HEINONLINE

DATE DOWNLOADED: Mon Oct 14 15:38:59 2024
SOURCE: Content Downloaded from HeinOnline

Citations:
Please note: citations are provided as a general guideline. Users should consult their preferred citation format's style manual for proper citation formatting.

Bluebook 21st ed.
Shai Revzen, Contract Drafting, Programming Languages, and What They Can Teach Each Other, 6 WAYNE ST. U.J. BUS. L. 64 (2023).

ALWD 7th ed.
Shai Revzen, Contract Drafting, Programming Languages, and What They Can Teach Each Other, 6 Wayne St. U.J. Bus. L. 64 (2023).

APA 7th ed.
Revzen, Shai. (2023). Contract Drafting, Programming Languages, and What They Can Teach Each Other. Wayne State University Journal of Business Law, 6, 64-85.

Chicago 17th ed.
Shai Revzen, "Contract Drafting, Programming Languages, and What They Can Teach Each Other," Wayne State University Journal of Business Law 6 (2023): 64-85

McGill Guide 9th ed.
Shai Revzen, "Contract Drafting, Programming Languages, and What They Can Teach Each Other" (2023) 6 Wayne St UJ Bus L 64.

AGLC 4th ed.
Shai Revzen, 'Contract Drafting, Programming Languages, and What They Can Teach Each Other' (2023) 6 Wayne State University Journal of Business Law 64

MLA 9th ed.
Revzen, Shai. "Contract Drafting, Programming Languages, and What They Can Teach Each Other." Wayne State University Journal of Business Law, 6, 2023, pp. 64-85. HeinOnline.

OSCOLA 4th ed.
Shai Revzen, 'Contract Drafting, Programming Languages, and What They Can Teach Each Other' (2023) 6 Wayne St UJ Bus L 64 Please note: citations are provided as a general guideline. Users should consult their preferred citation format's style manual for proper citation formatting.

Provided by:
University of Michigan Law Library

-- Your use of this HeinOnline PDF indicates your acceptance of HeinOnline's Terms and Conditions of the license agreement available at https://heinonline.org/HOL/License
-- The search text of this PDF is generated from uncorrected OCR text.

CONTRACT DRAFTING, PROGRAMMING LANGUAGES, AND WHAT THEY CAN TEACH EACH OTHER

SHAI REVZEN *

Although vastly disparate as disciplines, Law and Computer Science share a similar goal – using language to precisely specify behavior. The art of drafting a contract so as to avoid ambiguity and channel the performances into mutually agreed and understood bounds bears great similarities to the art of computer programming. Some of the tools and principles computer scientists have developed for the design of computer languages might be used for more effective contract drafting. I present a brief review of grammatical and referential ambiguity and its mitigation as it appears from the computer scientist's perspective, and relate these issues to well known case law, and drafting of large commercial contracts.

† Associate Professor of Electrical Engieering and Computer Science, University of Michigan. B.Sc. & M.Sc. Hebrew University, Jerusalem, 1993, 2003; Ph.D. Berkeley, CA. My deepest thanks to Vincent Wellman, and Eric Zacks for valuable feedback, and introducing me to legal publishing.

Table of Contents

I. INTRODUCTION..3

II. PROGRAMMING AS CONSTRAINTS ON COMPUTATION..................4

III. WHAT IS THE “MEANING” OF THIS?..6

A. Syntax, Semantics, and Grammatical Ambiguity.......................................9

1. The interaction of syntax and semantics..12

2. Compilation and Type Checking..13

3. Indexicles and Name Scoping..15

4. Binding and Capitalized Nouns..17

B. Inheriting, Default Clauses, and Object Oriented Type Theory......17

IV. TOTAL ORDERING – A RECOMMENDATION...................................20

1. A note about intent..22

V. CONCLUSION..22

I. INTRODUCTION

Law and theoretical computer science seem like disciplines that are rather distanced from each other. The one is venerably old, deeply tied into social institutions and the fabric of society, and almost entirely non-mathematical. The other is new – less than a century old – and was an outgrowth of modern mathematics. Yet both are concerned with the same matter – governing the actions of autonomous agents, be they people or computing devices.

In no field of law is the focus on governing actions clearer than in the drafting of contracts. Contracting parties, within some bounds defined by the prevailing legal system, make promises which are meant to govern their actions, presumably to attain some mutual benefit. While there are many approaches to the drafting of contracts, here we consider one of the most prevalent – that a contract be drafted primarily in a form expressing the procedure to be followed by the parties' performances.

Computer science also has a sub-field whose primary focus is governing the actions of agents. This is the field which focuses on the design and implementation of programming languages. A computer program is a specification of a computation, which is to govern the subsequent actions of the computer.

Much of the experience of programming, for those who learn its fundamentals, is that of struggling to adapt to the mindset of specifying the minutia of the computation to be performed with sufficient detail as to allow a computer to unambiguously perform such a computation. Yet few of the people who routinely program computers study the tools by which the programming itself is done, just as few of the people who employ contracts on an ongoing basis study contract drafting itself.

