunction "A, B, C and D or E" can be interpreted at least four different ways.

Formal languages, like L4, force the drafter to clarify their meaning, by triggering compiler warnings upon encountering statements that are not well-formed.

2.4 Machines consuming Law

One of the motivations for computational law is to apply the tools and techniques of computer science to represent legal constructs and automate legal reasoning in more formal ways, that are consumable by machine and ultimately useful to lay end-users.

The Contract Lifecycle Management industry was sized at $1.7B in 2021 and is expected to double to $3B by 2026. Similarly, there is increasing interest in "Rules as Code" by governments around the world as it promises to bring "digital transformation" to service delivery.

In 2021, Gartner identified machine-readable legislation as a key emerging technology to watch. If computational law is successful in putting laws and contracts on a firmly digital basis, it can serve as a basis for innovation in the CLM industry, the legal industry, and e-government.

2.5 Comparing concepts in Law and Computer Science

For the computer scientist new to law, it helps to recognize familiar concepts in unfamiliar guises.

For example, programmers are familiar with "default logic" in the form of if / then / else-if / then / else statements: each if condition is evaluated in turn. If all the branches are false, the default else is chosen.

In legal writing, the order is reversed: *the default rule is stated first, and the exceptions follow*.

Another example: as programs grow over time, periods of incremental, "organic" evolution are punctuated by refactorings.

Legal documents similarly tend to grow organically, as exceptions are piled upon exception to accommodate previously unanticipated scenarios; an experienced drafter reading a text will often remark to themselves, "ah, this was written first, and then this other bit was added later; you can tell."

In software, refactoring efforts are facilitated by the compiler and by test suites, which give confidence that the refactored code still works.

In law, as Ken Adams (MSCD) and Claire Hill (Why Contracts are Written in Legalese) observe, refactorings tend to be rarer: lawyers can be superstitous about "time-tested" phrasing, which has been before a judge and given objective interpretation.

This shows that "objective truth" tends to be expensive in law. Human lawyers advise human clients that the ultimate truth requires a human judge, and that in turn often requires a full-fledged lawsuit.

2.6 How L4 can help

We address some of the above difficulties by presenting L4, a DSL for law that allows some form of “”agreed truth”” to be calculated through computational means, rather than disputed before a judge: if the logic is explicit, the facts can be affirmed, and the outputs can be verified, then the meaning of a legal construct should be accessible to anyone with a computer.

This approach, while not universally applicable, can be useful in many situations where a dispute has not yet arisen, and parties are working in good faith toward a meeting of the minds.

Still, we regard it as a starting point and invite exploration and elaboration.

CHAPTER

THREE

FLOWCHARTS: PRACTICAL CONSIDERATIONS FROM A SOFTWARE ENGINEERING PERSPECTIVE

3.1 Flowcharts representing both statics and dynamics

Flowcharts are commonly taught in school. So, when formalizing the law, people often reach first for flowcharts.

The flowcharts approach is the least common denominator, used when other tools are not available, or when staff are not trained in them.

They use flowcharts to represent the statics: these are really decision trees.

They use flowcharts to represent the dynamics: these are really state diagrams.

These dual aspects tend to be highly coupled into a single flowchart.

The flowcharts tend to be drawn in a simple drawing app, at the level of syntax, not semantics. Boxes with text, arrows with nearby text.

From such a flowchart, a web interview is produced, in a direct and imperative form.

The output of the interview feeds into a document assembly system.

Often, such documents are intended to be finally embodied on paper.

3.2 Adding functionalities to flowcharts

A less imperative and more declarative approach preserves all the features of flowcharts, while adding functionality.

Statics are extracted to decision tables and diagrams, in DMN / DRD format.

Dynamics are left in the flowchart, and upgraded with swimlanes and deadlines to look more like BPMN format.

At this point we can reconstruct the original flowchart, but from a less tightly coupled source.

From these elements we can extract an interview.

The results of the interview can be submitted digitally.

Isomorphism can be preserved when the authoritative version of the rules are written in a higher-level language.

3.3 Why a functionality enhanced flowchart is preferable

Flowcharts are not the best way to represent decision logic. “The sorrow is obvious,” says the Camunda DMN tutorial.

