# A Programming Language for Future Interests

**Shrutarshi Basu**[*]
**Nate Foster**[†]
**James Grimmelmann**[‡]
**Shan Parikh**[§]
**Ryan Richardson**[¶]

*Learning the system of estates in land and future interests can seem like learning a new language. Scholars and students must master unfamiliar phrases, razor-sharp rules, and arbitrarily complicated structures. Property law is this way not because future interests are a foreign language, but because they are a* programming language.

*This Article presents Orlando, a programming language for expressing conveyances of future interests, and Littleton, a freely available online interpreter (at* `https://conveyanc.es`*) that can diagram the interests created by conveyances and model the consequences of future events. Doing so has three payoffs. First, formalizing future interests helps students and teachers of the subject*

[*] Postdoctoral Fellow in Computer Science, Harvard University. This work was supported by NSF Award FMitF-2019313. Versions of this work were presented at the 2017 Roundtable on Computer Science & Law at the University of Pennsylvania, the 2017 Domain-Specific Language Design and Implementation workshop, the 2018 Internet Law Works in Progress conference, the 2019 SPLASH Onward! conference, a 2021 Cornell Law School faculty workshop, and the 2022 ACM Workshop on Programming Languages and Law. We are grateful for their comments to the participants, and to Aislinn Black, Shawn Bayern, Sara Bronin, Charles Duan, Kate Klonick, Sarah Lawsky, Denis Merigoux, Christina Mulligan, Beth Noveck, Paul Ohm, Eduardo Peñalver, Jeremy Sheff, Emily Sherwin, Henry Smith, Jeffrey Stake, Stewart Sterk, Rebecca Tushnet, Laura Underkuffler, and Reid Weisbord. This Article is available under the Creative Commons Attribution 4.0 International license, https://creativecommons.org/licenses/by/4.0.

[†] Associate Professor of Computer Science, Cornell University.

[‡] Tessler Family Professor of Digital and Information Law, Cornell University.

[§] Software Engineer, Google.

[¶] Software Engineer, Oracle.

*by allowing them to visualize and experiment with conveyances. Second, the process of formalization is itself deeply illuminating about property doctrine and theory. And third, the computer-science subfield of programming language theory has untapped potential for legal scholarship: the programming-language approach takes advantage of the linguistic parallels between legal texts and computer programs.*

Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 76
I Programming Languages and Law . . . . . . . . . . 83
A Contract . . . . . . . . . . . . . . . . . . . . . . . . 87
B Tax . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
C Legal Drafting . . . . . . . . . . . . . . . . . . . . 91
D Visualization . . . . . . . . . . . . . . . . . . . . 94
II An Informal Overview . . . . . . . . . . . . . . . . . 100
A Previous Work . . . . . . . . . . . . . . . . . . . . 100
B Orlando and Littleton . . . . . . . . . . . . . . . 102
C An Example . . . . . . . . . . . . . . . . . . . . . 107
III The Formal Details . . . . . . . . . . . . . . . . . . 110
A Title Trees . . . . . . . . . . . . . . . . . . . . . . 112
B Semantics . . . . . . . . . . . . . . . . . . . . . . . 114
C Conveyances . . . . . . . . . . . . . . . . . . . . . 120
D Translation . . . . . . . . . . . . . . . . . . . . . . 123
E Conclusion . . . . . . . . . . . . . . . . . . . . . . 125
IV Lessons for Property Law . . . . . . . . . . . . . . 126
A Design Principles . . . . . . . . . . . . . . . . . . 127
1 Orlando . . . . . . . . . . . . . . . . . . . . . 128
2 Littleton . . . . . . . . . . . . . . . . . . . . . 130
B Insights into Property Doctrine . . . . . . . . . 132
1 Defaults . . . . . . . . . . . . . . . . . . . . . 132
2 Syntactic Ambiguity . . . . . . . . . . . . . . 134
3 "Theorems" of Property Law . . . . . . . . . 136
C Insights into Property Theory . . . . . . . . . . 138
1 The *Numerus Clausus* . . . . . . . . . . . . . 138
2 Recursivity . . . . . . . . . . . . . . . . . . . 140
3 Modularity . . . . . . . . . . . . . . . . . . . 141
Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . 144
Appendix: Orlando Reference . . . . . . . . . . . . . . . 147

## Introduction

> *The formulas that govern future interests are similar to those of chemistry. They seem to be more of the law of nature than law of men except for one crucial difference: The rules of future interests occasionally make no sense.*[1]

> *Though of feudal origin, it is not a relic of barbarism, or a part of the rubbish of the dark ages. It is part of a system; an artificial one, it is true, but still a system, and a complete one.*[2]

> *The logician must be rather like a lawyer . . . in the sense that he is there to give the metaphysician . . . the tense-logic that he wants, provided that it be consistent. He must tell his client what the consequences of a given choice will be . . . and what alternatives are open to him . . . .*[3]

Every law student and every law professor has a different reaction on reaching the unit on estates in land and future interests in Property. For some, it is the worst part of the course. They find the system of reversions, possibilities of reverter, and remainders vested subject to complete divestment to be an alien language: dull, desiccated, and divorced from the practical realities of the rest of law.[4] For others, it is the best part of the course. Here, there are no counter-arguments and indeterminate multi-factor tests, only rigorous deduction and clear right answers.[5]

[1] Daniel B. Bogart, *A Casebook for Teaching Teachers: Jesse Dukeminier and James E. Krier, Property*, 22 Seattle U. L. Rev. 921, 933 (1998).

[2] Hileman v. Bouslaugh, 13 Pa. 344, 351 (1850).

[3] Arthur Prior, Past, Present, and Future 59 (1967).

[4] *See, e.g.,* Palma Joy Strand, *We Are All on the Journey: Transforming Antagonistic Spaces in Law School Classrooms*, 67 J. Legal Educ. 176, 182 (2017) ("T&E has the reputation of being moldy and covered in cobwebs, akin to and perhaps even more arcane than the future interests of property law."); Bogart, *supra* note 1, at 935 ("At some point, that teacher will have to train students to do the hard and frustrating mechanical work of future interests.").

[5] *See, e.g., Volume 63 Joint Dedication*, 63 S.D. L. Rev. i, ix (2018) (statement of Barry R. Vickrey) ("Some of my most enjoyable times at USD in-

These two groups, polar opposites though they may be in their approach to law school, share an intuition: there is something logical and computational about estates and future interests. Whether they want the computer to serve as a junior associate that calculates the consequences of conveyances so they don't have to, or as a sparring partner that plays along with them, they share the sense that there is something about this particular system of legal doctrines that makes them particularly suited for automated algorithmic analysis. A life estate and a remainder fit together like a lock and a key, with the mathematical certainty that establishes $2 + 2 = 4$. *Couldn't someone program a computer do this?*

We did.

Our system, called Littleton,[6] can interpret stylized conveyances like `O conveys to A for life, then if B is married to B, but if B divorces to C` and correctly report that B holds a contingent remainder in fee simple subject to executory limitation. It knows that O holds an implied reversion; that if B marries while A is alive then B's remainder is upgraded from contingent to vested subject to complete divestment; and that if A conveys their interest to D for life, then D's interest will be limited to the shorter of A's and D's lifetimes. It can even apply the Rule Against Perpetuities to strike interests that could vest too remotely.

We designed Littleton to be useful to teachers trying to explain the system of future interests and to students trying to learn it. We have put a web version online at https://conveyanc.es. Just type a conveyance in the box, click on "Interpret," and Littleton will display an easy-to-understand diagram of the resulting interests. It comes with documentation and a tutorial of demonstration conveyances, and has been validated against examples drawn from one of the leading student guides, Linda

volved discussions and sometimes debates with Chuck about the most arcane aspects of the law of estates in land and future interests."); Byron S. White, *Tribute to Myres S. McDougal*, 66 MISS. L.J. 1, 2 (1996) ("Future interests *a la* McDougal was pure fun.").

[6] After Thomas de Littleton, author of the *Treatise on Tenures* (ca. 1481–82), an important codification of the doctrines of estates in land and future interests. *See* THOMAS LITTLETON, LITTLETON'S TENURES IN ENGLISH (Eugene Wambaugh ed., 1903) (1481) (translation of the *Tenures* and a biographical sketch).

Edwards's *Estates in Land and Future Interests*.[7] We have also placed Littleton's source code online, and released it under the permissive MIT license, allowing anyone to use and improve it however they want.[8]

But that's not even the interesting part.

Rather than using an existing programming language to write a program to model future interests, we treated the formalized, ritualized language of first-year Property conveyances as a programming language itself. Each term in this language, which we call Orlando,[9] has a precisely specified syntax and semantics. The expression `O conveys to A` in Orlando is like `x = y * 4` in a traditional programming language like Python, Java, or C: a command that causes the computer interpreting it to update its state in a predictable, objectively determined way.

This makes Orlando into a a ***domain-specific language*** (or "DSL").[10] Just like JavaScript is useful for writing interactive web pages, Ink[11] and Inform[12] and Twine[13] for creating text adventure games, Solidity for smart contracts,[14] or Flash for an-

[7] Linda Edwards, Estates in Land and Future Interests (3rd ed. 2009); *see* Shrutarshi Basu, Nate Foster & James Grimmelmann, *Property Conveyances as a Programming Language*, 2019 Proc. 2019 ACM SIGPLAN Int'l Symp. on New Ideas New Paradigms & Reflections on Programming & Software (Onward!) 128 [hereinafter Property Conveyances] (describing test suite).

[8] *See The MIT License*, Open Source Initiative, https://opensource.org/licenses/MIT.

[9] After Orlando Bridgeman, one of the most important conveyancers in the common-law tradition, who drafted the instrument at issue in the case that created the Rule Against Perpetuities. *See* The Duke of Norfolk's Case, 22 Eng. Rep. 931 (Ch. 1682); *see also* Virginia Woolf, Orlando: A Biography (1928); Orlando (Sony Pictures Classics 1992).

[10] *See generally* Arie van Deursen, Paul Klint & Joost Visser, *Domain-Specific Languages: An Annotated Bibliography*, SIGPLAN Notices., June 2000, at 26 (overview of DSLs); Martin Fowler, Domain-Specific Languages (2010) (textbook on DSL design and implementation).

[11] Ink, https://www.inklestudios.com/ink/.

[12] Inform 7, http://inform7.com.

[13] Twine, https://twinery.org.