Our purpose in this paper is to investigate the parallels between the tasks of contract drafting, interpretation, and enforcement, and the tasks of program coding, parsing, and execution. After some preliminaries, we will focus on ambiguity – an issue that has bedeviled contract law for as long as humans have made promises – and the technical approach and tools computer science has brought to bear on the subject. We will show that the analogies run deeper than

has commonly been realized, and there may be much that the legal profession can borrow and benefit from.

II. Programming as Constraints on Computation

Modern computing devices are machines which rapidly manipulate a giant table of symbols encoded in a binary code, with the sequence of manipulations being itself coded in such symbols and stored in that table. Further, external devices such as sensors and communication channels inject symbols into some locations in this table. The contents of this table, the computer's memory, are virtually unintelligible to anyone except to the few programmers who work on or with the "machine code" that defines the actual symbol manipulations. Almost all computer programming is done in "high level languages" – strings that are far easier for humans to read and write – and then translated ("compiled") into machine code or interpreted directly by a computer program. An enormous number of machine operations is usually performed for every operation requested in the high-level language.

Broadly speaking, high level computer languages for general use fall into three categories: "procedural" languages, "functional" languages, and "logic" languages. The logic languages are programmed through declarative statements, arranged to ensure that in resolving the logical problem they represent the computer will perform the desired computation. In this sense they may seem the most like statutory and regulatory law - they declare what must be, rather than providing a recipe for accomplishing a specified goal. Since the "private law" of contracts is usually more forthcoming as to details of a performance, we will forgo the discussion of logic languages.

Functional languages focus on the abstract notion of functions as maps from inputs to outputs, providing constructs for combining, modifying, and manipulating functions, thereby allowing for computations to create and manipulate other computations. Conceptually and philosophically, they are the most interesting class of languages, since they deal with precise means of manipulating computational procedures. The reflective and self-descriptive capabilities of functional languages capture in a precise form some of the ability of natural language to discuss language itself. In analogy to law, a rule or regulation about the required structure of a

legislative or contract drafting process would perhaps be best analogized to a functional programming language.

However, the bulk of programming is done in procedural programming languages. These are conceptually the simplest and the kind most taught and used. In this they are a bit like transactional law - the workhorse by which law is used to organize interactions in society. A procedural language consists of methods for expressing a sequential process by which the computer memory and input are used and manipulated to produce the desired output. It describes a sought-after computation much like a contract might describe a sought-after performance[1].

Starting with the Smalltalk-80 language in the late 1970s and early 1980s, the idea of "object oriented" languages has come to dominate the way procedural and functional programming languages are designed. The core concept of object orientation is that data and the "methods" used to manipulate it are intricately connected and should therefore be tightly integrated. In object-oriented programming languages, the memory of the computer is organized into "objects", each of which belongs to a "class". The class collects all the methods used to manipulate the memory allocated to the object. This allows for the notion of a "sub-class". An object of the sub-class defaults to its "super-class(es)" for resolving method not defined in the sub-class. Consequently, the programmer can define her representation of the computation using a hierarchy of kinds of objects, and the methods by which those objects' "attributes" can be changed.

As an illustrative example, the procedural part of a high-level language like Python, which has object-oriented features, might include a statement "a = b + 5", which is interpreted as: (1) search for some object, i.e. block of data, associated with the symbol "b" in an index of symbols; (2) query the class of "b" for its method, i.e. procedural list of operations, associated with the symbol "+" when applied to an object, which in this case will be an object of the built-in integer class which will contain the value 5; (3) when you find such a method, perform the sequence of instructions it entails, possibly leaving a resulting object in a pre-negotiated location in

[1]. Most functional languages have a procedural subset, and many modern languages straddle the divide between functional and procedural for reasons that will become clearer when we discuss "type systems" below.

memory; (4) Generate an index entry for the symbol “a” and point from that symbol to the result.

The purpose of this example is not to educate the reader on the internal workings of Python, or any other computer language. Rather, it is to illustrate the fact that what happens is ambiguous and open to multiple interpretations. There are dozens, if not hundreds, of ways to organize tables in computer memory that are indexed by symbols like “a” and “b”. The operation symbolized by “+” can mean different things in different contexts; etc. Thus, even as simple a statement as “a = b + 5” in a well-defined programming language is in fact merely a contract for what will be done, and not an actual performance. It expresses a set of promises about obtaining a previously referenced object indicated by “b”, querying it for its meaning of “+” in the context of “5” using a previously defined set of agreements about how such a query is performed, and then making the result available using the label “a”. From the perspective of the computer’s hardware – the actual physical substrate in which the operations will be performed – this simple statement can correspond to a vast range of ways to perform the desired operations. The design of the computer language defines which performances (“executions”) are valid but leaves many of the details to the actual implementation of the language. Computer languages in common use can have hundreds of implementations, which would nevertheless produce closely equivalent execution.

While machine language programs specify in near-physical terms the symbol manipulation desired, high-level languages do not. Instead, they specify constraints that any execution of the program must satisfy.