This approach is more amenable to automated verification. What edge cases are lacking?

With this approach, interfaces can be made interactive. Irrelevant subtrees trees can be folded away hidden from view.

The end-user can trace their way through the flowchart and examine alternative routes.

The availability of user-provided information does not always match the pre-determined order of questions.

CHAPTER

FOUR

COMMON EXPRESSIONS

This section gives examples of how common terms in computer science are used in L4.

Note that boldface and italicization does not matter to L4.

4.1 Simple Terms, Parties and Entities, and the AKA Keyword

Work in progress

4.2 Arrays AKA MultiTerm

```
Arrays      AKA     MultiTerm

    One or more cells on a single line:                     Foo     Bar     Baz
    The cells can contain strings or numbers:               1       2       3
    Single cells are fine:                                  Quux
```

4.3 Dictionaries AKA Parametric Text

One or more lines of arrays. Typically used to give detail to actions. Multiple indentation styles are supported. Here, all lines are indented at the same level.

```
Dictionaries    AKA     Parametric Text

Perform             some action
to                  some standard
with                some detail
in the amount of    some money
```

It is also possible for the 2nd and subsequent lines to be indented relative to the first line. This is fine. But they all have to be indented to the same depth.

```
Pay                 the fee
amount              2000
currency            USD
recipient           Vendor
```

Some conventions like to use prepositions for the keys. But, as you can see, that's not required.

The terms below "Pay", namely "amount", "currency" and "recipient", are commonly called "keys".

The terms below "the fee", namely "2000", "USD", and "Vendor", are called "values".

Together they are called "key/value pairs", "attributes", or "parameters".

The first line is special: in an action spec, it tends to follow the format of "Verb object".

Programmers may find this idiom familiar if they think it as a function call with parameters.

Positional parameters go on the first line, and named parameters go on the subsequent lines.

Note that currently nested dictionaries are not supported. We may add support in a future version of the language.

4.4 Boolean Structures AKA BoolStruct

A BoolStruct of some Thing connects multiple Things with AND, OR, and NOT operators.

Nesting is supported.

```
    Foo
AND Bar
AND         Baz
    OR      Quux
```

You can build a BoolStruct of Arrays

```
            is warm blooded
AND         is vertebrate
AND         milk production
AND                 has hair
            OR      has fur
AND                 live births
            OR      monotreme
```

You can also build a BoolStruct of Dictionaries. However, in BoolStructs, dictionaries must have their subsequent lines indented

```
            Red
                    hex FF0000
OR          Green
                    hex 00FF00
OR          Blue
                    hex 0000FF
```

Finally, you can also build a BoolStruct of RelationalPredicate. An example is currently in Work in progress.

4.5 Relational Predicates

A relational predicate can be intuitively understood through the following example:

Suppose you have two terms X and Y. They can be compared for equality (Eq) and along some dimension (Ord).

```
X IS Y
X <  Y      X <= Y
X >  Y      X >= Y
```

Maybe Y is an array:

```
Y  IS  Y1  Y2  Y3  Y4
```

Then we can see if X is in it:

```
X IN Y
```

The examples above show Relational Predicates as Constraints.

4.5.1 Relational Predicates inside Horn Clauses

Relational Predicates are used inside Horn Clauses. The head is a Relational Predicate. The body is a BoolStruct of Relational Predicates.

The usage looks like the following example:

```
DECIDE      foo     IS      bar
WHEN        baz     IS      quux

which parses to

HC2    { hHead = RPConstraint ["foo" ] RPis ["bar"]
       , hBody = Just (Leaf (RPConstraint ["baz"] RPis [ "quux" ]))}
```

But a RelationalPredicate can also contain types we are already familiar with:

- MultiTerm arrays
- ParamText dictionaries

These are used as atomic terms.

```
DECIDE      foo
WHEN        baz IS quux

which parses to

HC2    { hHead = RPMT ["foo"]
       , hBody = Just (Leaf (RPConstraint ["baz"] RPis ["quux"]))}
```

```
DECIDE      foo
WHEN        baz

which parses to
```

(continues on next page)