[14] Solidity [hereinafter Solidity], https://docs.soliditylang.org.

imations,[15] Orlando is a language for expressing property conveyances.

Drawing on the computer science discipline of programming language theory, we treat Orlando like any other DSL.[16] Littleton's processing is divided into stages:

- First, Littleton parses a conveyance written in Orlando, recognizing the individual clauses and their relationship.The language `O conveys to A for life, then to B`, for example, consists of two separate grants, linked by `then`. The first has a quantum (`for life`) attached to it; the second does not.
- Next, Littleton creates a data structure (which we call a ***title tree***) that keeps track of the current interests and their relationships. The title tree corresponding to

  ```
  O conveys to A for life, then to B until C marries.
  ```

  is shown in Figure 2.
- Littleton then applies substantive rules of property law to update the title tree as further events occur. That is, while the syntax of Orlando is given by the stylized language used in conveyances, Orlando's semantics are those of property law.
- Littleton analyzes the title tree in accordance with various rules used by lawyers and law students, so that the various interests can be properly named. For example, it classifies remainders as contingent or vested based on whether a condition precedent must be satisfied before that node in the title tree can be reached.
- Finally, Littleton displays the current state of the title by rendering the title tree in a graphical format that hides many of

[15] *But see* Steve Jobs, *Thoughts on Flash*, APPLE.COM (Apr. 2010), https://web.archive.org/web/20200430094807/https://www.apple.com/hotnews/thoughts-on-flash/.

[16] Computer *programming* is distinct from the field of *programming languages*. The former is the engineering practice of implementing useful software systems. The latter is an academic discipline that studies the characteristics of programming languages themselves, often using mathematical tools. They stand in roughly the same relationship as legal practice and legal theory. On programming languages, *see generally* ROBERT W. SEBESTA, CONCEPTS OF PROGRAMMING LANGUAGES (10th ed. 2012); SHRIRAM KRISHNAMURTHI, PROGRAMMING LANGUAGES: APPLICATION AND INTERPRETATION (2d ed. Apr. 4, 2017), https://cs.brown.edu/courses/cs173/2012/book/book.pdf.

Orlandus Bridgeman Miles et Baronettus,
Custos Magni Sigilli Angliæ

Orlando Bridgeman (1606–1674)

Thomas de Littleton (ca. 1407–1481)

Figure 1: Orlando and Littleton's namesakes

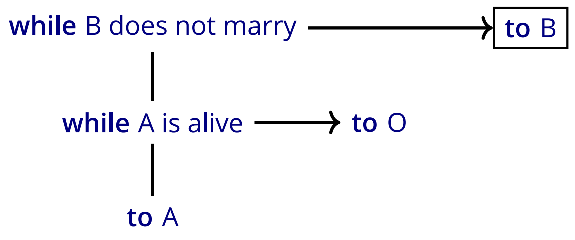


Figure 2: Orlando title tree for `O conveys to A for life, but if B marries to B.`

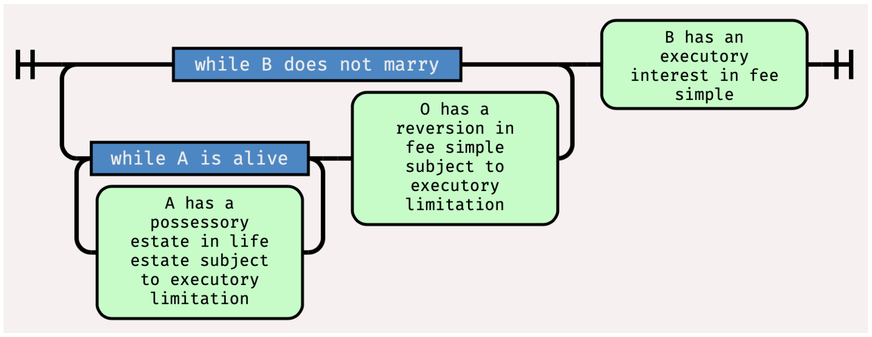

while B does not marry
B has an executory interest in fee simple
O has a reversion in fee simple subject to executory limitation
while A is alive
A has a possessory estate in life estate subject to executory limitation

Figure 3: Littleton output

the internal details and emphasizes the viable interests and the conditions on those interests. The resulting visualization is designed to be readily comprehensible to lawyers and law students who need not be aware of the sophisticated processing taking place under the hood. Figure 3 shows an example of Littleton's output.

Treating conveyances as a programming language yields insights into property doctrine, into property theory, and into legal theory more broadly. Doctrinally, Orlando brings the entire system of future interests into clearer focus by capturing the linguistic structure of property grants in a succinct and intuitive way. A confusing mess of doctrinal minutiae resolves itself into an orderly collection of well-specified rules. Facts about conveyances that previously became apparent only after detailed study are now immediately obvious—for example, that a grantor can recursively stack up an indefinite number of successive life estates. It is even possible to prove "theorems" of property law, such as that a fee simple is forever.

Theoretically, the fact that this fragment of property law can be formalized in this way is striking: other areas, like trademark law or international human rights law, almost certainly cannot. Orlando's simple but generative structure provides a new kind of support for a line of scholarship, associated with Thomas Merrill and Henry Smith and with the New Private Law movement, that emphasizes the modular and standardized elements in property's conceptual structure. For example, Orlando's design embodies the *numerus clausus* principle: that property interests only come in a finite set of forms.

Finally, Orlando is a proof by example that legal scholars can learn from programming-language theory. Law and programming languages can be to law and computers as law and linguistics is to law and language: a subfield that draws on the insight of another discipline to identify and systematize recurring structures of pervasive importance to law. The linguistic parallel between the natural languages of law and the artificial languages of software offers a fresh way to reflect on how law, lawyers, and legal texts work. In property and beyond, defining a programming language to model a body of law should be part of legal scholarship's methodological toolkit.

This Article provides a detailed exposition of a core subset of Orlando and Littleton, and a discussion of why they matter to legal scholars.[17] Part II introduces Orlando informally; Part III explains the formal details underneath the surface. Part IV discusses the design philosophy of Orlando and Littleton to show how they hold lessons for property law and property theory. And Part I surveys the scattered scholarship applying programming languages to law to argue that other scholars should consider creating their own legal DSLs.

## I. Programming Languages and Law

Computerizing legal reasoning is by no means new. There is a long-standing research program on the use of artificial intelligence (AI) systems for other areas of law. It has proceeded along two tracks, corresponding to the division within AI between systems using formal logical reasoning, sometimes called

[17] See the Conclusion for a list of additional features implemented in the full versions of Orlando and Littleton.

"symbolic" AI or "good old fashioned AI" (or GOFAI), and systems using statistical methods, sometimes called "subsymbolic AI" or, more recently, "machine learning."[18] Legal scholars draw on both tracks.[19] Orlando is squarely in the former tradition, so we focus on it here.

The use of AI systems to automate logical legal reasoning goes back decades.[20] Many scholars, legal-automation companies, and even teams of students have built "expert systems" that can walk the user through a questionaire to help them un-

[18] For a thorough history of AI discussing the interplay of these two traditions, see MARGARET A. BODEN, MIND AS MACHINE: A HISTORY OF COGNITIVE SCIENCE (2008).

[19] *See generally* KEVIN D. ASHLEY, ARTIFICIAL INTELLIGENCE AND LEGAL ANALYTICS: NEW TOOLS FOR LAW PRACTICE IN THE DIGITAL AGE (2017) (broad overview of both fields); MICHAEL A. LIVERMORE & DANIEL N. ROCKMORE, LAW AS DATA: COMPUTATION, TEXT, & THE FUTURE OF LEGAL ANALYSIS (2019) (recent collection on state of the art in statistical methods); Trevor Bench-Capon, *The Need for Good Old Fashioned AI and Law*, *in* 2020 INT'L TRENDS LEGAL INFORMATICS: A FESTSCHRIFT FOR ERICH SCHWEIGHOFER 23 (recent discussion of the division).

[20] *See, e.g.,* L. Thorne McCarty, *Reflections on TAXMAN: An Experiment In Artificial Intelligence And Legal Reasoning*, 90 HARV. L. REV. 837 (1976) [hereinafter *TAXMAN*]; John T. Welch, *LAWGICAL: An Approach to Computer-Aided Legal Analysis*, 15 AKRON L. REV. 655 (1981); John P .Finan, *LAWGICAL: Jurisprudential and Logical Considerations*, 15 AKRON L. REV. 675 (1981); Marek J. Sergot, Fariba Sadri, Robert A. Kowalski, Frank Kriwaczek, Peter Hammond & H. Terese Cory, *The British Nationality Act as a Logic Program*, 29 COMM. ACM 370 (1986); J.M. Trevor Bench-Capon, Gwen O. Robinson, Tom W. Routen & Marek J. Sergot, *Logic Programming for Large Scale Applications in Law: A Formalisation of Supplementary Benefit Legislation*, *in* 1987 PROC. 1ST INT'L CONF. ON A.I. & L. 190; Richard S. Gruner, *Sentencing Advisor: An Expert Computer System for Federal Sentencing Analyses*, 5 SANTA CLARA COMPUT. & HIGH TECH. L.J. 51 (1989); Cary G. Debessonet & George R. Cross, *An Artificial Intelligence Application in the Law: CCLIPS, A Computer Program that Processes Legal Information*, 1 HIGH TECH. L.J. 329 (1986); Phan Minh Dung & Giovanni Sartor, *The Modular Logic of Private International Law*, 19 A.I. & L. 233 (2011). For a good survey of the work through the 1980s, see Edwina L. Rissland, *Artificial Intelligence and Law: Stepping Stones to a Model of Legal Reasoning*, 99 YALE L.J. 1957 (1990); for a more recent survey see Henry Prakken & Giovanni Sartor, *Law and Logic: A Review From an Argumentation Perspective*, 227 A.I. 214 (2015). Although through the 1980s much of this research appeared in general-interest law reviews, most of it is now published in specialized journals such as *Artificial Intelligence and Law*.

derstand the application of a given body of law to their individual situation.[21] These programs range from simple decision trees up through complex tax preparation software. They are essentially hard-coded versions of a Choose Your Own Adventure, Mad Libs, or Excel spreadsheet, designed to slot the user's answers into the right blanks, with branching and calculations as needed to handle compound legal rules. More ambitiously, scholars have used increasingly sophisticated and powerful logics to model legal rules and sometimes to automate legal analysis.[22] These rule-driven systems typically have a knowledge base of legal rules encoded in a standard logical form and then use a search strategy to deductively derive valid legal conclusions on the basis of those rules. They have greater capacity to make chains of inferences and understand cascading consequences of interacting facts. Research in this tradition aims not just to understand the doctrines of legal fields, but also to formalizing the particular concepts of legal reasoning themselves, such as the elements of a cause of action, the scope of precedent, burdens of proof and presumptions, defenses, and defeasible conclusions.[23]

---

[21] *See, e.g.,* Hellawell, Robert, *A Computer Program For Legal Planning And Analysis: Taxation Of Stock Redemptions*, 80 COLUM. L. REV. 1363 (1980) (tax treatment of stock redemptions); Robert Hellawell, *CHOOSE: A Computer Program for Legal Planning and Analysis*, 19 COLUM. J. TRANSNAT'L L. 339 (1981) (tax planning for mining transactions); Elizabeth Townsend Gard, *The Durationator® Copyright Experiment*, *in* 2013 PROC. MEMORY WORLD DIGITAL AGE: DIGITIZATION & PRESERVATION 46 (copyright durations); Josh Goldfoot, SENTENCING.US: A FREE U.S. FEDERAL SENTENCING GUIDELINES CALCULATOR, https://www.sentencing.us (federal Sentencing Guidelines calculator). *See generally* Richard Gruner, *Thinking Like A Lawyer: Expert Systems For Legal Analysis*, 1 HIGH TECH. L.J. 259 (1986) (mid-1980s overview of state of the art in legal expert systems).