III. WHAT IS THE “MEANING” OF THIS?

In the practice of law, the meaning of a contract is typically examined with respect to a specific controversy regarding performance. Using the law and various methods of construction, the officers of the court translate the language of the contract into a collection of factual questions which are brought to trial. The results of the findings of fact answer questions such as: “was there a contract?” and: “did this performance meet this contract?” Thus, for legal contracts, the “meaning” of a contract is understood by people and their interpretation of language of the contract, considering a

putative example of the parties' conduct which should have been governed by the alleged contract.

In computer science it has been shown that there is a world of difference between following the instructions of a computer program and checking whether a given input-output pair is consistent with a computer program. What is perhaps the greatest open question in computer science, and the foundation of virtually all modern cryptography, is the question "is P=NP?", "P" being the class of problems that can are effectively computable, and "NP" being the class of problems whose answers can be checked for correctness in by effectively computable means. The legal system's insistence on trying only matters under controversy mimics a deep fundamental issue regarding which questions are effectively answerable altogether. Asking whether a performance matches a contract is treating the problem of contract construction as a problem in NP.

Taking as given that construction of a contract is possible only considering putative performance, it is only natural to ask what the relationship ought to be between the contract language and the actions that comprise performance of that contract. Despite being a closely examined question in the legal literature, the study of this question in law rests upon the shaky foundation of "plain language interpretation". The meaning of language is itself a deep philosophical question, which the practicing computer scientist cannot afford to be mired in resolving. It is here – in the grounding of meaning in a carefully constructed and provably correct foundation – that the design of computer languages might provide its greatest benefits to legal contract drafting.

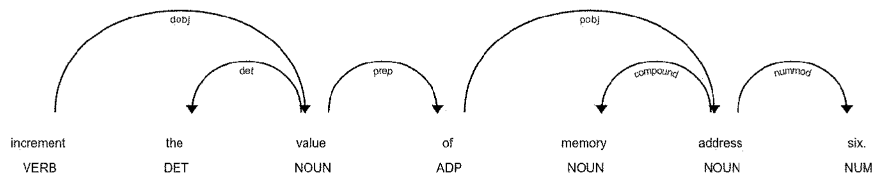
In computer science (and most other branches of mathematics) the notion of meaning is a relative one: X can mean Y if components of X interact with each other in a manner analogous to that by which the components of Y interact. In its broadest sense, this idea corresponds to the category theory notion of a "morphism", the subtle complexity of which is far outside the scope of this exposition. For purposes of our discussion here, we will use the computer science notion of "simulation" of "transition systems" as the relationship between a program and the computation it specifies.

A "transition system" is a collection of "states" and a collection of labelled "transitions" that are possible between those states. One

important example of a transition system has the states as the possible values stored in the computer's memory, and the transitions labelled by machine language commands that perform that command's transformation on the memory. This "Abstract Computer" transition system captures exactly what a correctly functioning modern electronic computer performs; in a very practical sense it "is" what a computer does when the electronics function as they were designed to.

Another important transition system is the transition system that has as its states all possible grammatically correct fragments of English language text and has as its transitions the grammatically correct ways by which they might be combined into longer fragments. This transition system captures all possible grammatical constructions of the English language; we will refer to this system as the "English Grammar" transition system.

Consider now a sequence of statements such as "increment the value of memory address six; copy the results to memory address five". The first part of this statement can be grammatically analyzed as follows:



Grammatical structure of a sentence, as produced by the spacy (http://spacy.io) natural language processing tool with the en_core_web_lg default configuration. We show the part of speech of each word, and draw arrows indicating the relationships between words. This diagram is a possible final state in a sequence of transitions of the English Grammar transition system whose underlying text is "increment the value of memory address six"

The formation of the words from the letters, and the grammatical relationships of the words from their sequence, eventually lead to a complete sentence in the "English Grammar" transition system. Sometimes, as in the case here, the sentence contains words that correspond to a description of a transition of "Abstract Computer" transition system. Thus a certain subset of transitions of the English Grammar systems is a faithful representation of the Abstract Computer transitions in the following sense: starting from any initial description of the world, which happens to be an abstract computer, one may follow a sequence of English language transitions manipulating that world, then as some point convert that representation into an abstract computer state, and follow the translated transitions of the Abstract Computer transition system. Regardless of when the transition from English Grammar to Abstract

Computer happened, the results will be the same. This sort of mathematical relationship is called a semi-conjugacy, or in the parlance of computer science transition systems a “simulation” relationship whereby the Abstract Computer is a simulation of some part of the English Grammar transition system.

It is important to note that the mapping from the English Grammar system to an Abstract Computer is not unique. Simulation relationships come at various levels of “refinement”, where the more refined simulations capture more information about the system being simulated, and coarser simulations capture less and less information. At the extremes of refinement are a least refined system, and a most refined system. The least refined system has a single state and a single transition as its simulation; this can be thought of as a state whose meaning is “the system is in some state”, and a transition whose meaning is “a transition happened” – it remembers nothing about the system it simulates or its history. The most refined transition system simulation has states corresponding to all possible finite transition sequences of the system it simulates. Its only transitions are the transitions that extend that history by a possible transition; thus it remembers everything that happened to the system being simulated. In that sense both the most refined “universal” simulation which can capture all meanings, and the least refined “unit” simulation which merely asserts existence, are un-informative. But in the space between then there exist a plethora of simulations, some classes of which are of particular value in the interpretation of computer programs.