(continued from previous page)

```
HC2    { hHead = RPMT [ ""foo"" ]
       , hBody = Just (Leaf (RPMT ["baz"]))}
```

```
DECIDE      Color   IS      blue
WHEN        baz

which parses to

HC2    { hHead = RPConstraint ["Color"] RPis ["blue"]
       , hBody = Just ( Leaf (RPMT ["baz"]))}
```

Simpler cases allow for more straightforward, unconditional definitions. You may think of these as variable assignments to values.

```
DECIDE      Color   IS  blue

which parses to

HC2    { hHead = RPConstraint ["Color"] RPis ["blue"]
       , hBody = Nothing}
```

A variety of syntaxes parse to the same Horn Clause constructs.

```
HC2    { hHead = RPConstraint ["Color"] RPis ["blue"]
       , hBody = Nothing}

The following syntaxes parse to the above Horn Clause construct.

DECIDE      Color   IS      blue

DECIDE      Color
MEANS       blue

DECIDE      Color   IS      blue

DEFINE      Color   IS      blue

DEFINE      Color   MEANS   blue

DEFINE      Color
MEANS       Blue
```

4.6 The TYPICALLY and AKA Keywords

This is Work in progress.

4.7 Rule Labels and Scope Qualifiers

This is Work in progress.

CHAPTER

FIVE

META-RULES,DEFEASIBILITY, AND DEFAULTS

The following treatment of meta-rules, defeasibility, and default logic arose as a research output in the course of performing the Rule 34 PoC.

We borrow from Default Logic the notion of priorities, as given in Brewka 2000 and Brewka 1994.

Notwithstanding

Let us consider Example Def-1:

```
Clause A    No integers are prime.

Clause B    Notwithstanding Clause A, some integers are prime:
            if a positive integer has no factors besides 1 and itself,
            then it is prime.
```

In this example, the default is Clause A, and the priority relation is B < A, meaning B describes exceptions to A.

Subject To

Let us consider Example Def-2:

```
Clause A    Subject to Clause B, no integers are prime.
Clause B    Some integers are prime:
            if a positive integer has no factors besides 1 and itself,
            then it is prime.
```

In this example, the default is Clause A, and the priority relation is B < A, meaning B describes exceptions to A.

Symmetry

In both examples above, the effective semantics are the same: B < A.

The examples differ only in the choice of which clause explicitly states the priority relation.

Operationalization

The above examples reduce to the same program:

```
isP n
|   n > 0 &&
    primeFactors n `remove` 1 `remove` n == [] = True
|   Otherwise                                  = False
where
    a `remove` b = filter (/= b) a
```

The priority relation is made explicit by the ordering.

The "otherwise" line gives the default answer "False", and is placed last.

The exception is tested before the default, and thus is placed above.

Scope

Note that priority relations do not wholly knock out the overridden clause.

Clause B is only dispositive when there are no other factors besides 1 and n.

If there are no other factors, Clause B is not dispositive, so the default applies.

Let us advance through the history of mathematics to 1938, and add:

```
Clause C    Notwithstanding Clause B, we deem 1 to be non-prime.
```

This clause C does not override clause B in full: it only removes a single element 1 from the domain.

```
isP n
|   n == 1                                          = False
|   n > 0 &&
    primeFactors n `remove` 1 `remove` n == []      = True
|   Otherwise                                       = False
where
    a `remove` b = filter (/= b) a
```

Above, the priority clause (""notwithstanding B"") was placed in clause C.

But we could also have placed the priority clause inside Clause B:

```
Clause B            Subject to Clause C, ...
Clause C            We deem 1 to be non-prime.
```

The Defeasibility Graph

The above examples are deliberately simple, and create a linear chain of overrides: C < B < A.

These overrides, ordered for correct execution in a program, constitute a topological sort of a graph.

One could imagine a more complex set of overrides, with two main branches rooted at X:

```
A < E < I < O < U < X       \\the "vowel branch"
B < C < D < F < G < X       \\the "consonant branch"
```

Such a graph admits multiple topological sorts.

In terms of Logic Programming

We can map the above example to logic programming.

If the body of a Horn clause is satisfied, we say the rule applies.

The head of the Horn clause is then evaluated (or unified), and gives the effect of the rule.