[22] *See, e.g.,* Layman E. Allen, *Symbolic Logic: A Razor-Edged Tool for Drafting and Interpreting Legal Documents*, 66 YALE L.J. 833 (1957) (propositional logic); *TAXMAN*, *supra* note 20 (predicate logic); Sarah B. Lawsky, *A Logic for Statutes*, 21 FLA. TAX REV. 60 (2017) [hereinafter *A Logic for Statutes*] (default logic); L. Thorne McCarty, *A Language for Legal Discourse I.: Basic Features*, 1989 PROC. 2ND INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE & LAW 180 (modal logic).

[23] *See, e.g., A Logic for Statutes*, *supra* note 22; Walter G. Popp & Bernhard Schlink, *Judith, A Computer Program to Advise Lawyers in Reasoning a Case*, 15 JURIMETRICS J. 303 (1974); L. KARL BRANTING, REASONING

What is most novel in Orlando and Littleton is the idea that programming languages have something distinctive to add to this research program.[24] To date, law and legal theory have engaged only intermittently with programming languages.[25] By far the most common point of contact is intellectual property. Whether a computer program is copyrightable or patentable depends on what a computer program is, and this is a question that cannot be answered sensibly without exploring the nature of the programming language it is written in.[26] Sometimes, it is the language itself that is the subject of an intellectual property claim, as in *Google v. Oracle*.[27] Similar issues arise in determining the

WITH RULES AND PRECEDENTS: A COMPUTATIONAL MODEL OF LEGAL ANALYSIS (2013); JAMES POPPLE, A PRAGMATIC LEGAL EXPERT SYSTEM (1996).

[24] A subsidiary point is that by isolating a well-defined fragment of law that is more amenable to formalization, our approach sidesteps some of the well-known challenges to the expert-systems approach to law. *See* Philip Leith, *The Rise and Fall of the Legal Expert System*, 30 INT'L REV. L. COMPUTERS & TECH. 94 (2016).

[25] *But see* Antônio Carlos da Rocha Costa, *Situated Legal Systems and Their Operational Semantics*, 23 A.I. & L. 43 (2015) (presenting ambitious operational semantics of Hans Kelsen's theory of legal systems).

[26] *See generally* Pamela Samuelson, Randall Davis, Mitchell D. Kapor & Jerome H. Reichman, *A Manifesto Concerning the Legal Protection of Computer Programs*, 94 COLUM. L. REV. 2308 (1994); Pamela Samuelson, *Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement*, 31 BERKELEY TECH. L.J. 1215 (2016); BEN KLEMENS, MATH YOU CAN'T USE: PATENTS, COPYRIGHT, AND SOFTWARE (2005); Peter D. Junger, *You Can't Patent Software: Patenting Software Is Wrong*, 58 CASE W. RES. L. REV. 333 (2007); Sebastian Zimmeck, *Patent Eligibility of Programming Languages and Tools*, 13 TUL. J. TECH. & INTELL. PROP. 133 (2010).

[27] Oracle Am., Inc. v. Google LLC, 886 F.3d 1179 (Fed. Cir. 2018). *See generally* Dennis S. Karjala, *Oracle v. Google and the Scope of a Computer Program Copyright*, 24 J. INTELL. PROP. L. 1 (2016); Marci A. Hamilton & Ted Sabety vol, *Computer Science Concepts in Copyright Cases: The Path to a Coherent Law*, 1997 HARV. J.L. & TECH. 239; Richard H. Stern, *Copyright in Computer Programming Languages*, 17 RUTGERS COMPUT. & TECH. L.J. 321 (1991). *See also* Michael Adelman, *Constructed Languages and Copyright: A Brief History and Pooposal for Divorce*, 27 HARV. J.L. & TECH. 543 (2013) (copyright in constructed natural languages).

scope of First Amendment coverage for software; the linguistic aspects of software are inescapable.[28]

In this Part, we sketch the engagement of law with programming languages in four: law, contract law, tax law, legal drafting, and visualization of law. What unites them is that in each domain, legal scholars have made meaningful progress by expressing legal relationships as a programming language. Sometimes they have used existing languages; sometimes they have created their own. In the right domains, this is a useful tool for gaining insights into legal doctrines and concepts.[29]

## A. *Contract*

Contract law is a good doctrinal fit for what programming languages can do.[30] Parties enjoy substantial autonomy to customize the terms of their contractual obligations, so the flexibility offered by programming languages is appealing. At the same time, contracting parties often want certainty about the meaning and effects of their contracts, so the clarity and precision of programming languages is also appealing. Thus, several scholars have discussed the prospects for expressing contract terms directly in computer-standardized forms.[31] From the computer-science side, there has been extensive work on finding appropriate logics to model contractual relationships.[32]

[28] *See, e.g.,* Lee Tien, *Publishing Software As A Speech Act*, 15 BERK. TECH. L.J. 629 (2000).

[29] We leave for another day the broader jurisprudential questions of what law and legal theory can learn from programming languages in general.

[30] *See* Harry Surden, *Computable Contracts*, 46 U.C. DAVIS L. REV. 629 (2012); *see also* Erik F. Gerding, *Contract as Pattern Language*, 88 WASH. L. REV. 1323 (2013). Surden describes contracts that have been expressed in computer-processable form as "data-oriented," which captures the fact that every computer program, by virtue of the fact that it can be stored on and processed by a computer, is also data. The duality between code and data is central to computer science.

[31] *See* Surden, *supra* note 30; Lawrence A. Cunningham, *Language, Deals, and Standards: The Future of XML Contracts*, 84 WASH. U. L. REV. 313 (2006). For a recent survey of approaches, see LSP Working Group, *Developing a Legal Specification Protocol* (2019), https://www-cdn.law.stanford.edu/wp-content/uploads/2019/03/LSPWhitePaperJan1119v021419.pdf.

[32] *E.g.,* Shaun Azzopardi, Gordon J. Pace, Fernando Schapachnik & Gerardo Schneider, *Contract Automata*, 24 A.I. & L. 203 (2016).

One notable project in using programming-language theory to model contracts is 2000's *Composing Contracts: An Adventure in Financial Engineering* by Simon Peyton Jones, Jean-Marc Eber, and Julian Seward.[33] It describes a carefully crafted library of primitive operators to model option contracts. Once standard contracts are encoded in this way, it becomes possible to do sophisticated financial analyses on them automatically, for example, computing the expected value of a contract that depends on changes in interest rates over time. There are now dozens of domain-specific languages for contracts.[34]

This approach is reflected in Orlando. Although property is a different problem domain than contracts, we followed *Composing Contracts*'s design principle of finding a minimal set of simple orthogonal primitive operators in creating the set of title tree nodes.[35] We also adopted a similar language choice; *Composing Contracts*'s system is implemented in Haskell, which like OCaml is a strongly typed polymorphic functional language with pattern-matching.[36]

More recently, there has been an explosion of interest in creating "smart" "contracts." [37] These are programs that adjust

[33] Simon Peyton Jones, Jean-Marc Eber & Julian Seward, *Composing Contracts: An Adventure in Financial Engineering*, 2000 ICFP0�0 280; *see also* Patrick Bahr, Jost Berthold & Martin Elsman, *Certified Symbolic Management of Financial Multi-Party Contracts*, *in* 2015 PROC. 20TH ACM SIGPLAN INT'L CONF. ON FUNCTIONAL PROGRAMMING 315; Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue & Christian Stefansen, *Compositional Specification of Commercial Contracts*, 8 INT'L J. ON SOFTWARE TOOLS FOR TECH. TRANSFER 485 (2006).

[34] *See* FIN. DOMAIN-SPECIFIC LANGUAGE LISTING, http://www.dslfin.org/resources.html (directory of projects).

[35] *See* Shrutarshi Basu, Anshuman Mohan, Nate Foster, & James Grimmelmann, *Legal Calculi*, *in* 2022 PROGRAMMING LANGUAGES & L. (PROLALA) (discussing this design principle for legal programming languages).

[36] There is another line of influence here. The OCaml libraries Littleton is built with are developed and maintained by Jane Street Capital, a quantitative trading firm. *See* Yaron Minsky & Stephen Weeks, *Caml Trading: Experiences with Functional Programming on Wall Street*, 18 J. FUNCTIONAL PROGRAMMING 553 (2008) (describing Jane Street's adoption of OCaml).