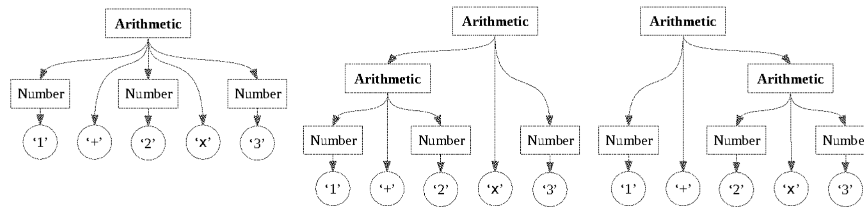
A. Syntax, Semantics, and Grammatical Ambiguity

When computer scientists wish to attribute a meaning to a computer program, they select several simulations of intermediate levels of refinement and use those to define the precise meaning of the computer language. This very formal process is necessary because unlike contract drafters, there is no “plain meaning” of the computer language text for them to fall back on. The choice of these simulations is guided by some of the most profound results in the theory of computation, and leads to the notions of syntax, semantics, and type-checking.

The most granular transition system that defines a computer language is usually its “syntax”. The need for syntax comes from the fact that while in theory one could attempt to define a computer language

where every possible finite sequence of symbols ("string") has meaning, this would likely be useless for humans. Thus, at the very least, we might hope for the computer to be able to identify which strings could possibly have a meaning (are "well formed") and which are clearly meaningless. We would hope for meaningful strings to follow some grammar sufficiently similar to human language to make this grammar intuitive to humans. Finally, we would like the grammatical analysis of these strings ("parsing") to be a fast and efficient computational process. All these requirements were answered together in the formation of formal language theory and the theory of compilation. This body of work, starting in the 1950s and still somewhat active today, discusses classes of "languages", where the quotes indicate the use of the technical term "language" in computer science – a designated sub-collection of the collection of finite strings. Thus a "language" is a mathematical object that can answer one question: "is this string in the language or not?" Formal language theory defines classes of languages using various representation of their grammatical rules, and identifies among them some languages that can be efficiently "parsed", i.e. for every string belonging to the language, a computer program could be written to efficiently reconstruct the grammatical rules by which that string can be derived.

Consider a simple example: specifying the "language" of those arithmetic computations in non-negative integers which use only addition, multiplication, and parentheses. We can represent that language as a sub-collection of strings formed from the "alphabet" of symbols {'+', 'x', '(', ')', '0', '1', … , '9'}. The string "1+17" is meaningful and part of the language we wish to represent; the string "++x)01" is meaningless and not part of the language. We could define this language as consisting of "valid expressions" which are either: (Number Rule, "N") any sequence of digits that does not begin with '0', because any number is a "valid expression"; (Arithmetic Rule, "A") a list of "valid expressions" separated by '+' or 'x' symbols; (Parenthesis Rule, "P") a "valid expression" preceded by '(' and followed by ')'. With this (recursive) definition, we will allow only those strings of "alphabet" symbols which are meaningful.



Three ways of deriving the string "1+2x3" from the grammatical rules provided. We denote alphabet symbols (circles with symbol inside) and the rules used to combine them (rectangles with rule name and arrows to the "valid expressions" and alphabet symbols to which the rule was applied). These can also be expressed in textual form (using the abbreviations listed in the rule names) as "A[N[1], +,N[2],x,N[3]]", "A[A[N[1],+,N[2]],x,N[3]]", and "A[N[1],+,A[N[2],x,N[3]]]". This is an example of syntactic ambiguity.

Consider now the string "1+2x3". This string follows the rules, as each number in the string is a valid expression according to the Number Rule, and the entire string is an application of the Arithmetic Rule to those valid expressions (see Figure, left-most diagram). Such a derivation of the rules as applied to the symbols that shows how the string could be obtained is called "(syntactically) parsing" the string. As it turns out, there are multiple ways in which the string "1+2x3" can be obtained from our grammar rules. This is referred to as "syntactic ambiguity" and is studiously avoided in the design of computer languages.