In the above examples, first we decide if each clause applies; if it does, the rest of the clause gives a True/False answer, which is its effect.

In terms of traditional programming

Conventional programming languages use the IF/THEN construct.

The IF part tests if the rule applies. The THEN part gives the effect.

CHAPTER

SIX

LANGUAGE REFERENCE

6.1 Types of Expressions in L4

This Backus-Naur Form (BNF) cheatsheet describes the syntax of the L4 language.

6.2 Top Level

```
Toplevel::= Regulative Rule
        |   Constitutive Rule
        |   Type Declaration
        |   Variable Definition
        |   Full Scenario Rule
        |   Simple Scenario Rule
```

6.3 Rules

6.3.1 Full Scenario Rule

```
Full Scenario Rule::=       [Toplevel...] \\(except Scenario Rules)
                            \\this allows us to reduce the ruleset as more and more data␣
↪is available
```

6.3.2 Simple Scenario Rule

```
Simple Scenario Rule::=     SCENARIO    RuleName    \\(except Scenario Rules)
                            GIVEN   RelationalPredicate
                                    [       ...       ]
                            EXPECT  RelationalPredicate
                                    [       ...       ]
```

6.3.3 Regulative Rule

```
Regulative Rule     ::=     EVERY | PARTY   Entity Label
                                  [Subject Constraint]
                                  [Attribute Constraint]
                                  [Conditional Constraint]
                                  [Upon Trigger]
                                      Deontic Action Temporal | Deontic Temporal Action
                                  [HENCE           Rule Label | Regulative Rule]
                                  [LEST            Rule Label | Regulative Rule]
                                  [WHERE           Constitutive Rule
                                                          [...]                       ␣
↪    ]
```

6.3.4 Constitutive Rule and Hornlike Rule

Hornlike clauses have the form: Head if Body

```
Constitutive Rule ::= [GIVEN  MultiTerm]
Hornlike Rule     ::= [Upon   Trigger  ]
                        DECIDE          Relational Predicate  [AKA Alias] [Typically␣
↪Boolish]
                      | IS              BoolStructR
                      | MEANS           BoolStructR
                      | HAS             Relational Predicate
                      | INCLUDES        Set Group
                        WHEN            RelationalPredicate BoolStruct
```

6.3.5 Compact Constitutives

```
Compact Constitutives ::= [GIVEN        MultiTerm]
                          [Upon Trigger          ]
                          DECIDE        Relational Predicate    WHEN        Relational␣
↪Predicate
                                                                            [ ... ]
                                    |   Relational Predicate    OTHERWISE | GENERALLY
```

6.4 Labels and Names

```
Entity Label    ::= Aliasable Name

Aliasable Name  ::= MultiTerm [AKA MultiTerm]
// in future - extend to BoolStruct of SetGroup
```

6.5 Constraints and 'Upon Trigger'

```
Subject Constraint      ::= WHO             RelationalPredicate BoolStruct
\\evaluated against the subject of the rule

Attribute Constraint    ::= WHOSE           RelationalPredicate BoolStruct

Conditional Constraint  ::= (WHEN | IF)     RelationalPredicate BoolStruct
                            [UNLESS         RelationalPredicate BoolStruct]
```

```
Upon Trigger ::= UPON               Aliasable Name
```

6.6 Deontics

```
Deontic Temporal Action     ::=     Deontic Keyword             Temporal Constraint
                                    -> | DO                     Action Expression

Deontic Keyword             ::=     (MUST | MAY | SHANT)
```

A semantically equivalent syntactic alternative allows the temporal keyword to line up with the other keywords:

```
Deontic Action Temporal ::= Deontic Keyword             Action Expression
                            Temporal Constraint
```

```
Temporal Constraint     ::= (BEFORE | AFTER | BY | UNTIL)   Temporal Spec
```

6.7 Dictionaries and Key/Value expressions

```
Action Expression   ::= Dictionary
example                     pay     vendor
                            amount| $20
                            by    |cheque

Dictionary          ::= Detail Key/Value
                            [     ...       ]

Detail Key/Value    ::= Single Term             [MultiTerm]
                            [newline indented       [Dictionary]  ]

Detail Key          ::= Single Term
```