[37] *See generally* Shaanan Cohney & David A. Hoffman, *Transactional Scripts in Contract Stacks*, 105 MINN. L. REV. 319 (2020); Usha R. Rodrigues, *Law and the Blockchain*, 104 IOWA L. REV. 679 (2018); Jason G. Allen, *Wrapped and Stacked: 'Smart Contracts' and the Interaction of Natural and*

the relationship between multiple parties and various resources automatically. Programs require programming languages, which are supported by a blockchain or other digital platform. These platforms typically have a basic virtual machine—effectively a shared, simulated computer—and one or more general-purpose programming languages.[38] Numerous groups interested in developing applications that can displace the need for traditional legal contracts, or integrate smoothly with legal contracts, have created special-purpose programming languages specifically for encoding contractual rights and obligations.[39]

Note, however, that expressing a contract in a programming language does not solve all of the problems of contract law. Interpretation and enforcement remain real problems,[40] programing languages can be inferior to natural languages in capturing the nuances of parties' relationships,[41] and turning contracts into programs means that contracts are all but certain to have bugs, too.[42]

*Formal Language*, 14 EUR. REV. CONT. L. 307 (2018); Lauren Henry Scholz, *Algorithmic Contracts*, 20 STAN. TECH. L. REV. 128 (2017); Kevin Werbach & Nicolas Cornell, *Contracts Ex Machina*, 67 DUKE L.J. 313 (2017).

[38] *E.g.,* SOLIDITY, *supra* note 14.

[39] *See, e.g.,* LEGALESE, https://legalese.com; OPENLAW, https://www.openlaw.io; ACCORD PROJECT, https://docs.accordproject.org; *see also* Shaun Azzopardi, Gordon J. Pace & Fernando Schapachnik, *On Observing Contracts: Deontic Contracts Meet Smart Contracts*, *in* 2018 PROC. 31ST INT'L CONF. ON LEGAL KNOWLEDGE & INFO. SYSTEMS (JURIX 2018) 21 (linking smart contracts to deontic logic); Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu & Zengxiang Li, A Survey of Smart Contract Formal Specification and Verification (2020) (unpublished manuscript), https://arxiv.org/abs/2008.02712 (survey of methods for verifying smart contracts, including for adherence to a legal specification); Jan Ladleif & Mathias Weske, *A Unifying Model of Legal Smart Contracts*, *in* 2019 PROC. INT'L CONF. ON CONCEPTUAL MODELING 323 (comparison of smart-contract specification support for legal desiderata).

[40] *See* James Grimmelmann, *All Smart Contracts Are Ambiguous*, 2 J.L. & INNOVATION 1, 19-22 (2019).

[41] *See* Karen E.C. Levy, *Book-Smart, not Street-Smart: Blockchain-Based Smart Contracts and the Social Workings of Law*, 3 ENGAGING SCI. TECH. & SOC'Y 1, 4-10 (2017).

[42] *See* Shaanan Cohney, David Hoffman, Jeremy Sklaroff & David Wishnick, *Coin-Operated Capitalism*, 119 COLUM. L. REV. 591, 634-39 (2019).

## *B. Tax*

Tax law is also a good fit for programming languages, but for slightly different reasons. Here, the goal of formalization is typically to express existing legal rules in as much detail and with as little ambiguity as possible. The underlying rules, more so than in any other area of law, are already computational; a tax code is in large part simply a statement of computations to be applied to a taxpayer's activities. It is not entirely a coincidence that one of the earliest notable attempts at formalizing statutory law was Thorne McCarty's TAXMAN, which modeled the tax treatment of corporate reorganizations—although ironically, McCarty selected this particular topic because it was *unlike* other parts of tax law in being more open-ended and indeterminate.[43]

Several tax scholars have deal with linguistic themes in formalizing tax law. Sarah Lawsky proposes using formalized logical models to make drafters more attentive to problems of definitional scope,[44] and her work on modeling statutes using default logic is grounded in tax law.[45] In other work, she observes that existing tax *forms* are in effect a *sub silentio* formalization of the tax code; they give concrete algorithmic form to the legal requirements of the code.[46] In the other direction, several research groups are working on using natural-language techniques to parse the tax code and extract a formalized underlying structure.[47]

[43] *TAXMAN*, *supra* note 20; *see also* David M. Sherman, *A Prolog Model of the Income Tax Act of Canada*, *in* 1987 PROC. 1ST INT'L CONF. ON A.I. & L. 127; Kathryn E. Sanders, *CHIRON: Planning in an Open-Textured Domain*, 9 A.I. & L. 225 (2001).

[44] Sarah B. Lawsky, *Formalizing the Code*, 70 TAX L. REVIEEW 377 (2016).

[45] *A Logic for Statutes*, *supra* note 22; Sarah Lawsky, Nonmonotonic Logic and Rule-Based Legal Reasoning (2017) (unpublished manuscript), https://escholarship.org/uc/item/59j2j45w; Marcos A. Pertierra, Sarah Lawsky, Erik Hemberg & Una-May O'Reilly, *Towards Formalizing Statute Law as Default Logic through Automatic Semantic Parsing*, *in* 2017 PROC. SECOND WORKSHOP ON AUTOMATED SEMANTIC ANALYSIS INFO. LEGAL TEXT.

[46] Sarah Lawsky, *Form as Formalization*, 16 OHIO ST. TECH. L.J. 114 (2020); *see also* Richard J. Kovach, *Application of Computer-Assisted Analysis Techniques to Taxation*, 15 AKRON L. REV. 713 (1981).

[47] Nils Holzenberger, Andrew Blair-Stanek & Benjamin Van Durme, *A Dataset for Statutory Reasoning in Tax Law Entailment and Question An-*

Another use of formal programming-language methods in law is an ongoing overhaul of the tax computation software used by the French Public Finances Directorate (DGFiP).[48] It is common for governments to automate tax and other computations with software; whether the DGFiP's 125,000 lines of custom-written software correctly implements the 3,500-page French tax code is another and harder question. A research group working with the DGFiP are helping it transfer its software into a language with precise formal semantics so that parts of its algorithms can be proven correct and other parts subjected to better public auditing. In areas like tax where the computational parts of legal rules can be stated with high precision, formal methods drawn from programming-language theory are useful in reducing the gap between law on the books and law on the server.

## C. *Legal Drafting*

The process of expressing conveyances in Orlando's specific syntax exposes users (gently) to the discipline of programming. It invites them to think about what legal outcomes they are trying to achieve and then come up with specific expressions to generate those outcomes. In other words, using Littleton is a kind of legal drafting. Writing Orlando conveyances is a kind of programming that may help develop the same kinds of skills that are useful in legal drafting of all sorts. (Again, Littleton's ability to provide instant feedback may be particularly useful.)

The parallel between programming and drafting is of scholarly interest, too.[49] There is a large literature on the kinds of legal rules that can and cannot be made precisely computable, and on the consequences of doing so.[50] Some of this work engages with the tools that legal drafters use. For highly standard-

*swering*, *in* 2020 Proc. 2020 Nat. Legal Language Processing (NLLP) Workshop; Pertierra, Lawsky, Hemberg & O'Reilly, *supra* note 45.

[48] Denis Merigoux, Raphaël Monat & Jonathan Protzenko, A Modern Compiler for the French Tax Code (2020) (unpublished manuscript), https://arxiv.org/abs/2011.07966.

[49] *See* Houman B. Shadab, Software is Scholarship (2020) (unpublished manuscript), https://papers.ssrn.com/sol3/Papers.cfm?abstract_id=3632464; Ohm, Paul, *Computer Programming and The Law: A New Research Agenda*, 54 Vilanova L. Rev. 117 (2009); Grimmelmann, *supra* note 40.

[50] *See* Frank Pasquale & Glyn Cashwell, *Four Futures of Legal Automation*, 63 UCLA L. Rev. Discourse 26 (2015); Surden, *supra* note 30; William

ized instruments, such as wills, early expert systems and modern services like LegalZoom have worked to computerize the process of filling in an appropriate template from a legal formbook. Some such projects are one-offs: systems purpose-built to generate a particular kind of formulaic document, such as UCC financing statements.[51] Others are designed to work with multiple kinds of forms, which means that the templates themselves must be specified in a domain-specific language.[52]

More ambitiously, some projects hybridize the drafting process so that users are essentially drafting a natural-language legal text and an exact computational model of that text in parallel. Constraining the form of drafts in this way in essence compels the drafter to become a programmer of the specialized format being used. In some cases, such as the software used by legislative drafters to produce properly numbered and formatted statutes and track amendments, the constraints are relatively weak.[53] But other research efforts "force the attorney or paraprofessional to proceed in a highly organized fashion . . . so that the computer, and not the attorney or paraprofessional, keeps track of the complex linkages between the elements of the system as it evolves." This is not quite a programming language; rather it is a "controlled" or "normalized" language in which some elements, such as conjunctions and deontic expressions of obligation, have precisely defined meanings.[54] Some scholars in this tradition recognize and embrace the idea that

McGeveran, *Programmed Privacy Promises: P3P and Web Privacy Law*, 76 N.Y.U. L. Rev. 1812 (2001).

[51] William E. Boyd & Charles S Saxon, *The A-9: A Program for Drafting Security Agreements Under Article 9 of the Uniform Commercial Code*, 6 L. & Soc. Inquiry 639 (1981).

[52] *E.g.,* Charles S. Saxon, *Computer-aided Drafting of Legal Documents*, 7 L. & Soc. Inquiry 685 (1982). For a recent survey of efforts at legal specification see LSP Working Group, *Developing a Legal Specification Protocol* (2019), https://www-cdn.law.stanford.edu/wp-content/uploads/2019/03/LSPWhitePaperJan1119v021419.pdf.

[53] LegisPro, https://xcential.com; Open L. Draft, https://www.openlawlib.org/platform/open-law-draft/. *See generally* Timothy Arnold-Moore, Advanced tools for legislation (2019), https://ial-online.org/wp-content/uploads/2019/01/1.-Advanced-tools-for-legislationTimothy-Arnold-Moore.pdf (discussing functions performed by legislative drafting software).

[54] *See, e.g.,* Layman E. Allen & C. Rudy Engholm, *Normalized Legal Drafting and the Query Method*, 29 J. Legal Educ. 380 (1977); James A. Sprowl,

they are making the language of law more like a programming language, and are thoughtful about the language-design issues involved.[55]

Similarly, other authors describe the legal drafting of documents that are *semi-structured*.[56] These documents are not themselves programs; they consist mostly of natural language. But key terms are marked where they appear with specific tags indicating that they are being used consistently throughout, are referring to identified other sections, or other specified meanings designed to reduce ambiguity. This is similar to what legal research services already do when they hyperlink citations to the cited source and cross-reference defined terms in statutes to the sections where they are defined, except that it is carried out by the drafters themselves as a discipline for avoiding mistakes, making their meaning clearer to readers, and enabling limited automated analysis.[57]

Another interesting line of research draws on ideas from software engineering to improve legal drafting at a slightly higher level of abstraction. These scholars observe how programmers develop complex high-quality programs through specific design practices, such as packaging discrete units of functionality into self-contained "objects," and suggest ways that legal drafters could realize some of the same benefits by adopting similar practices.[58] Of particular interest here, these practices are typically supported by the programming languages that these developers work in. For example, an "object-oriented" language is precisely one that has built-in features for dividing functionality into discrete and self-contained objects.