Lest it be presumed that syntactic ambiguity is not of great legal importance, consider an example provided by Ken Adams, of "A Manual of Style for Contract Drafting" fame[2]: in *O'Connor v. Oakhurst Dairy*, No. 16-1901, 2017 WL 957195 (1st Cir. Mar. 13, 2017) the First Circuit considered the meaning of the following: "The canning, processing, preserving, freezing, drying, marketing, storing, packing for shipment or distribution of:". We can express a "list" as consisting of: (Item Rule, "I") an item on the list; (Item with Purpose Rule, "P") an item followed by "for" and a list of purposes; or (List Rule, "L") a list of items separated by commas, the last of which may be separated by the words "and" or "or". With these rules, is the correct parsing of the O'Connor clause "L[I[canning], I[processing], I[preserving], I[freezing], I[drying], I[marketing], I[storing], P[packing for L[I[shipment]]], I[distribution]]" or is it "L[I[canning], I[processing], I[preserving], I[freezing], I[drying], I[marketing], I[storing], P[packing for L[I[shipment], I[distribution]]]]"? The

[2] Ken Adams, Why I Don't Pin My Hopes on the Serial Comma, Adams on Contract Drafting Blog (Mar 15, 2017) https://www.adamsdrafting.com/why-i-dont-pin-my-hopes-on-the-serial-comma/ .

former considers one kind of packing, only used for shipment; the latter has two kinds of packing – packing for shipping and alternatively packing for distribution. The very fact that the written notation of the parsing structure removed the syntactic ambiguity suggests that the formal language approach used in computer science may have some salubrious properties.

1. <u>The interaction of syntax and semantics</u>

Computer scientists use the term "semantics" to describe the process of producing a uniquely specified computation from the parsed computer language program. In our simple arithmetic language, this raises an immediate problem: the operations of ordinary arithmetic act on two numbers to produce a result, but the Arithmetic Rule allowed for multiple valid expressions to be combined, separated by multiple operations. To make it clear how to produce a conventional arithmetic semantic for the language we specified, we could amend the Arithmetic Rule "A" to: (Binary Operator Arithmetic Rule, "B") a pair of "valid expressions" separated by an operator which is one of '+' and 'x'. The string "1+2x3" now has only two parsing options: "B[N[1],+,B[N[2],x,N[3]]]" and "B[B[N[1],+,N[2]],x,N[3]]".

We can define the semantics associated with the revised arithmetic language as a computation that provides a value for any valid expression. For each of our rules we define a computation: the value of a Number Rule is the number obtained from interpreting the digits as a decimal number; the value of a Parenthesis Rule is the value of the valid expression in the parenthesis; the value of a Binary Operator Arithmetic Rule is the sum of its constituent values if the operator was '+' and the product of its constituents if the operator was 'x'.

Sadly, our syntactic ambiguity produces a semantic ambiguity: is the value of "1+2x3" the number 9 or the number 7? This depends on which of the two ambiguous choices of parsing we used. The convention that multiplication takes precedence over addition has not been encoded in our method of parsing and evaluating (or "executing") valid expressions.

This precedence problem was traditionally resolved by further refinement of the Binary Operator Arithmetic Rule into rules for products and sums. A complete set rules could define a "valid expression" as: (Sum Rule, "S") a "product expression", possibly

followed by '+' and a "valid expression". We also define a "product expression" as: (Multiplication Rule, "M") an "element", possibly followed by 'x' and a "product expression". Finally, we define an "element" as: (Number Rule, "N") any sequence of digits that does not begin with '0'; or (Parentheses Rule, "P"), a "valid expression" preceded by '(' and followed by ')'. These rules allow parsing of "1+2x3" into the unique form "S[M[N[1]],S[M[N[2],N[3]]]]"

Taking a step back and examining the bigger picture of syntactic ambiguity, it becomes clear that a big part of the problem comes from attempting to define a language independently from the how that language is parsed. When the parsing mechanism and the rules of the language are defined together, using e.g. Parsing Expressing Grammars[3], this can lead to efficient and unambiguous parsing. Using an expressly stated provably unambiguous grammar could completely remove an entire class of ambiguities that appear in contract drafting.

Unfortunately, resolving syntactic ambiguities is only the first step towards obtaining a clear meaning. The famous linguist Noam Chomsky gave the example sentence[4] "[c]olorless green ideas sleep furiously" as an example of a syntactically sound sentence which is semantically nonsensical and meaningless. How then can we ensure that no nonsensical clauses are present?

2. Compilation and Type Checking

One of the most profound insights in the theory of computation is the "halting problem". This result states that it is impossible to create a computer program that can identify all computer programs which get stuck in infinite loops. The relevance of this result to the design of programming languages and the drafting of contracts is that it is provably impossible to create a procedure that will identify all bugs in programs, or by analogy, all flaws in contracts. Part and parcel with realizing the futility of a general solution to the problem, comes the understanding that there is room for approximate solutions of various forms.

---

[3] Bryan Ford, *Parsing Expression Grammars: A Recognition Based Syntactic Foundation*, Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 111–122 (2004) https://dx.doi.org/10.1145/964001.964011

[4] Noam Chomsky, Syntactic Structures (1957)

The process of parsing the language of a program and identifying the rules by which that particular expression in the language was formed, can, as we saw before, assign semantics for evaluating the expression, equivalent specifying a class of correct performances of the contract. While the halting problem tells us that any sufficiently powerful semantics becomes impossible to test for bugs, it also suggests that we may choose some intermediate semantics which both provably provide an answer and simulate the true semantics we want. These kinds of intermediate semantics are called "type-systems" in computer science.