6.8 Single Term and MultiTerm

```
Single Term         ::= a string or number within a single cell

MultiTerm           ::= Single Term     [Single Term...     ]
```

6.9 Type Declaration and Variable Definition

```
Type Declaration    ::= DECLARE     MultiTerm       [Type Signature ]
                        HAS         MultiTerm       [Type Signature ]
                                                    [ ... ]
example                 DECLARE     Point
                        HAS         position x      IS A Number
                                    position y      IS A Number

Variable Definition ::= DEFINE      Value Term      [Type Signature] //class-object␣
↪instantiation
                        HAS         MultiTerm       [Type Signature]
                                                    [ ... ]
```

6.10 Booleans and BoolStructs

```
Boolish              ::= (TRUE | FALSE | Yes | No)

BoolStruct Expression::= Expression
"BSE"                  | BSE AND BSE
                       | BSE OR  BSE
                       | NOT     BSE
                       | (Expression)

BoolStructR          ::= BoolStruct      Relational Predicate
```

6.11 Relational Predicate

This section is to be rewritten.

6.12 Value Term and Set Group

This is Work in progress.

CHAPTER

SEVEN

SYNTAX REFERENCE BY KEYWORD

7.1 WHOSE

7.1.1 WHOSE in a regulative rule

The "WHOSE" keyword can appear at the top level in a regulative rule, where it acts as a qualifier constraint.

```
EVERY   P
WHOSE   attribute   predicate              [TYPICALLY Boolean-expression]
WHOSE   color   IS  blue
```

The WHOSE line adds a precondition to the rule. If the WHOSE block does not return a true result, the rest of the rule does not proceed.

The attribute term is interpreted with respect to the party P.

The predicate takes up the rest of the line and applies to the attribute. As with most predicates, a TYPICALLY default can be supplied to improve UX.

This is operationally equivalent to:

```
function rule (..., party, attribute, value,... ) {
    if (! predicate(party[attribute]) { return }
}
```

and is logically equivalent to (See swipl dicts for syntax):

```
rule(..., Party, Attribute, Predicate, ...) :-
call(Predicate, Party.Attribute), ...
```

7.1.2 WHOSE in top-level constitutive definitions

The "WHOSE" keyword can appear in a top-level constitutive definition, where it acts as a qualifier constraint.

```
DEFINE      Retriever
IS A        Dog
WHOSE       Breed   IS IN   Chesapeake Bay          Golden
                            Curly-Coated            Labrador
                            Flat-Coated             Nova Scotia Duck Tolling
```

A Retriever is a Dog whose attribute Breed matches one or more of the elements given in the following list.

If the Breed attribute is itself a list, then the test is a set intersection.

If the Breed attribute is not defined, the test is negative.

See remarks about **vacuous truth**.

7.1.3 WHOSE in inline constitutive definitions

The "WHOSE" keyword can appear in an inline constitutive definition in a regulative rule, where it acts as a qualifier constraint.

```
EVERY   Dog Walker
MUST    muzzle  their   Dog
                          WHOSE   Breed   IS IN   Pit Bull
                                                  German Shepherd
```

Assuming the MUST does not contain any AND or OR branches, this is effectively similar to saying, at the top level,

```
WHEN    Dog Breed   IS IN       Pit Bull German Shepherd
```

Because the WHOSE does not appear under an AND/OR/XOR limb, the qualifier attaches to the top-level rule, and voids the entire rule if the constraint is not met.

7.1.4 WHOSE in a junction list

The "WHOSE" keyword can appear under a limb of a junction list, where it acts as a qualifier constraint on the associated limb.

```
        Motorcycle
MEANS   Two-wheeled     vehicle     equipped with an    internal combustion engine
OR      Two-wheeled     vehicle     equipped with a     battery-powered motor
        WHOSE           maximum speed   >               11  miles per hour
```

Because the WHOSE appears under an AND/OR/XOR limb, the constraint is ANDed within the nearest limb.