*Automating The Legal Reasoning Process: A Computer that Uses Regulations and Statutes to Draft Legal Documents*, 4 L. & SOC. INQUIRY 1 (1979).

[55] *E.g.,* Sprowl, *supra* note 54; Thomas F. Blackwell, *Finally Adding Method to Madness: Applying Principles of Object-Oriented Analysis and Design to Legislative Drafting*, 3 NYU. J. LEGIS. & PUB. POL'Y 227 (2000).

[56] *A Logic for Statutes*, *supra* note 22; Matthew Roach, *Toward A New Language Of Legal Drafting*, 17 J. HIGH TECH. L. 43 (2016).

[57] Also worth of note is Lynn LoPucki's VisiLaw, in which statutory texts are marked up with standardized symbolic annotations to make their grammatical structure clearer. *See* VISILAW, https://www.visilaw.com.

[58] *See* Gerding, *supra* note 30.

Another such technique is "literate programming," in which a program is interwoven with its documentation.[59] A few authors have proposed literate programming specifically for implementing computational versions of statutes.[60] Again, the linguistic parallel between legal text and computer program is evident.

## *D. Visualization*

Generations of Property teachers have sketched diagrams of future interests for their students. A few of them have done so in a reasonably systematic way and published their diagrams. Most closely on point is Roger Anderson's catalog of geometric shapes for various future interests, such as a square for a life estate and a triangle with a dot in it for a reversion in fee simple.[61] Hopperton, for his part, illustrated his step-by-step analysis with a two-page summary chart of different estates.[62]

Further afield, Mark Reutlinger used diagrams to create timelines of events relevant to a RAP analysis,[63] William H. Lawrence used them to summarize commercial-paper transactions,[64] and William M. Richman used them to map the facts in conflict-of-laws cases.[65] And there is a tradition going back to

[59] *See* Donald Ervin Knuth, *Literate Programming*, 27 COMPUT. J. 97 (1984).

[60] *See also* Denis Merigoux & Liane Huttner, Catala: Moving Towards the Future of Legal Expert Systems (2020) (unpublished manuscript), https://hal.inria.fr/hal-02936606/document; Ohm, *supra* note 49 (discussing relevance of literate programming to law, in the form of a law-review article that is also a computer program).

[61] Roger W. Andersen, *Present and Future Interests: A Graphic Explanation*, 19 SEATTLE U. L. REV. 101 (1995).

[62] Robert J. Hopperton, *Teaching Present and Future Interests: A Methodology for Students that Unifies Estates in Land Concepts, Structures, and Principles*, 26 U. TOL. L. REV. 621 (1994) [hereinafter *Teaching Present and Future Interests*].

[63] Mark Reutlinger, *When Words Fail Me: Diagramming The Rule Against Perpetuities*, 59 MO. L. REV. 157 (1994).

[64] William H Lawrence,, *Diagramming Commercial Paper Transactions*, 52 OHIO ST. L.J. 267 (1991).

[65] William M Richman, *Diagramming Conflicts: A Graphic Understanding of Interest Analysis*, 43 OHIO ST. L.J. 317 (1982); William M. Richman, *Graphic Forms in Conflict of Laws*, 27 U. TOL. L. REV. 631 (1995); *cf.* Dung & Sartor, *supra* note 20 (formal logic for choice of law).

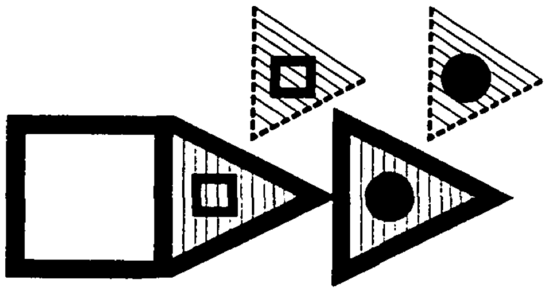


Figure 4: Anderson's future-interest diagrams

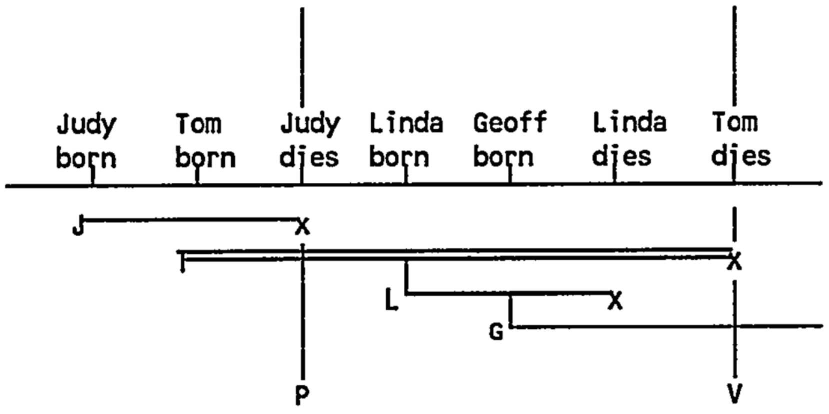
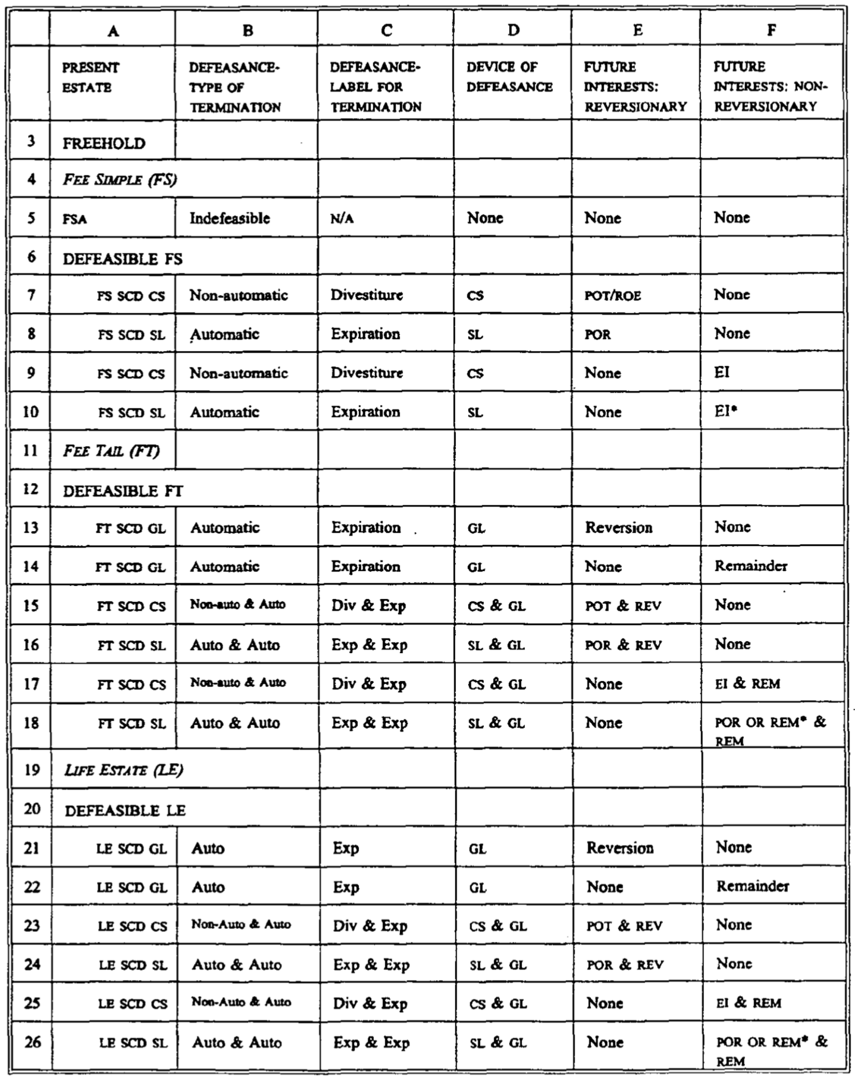


Figure 5: Reutlinger's RAP timelines

Wigmore of using diagrams to present the logical structure of legal arguments.[66]

What unites these legal diagrams is their attempt to be consistently rule-bound about what elements their diagrams include and how they are arranged. The synergy with computational law is obvious: in recent years scholars have developed systems to generate a variety of illuminating visualizations algorithmi-

[66] *See* John H. Wigmore, *The Problem of Proof*, 8 ILL. L.R. 77 (1913). *See generally* Chris Reed, Douglas Walton & Fabrizio Macagno, *Argument Diagramming in Logic, Law and Artificial Intelligence*, 2007 KNOWLEDGE ENGINEERING REV. 1 (tracing history).



| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| | PRESENT ESTATE | DEFEASANCE-TYPE OF TERMINATION | DEFEASANCE-LABEL FOR TERMINATION | DEVICE OF DEFEASANCE | FUTURE INTERESTS: REVERSIONARY | FUTURE INTERESTS: NON-REVERSIONARY |
| 3 | FREEHOLD | | | | | |
| 4 | *FEE SIMPLE (FS)* | | | | | |
| 5 | FSA | Indefeasible | N/A | None | None | None |
| 6 | DEFEASIBLE FS | | | | | |
| 7 | FS SCD CS | Non-automatic | Divestiture | CS | POT/ROE | None |
| 8 | FS SCD SL | Automatic | Expiration | SL | POR | None |
| 9 | FS SCD CS | Non-automatic | Divestiture | CS | None | EI |
| 10 | FS SCD SL | Automatic | Expiration | SL | None | EI* |
| 11 | *FEE TAIL (FT)* | | | | | |
| 12 | DEFEASIBLE FT | | | | | |
| 13 | FT SCD GL | Automatic | Expiration | GL | Reversion | None |
| 14 | FT SCD GL | Automatic | Expiration | GL | None | Remainder |
| 15 | FT SCD CS | Non-auto & Auto | Div & Exp | CS & GL | POT & REV | None |
| 16 | FT SCD SL | Auto & Auto | Exp & Exp | SL & GL | POR & REV | None |
| 17 | FT SCD CS | Non-auto & Auto | Div & Exp | CS & GL | None | EI & REM |
| 18 | FT SCD SL | Auto & Auto | Exp & Exp | SL & GL | None | POR OR REM* & REM |
| 19 | *LIFE ESTATE (LE)* | | | | | |
| 20 | DEFEASIBLE LE | | | | | |
| 21 | LE SCD GL | Auto | Exp | GL | Reversion | None |
| 22 | LE SCD GL | Auto | Exp | GL | None | Remainder |
| 23 | LE SCD CS | Non-Auto & Auto | Div & Exp | CS & GL | POT & REV | None |
| 24 | LE SCD SL | Auto & Auto | Exp & Exp | SL & GL | POR & REV | None |
| 25 | LE SCD CS | Non-Auto & Auto | Div & Exp | CS & GL | None | EI & REM |
| 26 | LE SCD SL | Auto & Auto | Exp & Exp | SL & GL | None | POR OR REM* & REM |