For example, assume we have language for describing the breeding of domesticated animals, with the ability to express the actions such as Breed(X,Y) to produce a set of offspring, Raise(X) to raise a juvenile to breeding age, and Cull(X) to cull or harvest the animal. It would be patently idiotic to have a procedure that contains Breed(Romeo, Juliette) if Romeo is a horse and Juliette a chicken. Regardless of what operations we do on animals (absent very significant genetic manipulation) we cannot breed unrelated species. Thus, we could have types like "horse" and "chicken", and the semantics of Breed(horse, horse)=horse, Breed(chicken, chicken)=chicken, Breed(horse, chicken)=ERROR, Raise(chicken)=chicken, Raise(horse)=horse, and, importantly, any function of ERROR is ERROR. Evaluating our breeding program with these semantics will detect all species mismatches, and evaluate nonsensical breeding plans to ERROR.

Like all simulations, type-systems can be more refined or less refined in their resolution of details. The type-system above does not detect errors that involve trying to breed newly hatched, immature chicks with each other. We could refine it further, by adding types like "foal" and "chick", that are converted to "horse" and "chicken" using the Raise() function. We could even add additional semantics, such as numerical ages, sexes, etc. There is a continuum of type systems, ranging from the trivial type-system which certifies as good any expression that can be parsed, to some (as of yet unknown, possibly unknowable) maximally refined type-systems that are still weak enough to not be subject to the halting problem.

One of the ways in which the study of computer languages could inform the drafting of contracts is in suggesting that large, complex contracts contain an expressly defined type-system for their

interpretation. While this type-system would not fully specify the performances governed by the contract, it could greatly enhance the readers' ability to identify problematic procedures and provisions.

3. Indexicles and Name Scoping

When expressing ideas in a language, it is often impractical or impossible to refer to every object every time with some fully unambiguous form of reference. Even in what is conventionally seen as unambiguous use, the object will often be uniquely identified once and then referred to with "indexical" words, such as "that", "whom", etc. It is left to the reader to deduce from the text whether the "that" of one clause refers to the same as the "this" of another clause. In contract drafting, it is quite common to use capitalized nouns instead of indexicals – terms like "Buyer" and "Seller", or the "Parties" are used to refer to entities defined elsewhere in the document – thereby reducing some of the ambiguity.

More specifically, we can identify several possible forms of ambiguity associated with indexicals: (1) the indexical could refer to more than one possible definition, and those definitions are mutually inconsistent; (2) what the indexical is referring to is never defined; (3) two or more indexicals refer to the same number of referants but which indexes which is unclear.

Computer languages make extensive use of indirect references to objects, primarily through the use of "identifiers" which are labels the programmer can use to refer to objects. These can be "variable names" which refer to values, and "methods names" or "function names", which refer to sequences of operations the computer could be required to execute. Computer languages used for large projects are usually "strongly typed" - they have a detailed type-system that is strictly enforced. Part of that enforcement is that the type associated with the values of every identifier is declared before that identifier is used, and the scope in which the identifier can be used is made expressly clear.

In human languages, indexicals are disambiguated using attributes such as gender and plural or singular. For example, the English sentences: "The father, brothers, and sisters must contact the insurance provider. They will only receive service if they provide photo ID." are ambiguous: does "they" refer to all the people in the

first sentence, to only the siblings, to only the brothers, or only the sisters? In English, "they" refers to a plural and so could not refer to the father alone; in Hebrew and Arabic the "they" would be gendered and could be used to refer unambiguously only to the sisters, but not unambiguously only to the brothers; and in other languages the indexical could be absent a distinction between plural and singular. When drafting contracts between people with different native languages, the ambiguity of indexicals becomes an even more complex multi-cultural issue.

Because of the various ambiguities that natural language indexicals can introduce, and the multi-cultural nature of many business contracts, it is probably a good idea to ensure that indexicals are used very sparingly, if at all. If they are used, it should be within a very immediate scope, such as referring within a sentence or to the previous sentence.

While it may seem that the issues arising from the use of indexicals could be addressed by a purely syntactic scan for a short list of words with that grammatical role, the famous case of *Raffles v. Wichelhaus*, EWHC Exch J19 (1864), also known as the "Peerless" case, suggests otherwise. In this case, parties contracted to carry goods on a ship named the "Peerless". However, there were two ships with that name, taking this same route several months apart. The goods arrived with the later ship, causing significant losses to the contracting merchants, giving rise to the legal controversy in the case – who must bear the cost of the wrong Peerless? Whenever even the slightest ambiguity exists in a contract, sufficiently interested parties could pursue litigation.

For our purposes, we should note that even the use of a proper noun cannot prevent referential ambiguity. Referential ambiguity in computer languages is usually resolved by a combination of two tools: a scoping rule which defines the scope in which every identifier can be used, and a name table which ensures that within its scope every identifier is unique. It is similarly advisable to use provably unique identifiers in contract drafting.

4. Binding and Capitalized Nouns

Unlike natural language indexicals, capitalized nouns are a construct of contract law. In well-written contracts, they are declared well before they are used, and the type of objects they refer to is made clear. Yet problems arise when additional contracts are included by reference, which is a common practice in commercial drafting, e.g. including an already existing confidentiality agreement in a service contract.