Internally, with the help of some DEFINE rules (shown below) the rule is transformed to:

```
    Motorcycle
MEANS       vehicle wheel count             IS      4
OR          vehicle wheel count             IS      2
    AND     vehicle engine                  IS      internal combustion engine
    AND     vehicle maximum speed           >       11 miles per hour
```

7.2 WHO

7.2.1 WHO in a regulative rule

The "WHO" keyword can appear at the top level in a regulative rule, where it acts as a qualifier constraint.

```
EVERY       P
WHO parameterizable attribute
```

The WHO line adds a precondition to the rule. If the WHO block does not return a true result, the rest of the rule does not proceed.

The parameterizable attribute term is interpreted with respect to the party P.

This is operationally equivalent to:

```
function rule (..., party, attribute, ...) {
    if (! party.attribute) { return }
}
```

and is logically equivalent to:

```
rule(..., Party, Attribute, ...) :-
call(Verb, Attribute), ....
```

There is some subtlety here: sometimes an attribute turns out to be a method, meaning a function that runs against the party, with arguments.

In other words, you might want:

```
function rule (..., party, attribute, attributeParameters, ...) {
    if (! party.attribute( attributeParameters )) { return }
}
```

The arguments are given as a dictionary of sub-attributes and predicates:

```
EVERY   P
WHO     attribute
            sub-attribute   predicate
            sub-attribute   predicate
```

This enables the more natural phrasing:

```
EVERY   P
WHO     runs
        with        scissors
        speed       >3 mph
```

7.3 DEFINE

```
DEFINE      F
GIVEN       P1      P2      P3

DEFINE      F1      F2
MEANS       something possibly involving F1 and F2

DEFINE              two-wheeled vehicle
MEANS               vehicle wheel count     IS   2

Note that the indentation follows the first word of the rewritten phrase.

DEFINE              vehicle equipped with an      X
MEANS               vehicle drive           IS    X
```

Note that you get a/an-equivalence for free, when it appears at the end of a cell, as above.

When a rewrite rule operates twice against the same sentence, on both the left and the right of the central term, the limbs are conjoined with an AND and reindented accordingly.

7.4 AS IN

This keyword is shorthand for importing a particular keyword block from another section.

Suppose we have:

```
§       Section One
PARTY   Seller
WHEN    sale        date    IS IN   promotional period
AND     sale        store   IS IN   stores participating in promotion
AND     blah        blah
MUST    do something
```

Rather than repeat all the WHEN bits,

```
§   Section Two
PARTY   Buyer
WHEN    AS IN       Section One
MUST    do something else
```

CHAPTER

EIGHT

ADVANCED TOPICS

This is a collection of deeper explanations on how the L4 language works. As of Jan 2023, this is mostly work in progress.

8.1 Natural Language Generation

This section will explain how questions are constructed in L4.

8.2 Planning Problems

This section will explain how L4 can be used for abductive reasoning. It will also discuss automatic generation of a dependency graph for corporate paperwork.

8.3 Developer IDE

This section will discuss the configuration of realtime feedback, syntax errors and linting advice for parser feedback, visualizers such as Petri Nets and Boolean Circuits, the web app, and Formal Verification backends.

8.4 Exporting from L4

The flowchart output provided by L4 can be exported into several different formats, shown below.

8.4.1 Exporting to LegalDocML

This section will discuss considerations regarding surface realizations of concrete syntax, generating valid LegalDocumentML, and Akoma Ntoso.

8.4.2 Exporting to LegalRuleML

This section will discuss interoperability considerations and semantic compatibility between languages and tools.

8.4.3 Exporting to a Logic Language

This section will discuss the paper The British Nationality Act as a Logic Language and its impact three decades after it was written.

8.4.4 Exporting to a Model Checker

This section will discuss the PDPA DBNO example in the LegalSS spreadsheet and discovering a race condition. For more information regarding the PDPA DBNO use case, refer to the Quickstart manual.

8.4.5 Exporting to Doc Assemble

Document Assembly using the output of an L4 interview.

This section will discuss direct generation using Mustache, Markdown, and Pandoc to HTML and Docx. It will also discuss exporting the entire interview to DocAssemble.

8.4.6 Exporting to Rule Engines

This section will discuss exporting to enterprise rule engines, including Clara and Drools.

8.4.7 Exporting to BPMN and DMN

This section will discuss open standards for business process and decision modeling.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search