Figure 6: The first half of Hopperton's summary chart of future interests

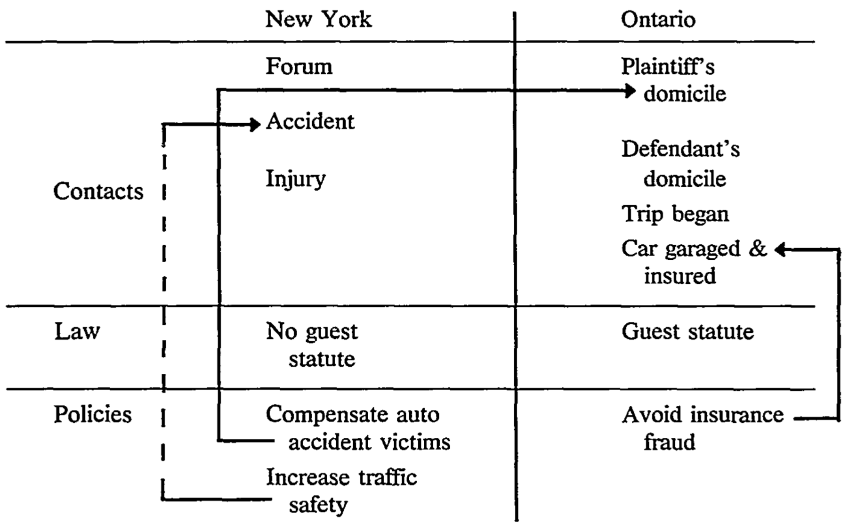


Figure 7: Richman's' conflict-of-laws diagrams

$$\begin{tikzcd}
\text{D' or} \arrow[r, "\substack{\text{``Pay Order P''} \\ \text{``D' or''}}"] & \text{P} \arrow[r, "\substack{\text{``Pay X''} \\ \text{``P''}}"] & \text{X} \arrow[r, "\substack{\text{``W/out rec.''} \\ \text{``X''}}"] & \text{Y} \arrow[r, "\text{No Indor.}"] & \text{Z} \arrow[r, "\substack{\text{``For} \\ \text{deposit only''} \\ \text{``Z''}}"] & \text{DB}
\end{tikzcd}$$

Figure 8: Lawrence's commercial-paper diagrams

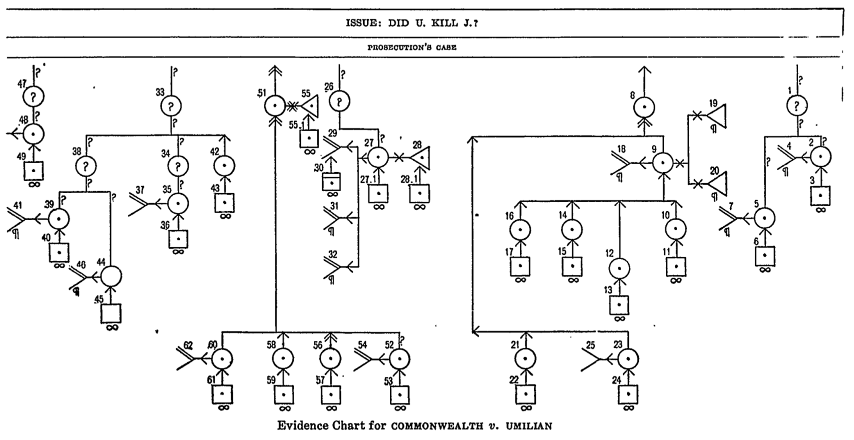


Figure 9: Wigmore's argument diagrams

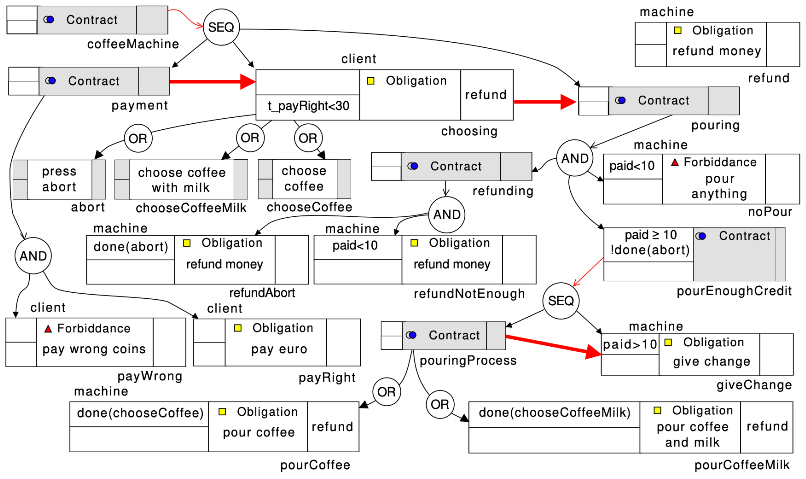


Figure 10: Camilleri, Paganelli, and Schneider's diagrams of formalized contracts

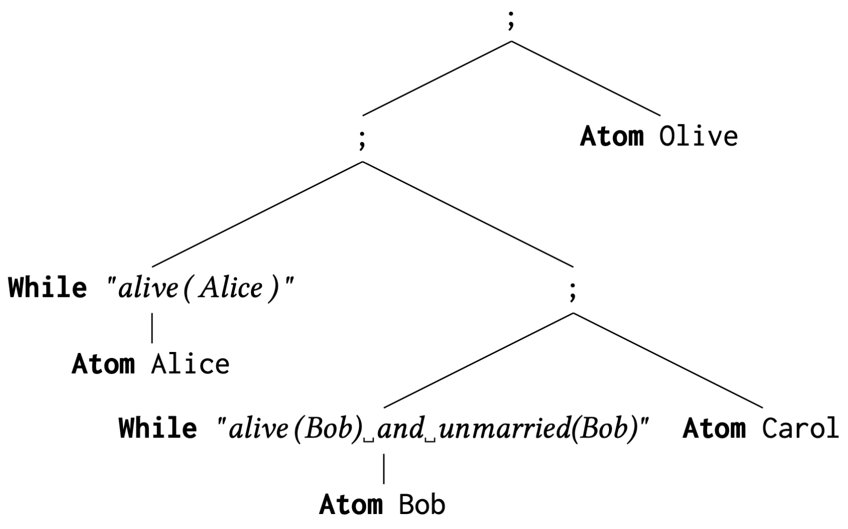


Figure 11: A previous version of Orlando visualizations

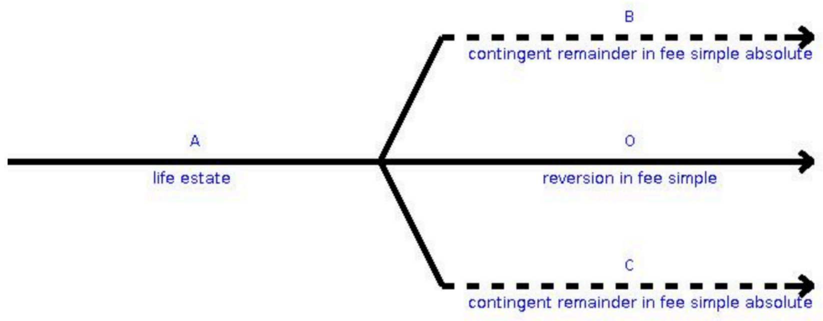

B
contingent remainder in fee simple absolute
A
life estate
O
reversion in fee simple
C
contingent remainder in fee simple absolute

Figure 12: Bayern's' automatically generated future-interest diagrams

cally.[67] The most ambitious work linking visualizing computable representations of legal relationships comes from contract law, where there have been various attempts to add visualizations to formalizations of contracts.[68]

Programming languages add another arrow to the quiver by describing a useful way to do visualization. As in Littleton, the human-readable "program" of a legal text, its digital representation as an abstract data structure, and its graphical visualization are three different views of the same object. Shawn Bayern's work on parsing conveyances does a version of this; it generates simple and elegant diagrams.[69]

Legal scholars looking for interesting ways to visualize legal law should consider writing a legal DSL for their domain of

[67] *See, e.g.,* Daniel Martin Katz & Michael James Bommarito, *Measuring the Complexity of the Law: The United States Code*, 22 A.I. & L. 337 (2014) (hierarchical diagrams of the United States Code); *The Supreme Court Mapping Project*, U. BALT., https://law.ubalt.edu/faculty/scotus-mapping/index.cfm (timelines of precedent in Supreme Court cases); Joseph Scott Miller, *Law's Semantic Self-Portrait: Discerning Doctrine with Co-Citation Networks and Keywords*, 81 U. PITT. L. REV. 1 (2019) (networks of related Supreme Court cases).

[68] *See, e.g.,* John J. Camilleri, Gabriele Paganelli & Gerardo Schneider, *A CNL for Contract-Oriented Diagrams*, 2014 CONTROLLED NAT. LANGUAGE 135.

[69] Shawn Bayern, CONVEYANCE INTERPRETER [hereinafter Conveyance Interpreter], https://essentially.net/property/.

interest, and then exploring different ways of interpreting programs in that DSL to generate diagrams. Figure 11 shows an earlier version of our title-tree diagrams, in which → nodes were explicit as nodes. By separating the translation code that generates title trees from the graphical code that transforms them into diagrams, we were able to experiment with different visualizations without having to rewrite any of Littleton's core syntactic and semantic logic.