The same problem arises in computer programming, when one function wishes to use another function as part of its computation. The value referred to by the identifier 'cow' in the using function ("caller"), might be referred to as 'animal' in the function being used ("sub-routine"). In computer languages every "invocation" of a function includes the process of "binding" its parameters. An express and unambiguous correspondence is formed by the semantics of the language that defines which names in the caller correspond to which names in the sub-routine. More importantly, it is expressly clear that these are the only correspondences. If the sub-routine has variables that have the same names as variables in the caller, they are treated as separate variables. If both have a variable 'x' which is not a parameter, these are treated as 'caller-x' and 'sub-routine-x' - two completely independent variables.

The bindings in computer languages are usually expressed as part of the process of invocation of the sub-routine. By analogy, this would suggest that rather than using contract language like "the terms of the Confidentiality Agreement (annex A) will apply to the parties", it would be advisable to draft language like "the terms of the Confidentiality Agreement ("CA", annex A) will apply to Contractor as the CA Employee, and the Company as the CA Employer; no other capitalized terms are in correspondence between this Agreement and the CA." This kind of language expressly binds the relevant indexicals, and expressly frees all other terms as unrelated, precluding conflicts of definition.

B. Inheriting, Default Clauses, and Object Oriented Type Theory

In the drafting of contracts under the UCC it is common to have default provisions come into force unless the contract expressly states otherwise. In more complex commercial contracts, it is common to have contracts that include several other form contracts, possibly with overlapping provisions. Previously we addressed the issue of binding the capitalized nouns of the included contracts and clarifying

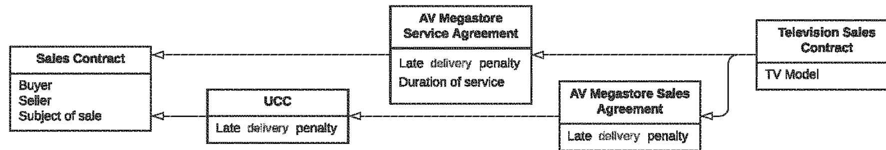
the scope of definition of such nouns. Here we address the trickier question of which default provisions come into force when included contracts and the contract including them have provisions that may apply.

The use of default methods is a key idea in Object Oriented Programming, which is by far the dominant framework used in procedural programming languages. All fully object-oriented languages define a mechanism of "inheritance" that allows objects of a sub-class to default to using methods of their super-class. As an example, there could be a class of "Sales of Goods contracts", with a sub-class "UCC sales contracts", and the sub-class of that as the "Detroit AV-Megastore sales contracts". A particular instantiation of the latter would be a specific purchase from that store. Any provisions missing from the AV-Megastore contracts would default to using the UCC provisions that apply to the same matter. The general structure of a contract for the sale of goods would be in the top-level Sales of Goods class.

The equivalent structure exists in law and resolves matters in essentially the same way as the programming example would. The general structure of a contract for the sale of goods is established common law. The UCC laws of the jurisdiction provide the UCC provisions and the document containing the AV-Megastore's sales contract overrides some of those provisions to establish a form contract for the store with its customers.

Complications arise when this simple hierarchy of inheritance is disrupted by "multiple inheritance", e.g. the AV-Megastore sells a TV with a bundled installation service, and the "Television Sales Contract" stipulates that it follows the provisions of the "AV-Megastore Service Agreement" and the provisions of the "AV-Megastore Sales Agreement", which happen to overlap and disagree on some matter such as the late delivery penalties and remedies.

A possible object-oriented class inheritance structure for this could be (where arrows point from sub-class to super-class):



The problem to resolve would be which of the three "Late delivery penalty" resolutions will be applied - UCC 2-601, Sales Agreement, or Service Agreement. While it is clear that the AV-Megastore Sales Agreement method overrides the UCC provisions, it is not clear whether to inherit the Service Agreement method or the Sales Agreement method. Here different programming languages adopt different conventions, but all programming languages define and enforce an unambiguous resolution to this problem. Perhaps the easiest of these solutions is that found in the Python language, which searches for methods in the order in which the super-classes are listed and uses the first match found.

Language designers also needed to face an additional complication – the "Sales Contract" super-class is shared. Are the attributes inherited through each inheritance path the same or different? In the case at hand, we intend the Buyer and Seller to be the same, but the Subject of sale to be a service in one path and a TV in the other path. Properly annotated for this choice, the Television Sales Contract class would define attributes: Sales Contract Buyer, Sales Contract Seller, Sales Contract Subject of Sale through Service Agreement, Sales Contract Subject of Sale through Sales Agreement, Service Agreement Duration of Service, and Television Sales Contract TV Model.

The problem of attributes inherited through multiple distinct inheritance pathways is similar to the problem of variable scope and bindings which we previously discussed. Just as variables are assumed local unless bound, multiply inherited attributes can be assumed different unless expressly unified.