## II. An Informal Overview

Only a few projects in formalizing law deal with property law,[70] and most of those focus on logical decomposition of the idea of rights in a thing, rather than with the specifics of future interests.[71] There are, however, a few notable exceptions. This Part surveys the prior work on formalizing property law, and then explains the approach taken in Orlando.

### A. *Previous Work*

In 1988, John Finan and Albert Leyerle described a program called Perp Rule, for testing future interests for compliance with the RAP.[72] Perp Rule asked users a series of yes/no questions such as, "IS THERE ANY POSSIBILITY THAT THE CLASS COULD INCREASE IN SIZE BY THE BIRTH

[70] John Zeleznikow, Andrew Stranieri & Mark Gawler, *Project Report: Split-Up–A Legal Expert System Which Determines Property Division upon Divorce*, 3 A.I. & L. 267 (1995) (division of property on divorce); Donald H. Berman & Carole D. Hafner, *Representing Teleological Structure in Case-Based Legal Reasoning: The Missing Link*, 1993 PROC. 4TH INT'L CONF. ON A.I. & L. 50 (first possession); Trevor Bench-Capon, *Arguing with Dimensions in Legal Cases*, 2017 18TH WORKSHOP ON COMPUTATIONAL MODELS NAT. ARGUMENT 2 (same); Katie Atkinson, *Introduction to Special Issue on Modelling Popov v. Hayashi*, 20 A.I. & L. 1 (2012) (same); Sanders, *supra* note 43 (types of property transactions).

[71] L. Thorne McCarty, *Ownership: A Case Study in the Representation of Legal Concepts*, 10 A.I. & L. 135 (2002); *see also* Layman E. Allen,, *Formalizing Hohfeldian Analysis to Clarify the Multiple Senses to Legal Right: A Powerful Lens for the Electronic Age*, 48 S. CAL. L. REV. 428 (1974) (initial entry in decades-long project to formalize Hohfeldian relationships).

[72] John P. Finan & Albert H. Leyerle, *The Perp Rule Program: Computerizing the Rule Against Perpetuities*, 28 JURIMETRICS J. 317 (1988).

OF A NEW MEMBER?" and, "ARE ANY ONE OR MORE MEMBERS OF THE CLASS ENTITLED TO IMMEDIATE DISTRIBUTION OF THE PRINCIPAL?"[73] As these examples, show, Perp Rule dealt with different aspects of the RAP than Littleton currently does. More importantly, Perp Rule was incapable of answering these questions for itself; it had to *ask* the user to do the necessary analysis at each step. In essence, it was an elementary expert system for walking the user through a decision tree that models the RAP. [74]

In 1989, David Becker also attacked the RAP with an astonishingly detailed step-by-step procedure for analyzing compliance.[75] His article runs to an astonishing 187 pages and nearly 100,000 words. Becker's "methodology" shows how formalization can exert a disciplining effect. It forthrightly confronts many of the details and special cases that a more casual treatment can sweep under the rug. But unlike Finan and Leyerle, Becker made no attempt to actually implement it as a program. Indeed, the article does not even contemplate that computerization might be possible or desirable. As a result, the procedure is riddled with "exceptions," "observations," and "adjustments." Robert Hopperton, in articles published in 1994 and 1999, also attempted to impose greater logical structure on the teaching of future interests and the RAP.[76] And a few professors have created interactive study aids, some of which can generate problems for students to try.[77]

[73] *Id.* at 328-29.

[74] *Id.* at 325. The full version of Littleton implements a significantly more sophisticated RAP algorithm. But that is a tale for another time.

[75] David M. Becker, *A Methodology for Solving Perpetuites Problems under the Common Law Rule: A Step-by-Step Process that Carefully Identifies All Testing Lives in Being*, 67 WASH. U. L.Q. 949 (1989).

[76] *Teaching Present and Future Interests*, *supra* note 62; Robert J. Hopperton, *Teaching the Rule against Perpetuities in First Year Property*, 31 U. TOL. L. REV. 55 (1999).

[77] *Lawsky Practice Problems*, LAWSKYPRACTICEPROBLEMS.ORG, https://www.lawskypracticeproblems.org; Peter B. Maggs & Thomas D. Morgan, *Computer-Based Legal Education at the University of Illinois: A Report of Two Years' Experience*, 27 J. LEGAL EDUC. 138 (1975); John A. Humbach, EST. SYS. & BASIC FUTURE INTERESTS (2010), http://webpage.pace.edu/jhumbach/BES00page-Gateway.htm; Ned Snow, FUTURE INTERESTS MADE SIMPLE (2015), https://apps.apple.com/us/app/future-interests-made-simple/id933368390; *see also* FUTURE INTERESTS APPLICATION (defunct), https://

In 2010, Shawn Bayern wrote a conveyance interpreter in the Java programming language.[78] Like Littleton, Bayern's interpreter parses a conveyance written in English, displays a diagram of the resulting interests, and is capable of naming most of the standard interests taught in First-year property. Bayern's was the first formal treatment to truly capture the recursive linguistic structure of standard conveyances; his interpreter can parse conveyances containing an arbitrary number of clauses. In addition, its linguistic analysis of granting clauses is quite insightful; it clearly distinguishes conditions precedent, durations, and limitations.

Orlando and Littleton build on Bayern's work by linking an interpreter to a language with precisely specified syntax and semantics. That clean and well-theorized core enables it to (1) interpret a wider range of constructions, (2) handle more complicated interrelationships among conditions, (3) update the state of title in response to events and subsequent conveyances, (4) reason formally about future events (and thus about vesting and the Rule Against Perpetuities), (5) and clarify important property concepts. In short, Orlando and Littleton provide a firm theoretical foundation for systematic research in a new field for which Bayern developed the initial proof of concept.

## *B. Orlando and Littleton*

More precisely, Orlando deals with ***conveyances*** like "O conveys to A and her heirs." The actual language used by lawyers past and present is more complicated, but when explaining the system, it is customary to write stylized conveyances like this one. For simplicity, we will omit the "O conveys" part when it is clear from context. In addition, we will write the conveyances in a distinctive fixed-width typeface—e.g., `to A and his heirs`—

web.archive.org/web/20180820192604/http://www.futureinterestsapp.com/; RULE AGAINST PERPETUITIES APPLICATION (defunct), https://web.archive.org/web/20180827233129/http://www.rapapp.info/.

[78] Conveyance Interpreter, *supra* note 69. Bayern documented it in a short conference article. Shawn J. Bayern, *A Formal System for Analyzing Conveyances of Property Under the Common Law*, 23 JURIX 139 (2010). We are also grateful to Bayern for making available the source code to his interpreter, from which we have learned much, even though we ultimately made very different design decisions.

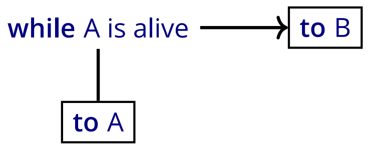


Figure 13: `to A for life, then to B and her heirs`

when they follow the rules of Orlando's formal syntax, rather than than the looser standards of legal English. Orlando specifies, and Littleton carries out, a translation of a conveyance into a data structure called a *title tree*. As an initial example, Figure 13 shows the title tree for `to A for life, then to B and her heirs`.

This picture contains ***nodes*** to represent A and B's interests and the relationship between them:

- Two are "**to**" nodes that represent their ownership interests. Each of them consists of the keyword **to** plus the name of the person who owns the interest: A and B, respectively.[79]
- One is a"**while**" node that describes the circumstances under which another interest terminates. A **while** has a ***condition*** (here "A is alive") and a vertical line downwards to whatever interest should be terminated when that condition becomes false (here, A's).
- Finally, the horizontal $\rightarrow$ arrow from the **while** to B's interest shows what order their interests come in. Note that the arrow starts at the **while** node, not at the **to** node beneath it.[80]

Initially, while the condition "A is alive" is true, A's interest is possessory. But when the condition becomes false, the **while** terminates A's interest and B's interest becomes possessory. Thus, this picture as a whole shows a life estate followed by a remainder. The **while** and the **to** A beneath it are a unit—a ***subtree***—that as a whole represents A's life estate. The **to** B represents B's remainder.

Note that nothing in the tree is labeled a "life estate" or a "remainder." Indeed, the entire concept of "life estate" takes more than one node to represent, and the only reason we can recognize **to** B as a "remainder" is because of where it appears

[79] These **to** nodes are displayed in a box to visually distinguish them and emphasize that they corresponds to ownership interests.
[80] Formally, this kind of arrow is another type of node. See *infra* III.

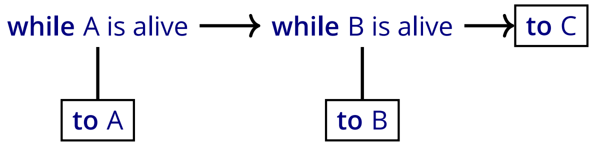


Figure 14: `to A for life, then to B for life, then to C`

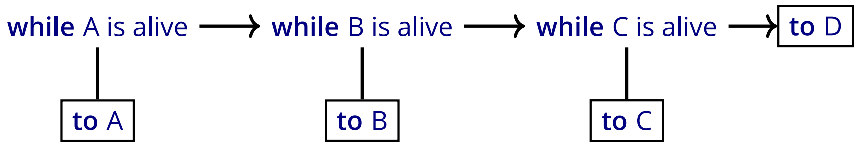


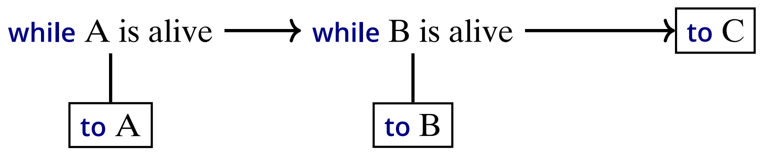
Figure 15: `To A for life, then to B for life, then to C for life, then to D`

in the tree as a whole. This is fundamental to the design of Orlando. Rather than have separate types of nodes for every distinctly named type of interest, it uses a small set of node types to model the behavior of interests: who is entitled to possession under what circumstances.

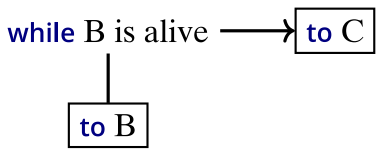
In Orlando, both the language of conveyances and the structure of title trees are ***recursive***. They are built up from smaller parts. The conveyance `to A for life, then to B for life, then to C and his heirs` adds an additional granting clause; it creates two life estates, rather than one. This additional granting clause becomes an additional "life estate" in the resulting title tree: additional **while** and **to** node, as shown in Figure 14. The process can be extended indefinitely. Figure 15 shows three successive life estates and a remainder. For four, five, six, or more, all one needs is a big enough piece of paper.