Viewed from a broader perspective, the procedure that associates with each object its classes, attributes, and methods, is a choice of type-system. Type checking an object-oriented program requires "resolving" the types of all objects, and then testing that all methods and all attributes mentioned do exist and permit the operation or access that was requested.

Several important ideas can be borrowed from object-oriented type-systems to the domain of contract drafting. First, when including other contract language by reference, it must be clear by what order of precedence the provisions of included contracts should apply. Second, if any capitalized nouns appear in the included contracts, they should be expressly bound in the including contract, and all unbound capitalized nouns should be expressly unbound from each other.

IV. TOTAL ORDERING – A RECOMMENDATION

The problems of contract construction ambiguity discussed so far often boil down to the need to resolve which of several possible interpretations should apply. Taking the collection of all provisions in a contract, one may wish to define a "takes priority over" relation between them. Then, among the provisions that apply to a matter at hand, the highest priority provision, if such can be found, should apply. The discussion is somewhat muddied because often provisions provide independent effects and can be applied in any order of priority to produce the same results.

Borrowing the language of transition systems above, we can take provisions to be states, and transitions to be made from a provision to the provisions that take priority over it. The highest priority provision can only be resolved if the transition system does not have any repeating loops. Such transition systems naturally arise when defining computations that are guaranteed to terminate. These structures are also called "partial orderings" in mathematics. Partial orderings and their transition system admit refinements, with a maximally refined mathematical object in the class being a "total ordering" – one where the question of priority can be answered between any two provisions.

The most familiar total ordering is that of real numbers, but numbering provisions in a contract with real numbers to indicate their priority might be less than ideal. Thankfully, a more natural choice exists – "lexicographic ordering" – which is the ordering used when sorting words in the dictionary. This is also the ordering used when sorting dot-separated numbered paragraphs. It provides, for example, that any number starting with 2.2 would come before numbers starting with 2.3, and after number starting with 2.1. Thus, e.g., 2.2.89 is "prior to" 2.3.1, and "after" 1.9.9.8.

Unless otherwise specified, the order in which the provisions of a contract are listed has no role in conventional contract construction. This provides an opportunity – to use the sequential order of provisions to provide the total ordering of their priority. Just as Python's multiple inheritance resolves potential conflicts using the order of the super-classes, it may be possible to add language such as "if two numbered provisions of this Agreement are found to be in conflict, the provision with the earlier number shall preferentially be enforced". This would make little difference under most circumstances, but it would also provably preclude any ambiguity in choosing between competing provisions.

An additional benefit of this approach is that it allows the numbering of an included contract to be inserted into the including contract by reference, thereby resolving questions of priority that may arise from multiple inheritance. Language such as "the numbered provisions of the Included Agreement shall be deemed prefixed with 2.12." would provably suffice to resolve all conflicts of priority between provisions of the included contract and the provisions of the including contract.

Furthermore, if the practice of assigning a priority number to provisions takes root, it is conceivable that it will take on some of the formal dimension of contract formation[5] in that it will become clear that a provision can only be formally included in a contract when it is properly numbered with a priority.

The rules of contract construction under, e.g. the UCC, already contain conventions of priority, e.g.. UCC 2-303(e) "[…] (1) express terms prevail over course of performance, course of dealing, and usage of trade; (2) course of performance prevails over course of dealing and usage of trade; and (3) course of dealing prevails over usage of trade." The UCC priority rules could be captured by assigning constructive provisions to the course of performance, course of dealing, and usage of trade. These constructive provisions will be numbered in the order listed above.

---

[5] Lon L. Fuller, *Consideration and Form*, Columbia Law Review, Vol. 41, No. 5 799-824 (1941) https://doi.org/10.2307/1117840

1. A note about intent

The astute reader may ask why we have given little to no attention to the question of the contract drafters' intent. It is because the post-hoc question of determining intent in court is usually an indicator of an ambiguity in the contract. Regardless of the drafter's actual intent, if the contract is unambiguous, it is unlikely that a legal controversy will arise. The party whose wishes do not correspond to this postulated unambiguous contract language will need to negotiate for a resolution from an inferior bargaining position or perform best they can. The role of the contract itself is, among other things, to make such legal controversies as unlikely as possible.

V. CONCLUSION

There is a close relationship between the problems faced by designers of programming languages, and lawyers who consider the practice of contract drafting. Due to the highly mathematical nature of computer science, its practitioners have considered mathematically provable methods for resolving problems of ambiguity, some of which could be carried over into law.

We reviewed some of the formal language and tools used in computer language design. From these, we highlighted some practical recommendations: (1) to make extensive use of properly defined capitalized nouns, (2) bind them clearly when contract provisions are included by reference, and (3) to clearly decouple and unbind all other terminology that was not expressly bound. Furthermore, we suggested the use of (4) a numbering scheme and related express rule that earlier numbered provisions should dominate any later numbered provisions. This scheme should then be used to expressly insert any contract language included by reference into the numbering scheme.

The use of these recommendations can provably prevent any ambiguity in deciding which provision dominates if a conflict arises in the construction of the contract. These recommendations can also make it easier and safer to include external language by reference in a contract – a common practice in many commercial settings.