Time, and possession, flow from left to right in a title tree. In Figure 14, possession will start at A's interest at the left, then move to B's interest in the center, and ultimately to C's interest at the right. Imagine putting your finger on the currently possessory interest and moving it forward as time passes and the state of title changes. Your finger will move only forward to the right, never backwards to the left against the direction of an arrow. This means that any interests to the left of your finger are *irrelevant* to the state of title; they can never become possessory.

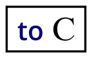
Thus, Littleton discards all nodes in the "past," to the left of the currently possessory interest. When a node ***terminates***, it is removed from the title tree. A **while** node that is in the



(a) `to A for life, then to B for life, then to C`



(b) `A dies`



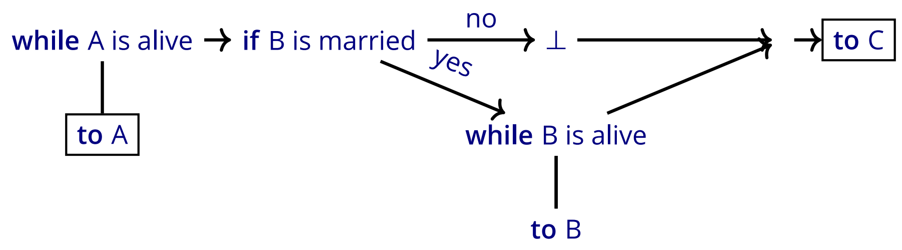
(c) `B dies`

Figure 16: Updating a title tree

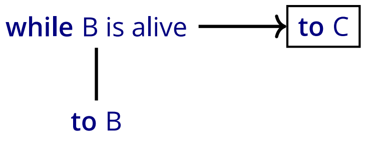
present—at the left of the title tree–terminates when its condition becomes false, along with any nodes beneath it. Figure 16 shows what happens to the title tree in Figure 14 at A's death. The condition "A is alive" in the left **while** node becomes false, so that node—and A's interest beneath it—disappears. Possession passes to the right, ending up with the newly leftmost interest: B's life estate. If B then dies, the same thing happens again and C takes possession. In an Orlando title tree, a **while** node is not just a static description of the duration of an interest. It also responds dynamically to events.

There are two more types of node to introduce. One is needed for conditions precedent, such as in the conveyance `to A for life, then if B is married to B for life, then to C`. The other does some useful bookkeeping and plays an important role in the formal version of title trees. Both are illustrated in Figure 17. The **while** and **to** at the left are familiar, as is the **to** at the right. The new parts are in the middle.

- The **if** represents the branching possibilities at A's death. If the condition B is married is true when possession reaches the **if** node, then possession passes immediately to the node on the "yes' branch: the **while** in B's life estate. If the condition is false, possession instead follows the "no" arrow, to . . .



(a) `to A for life, then if B is married to B for life, then to C`



(b) B is married when A dies

(c) B is unmarried when A dies

Figure 17: An **if** node

- The ⊥ (pronounced "bottom") on the "no" branch is a symbol used in computer science theory to denote an absence, literally nothing.[81] In Orlando, it represents a term that has terminated, or one that was never there to begin with. Here, it means that if B is unmarried at A's death, possession should bypass B's interest and go immediately to C's.

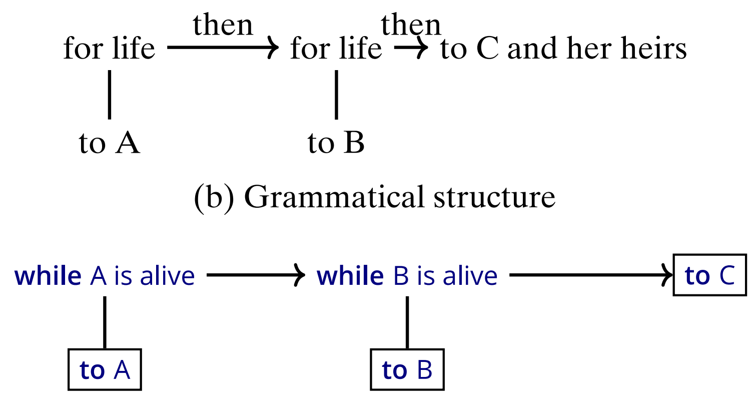
Again, we follow the convention that nodes in the "past" are eliminated, which means that an **if** immediately disappears as soon as it is reached. Its only job is to pass possession forward along one branch or the other. The title trees corresponding to these two cases are illustrated in the rest of Figure 17.

The English statement "the language `to X and his heirs` creates a fee simple in X" is an informal description of the relationship between the language of a conveyance and the resulting interests. Figure 18 shows the correspondence informally.

[81] *See* Dana Scott & Christopher Strachey, Toward a Mathematical Semantics for Computer Languages 23 (1971) (unpublished manuscript) (Oxford University Computing Laboratory Technical Monograph PRG-6), https://home.cs.colorado.edu/~bec/courses/csci5535/reading/PRG06.pdf ("The new element *bot* can be regarded as an 'embodiment' of the undefined.").

```
to A for life, then to B for life, then to C and her heirs
```

(a) Conveyance



(b) Grammatical structure

(c) Title tree

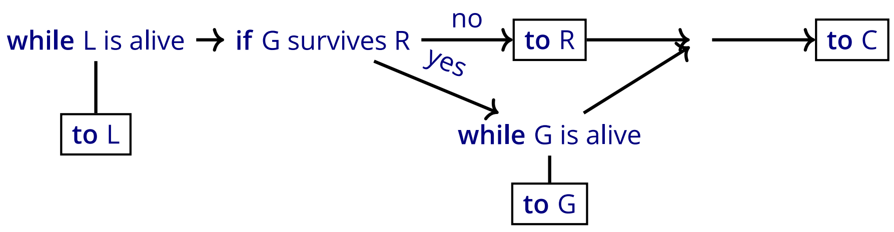
Figure 18: Translation of a conveyance

Each **to** node corresponds to an occurrence of `to X` in the conveyance; each **while** node corresponds to an occurrence of `for life`; each horizontal line to an occurrence of `then`. The essence of the programming-language approach to estates and future interests is to formalize this mapping.

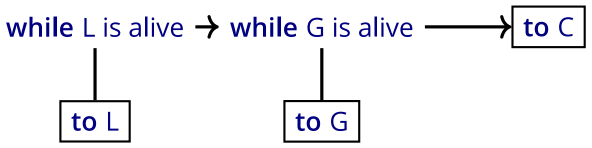
## C. *An Example*

Orlando and Littleton can handle some quite intricate conveyances. For example, consider the conveyance `To Lear for life, then if Goneril survives Regan to Goneril for life, otherwise to Regan, then to Cordelia and her heirs`, which creates four interests. Lear has a possessory life estate, then Regan and Goneril have mutually exclusive remainders, and finally Cordelia has a remainder.

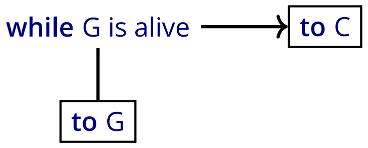
Orlando and Littleton represent this state of title using a title tree with four nodes corresponding to interests. This title tree are is depicted in the top diagrams of Figures 19 (Orlando) and 20 (Littleton). The Orlando diagram is abstract; it is a formal, mathematical representation of the present and future interests in this property. The Littleton diagram is concrete; it is the output of our actual computer program to help users visualize future interests. The Orlando representation is skeletal; it



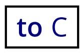
(a) `To Lear for life, then if Goneril survives Regan to Goneril for life, otherwise to Regan, then to Cordelia and her heirs.`



(b) `Regan dies.`



(c) `Lear dies.`

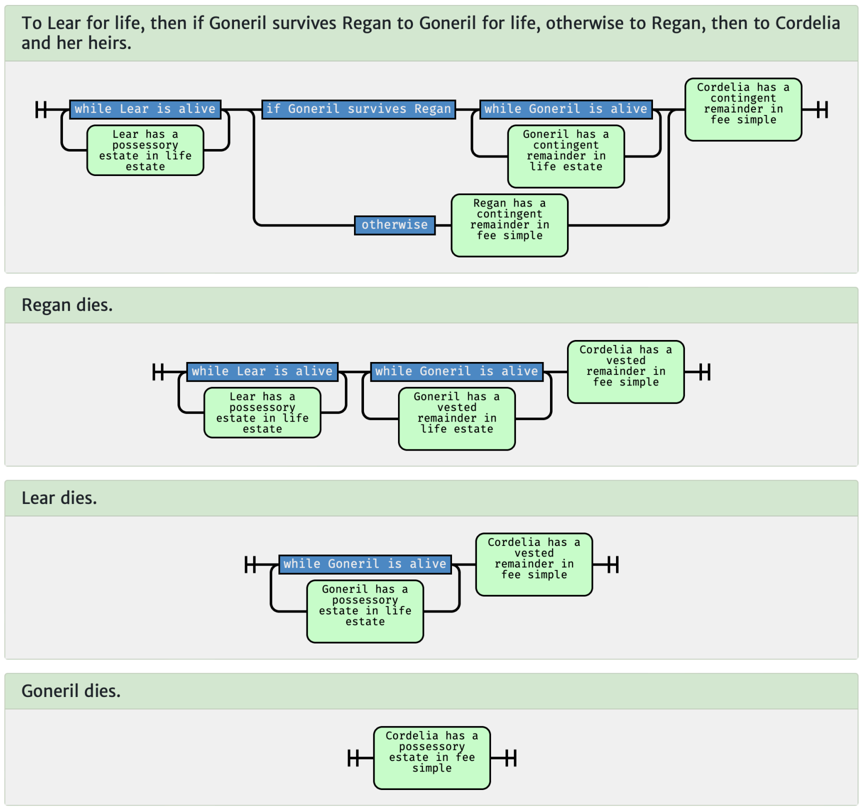


(d) `Goneril dies.`

Figure 19: A more complicated example of successive and alternative interests (Orlando)

contains the bare minimum needed to describe the state of title. The Littleton visualization is more verbose; it includes such details as that Goneril's remainder is in life estate whereas Regan's is in fee simple, and that all three remainders are contingent.

A professor teaching this example might test their students' understanding by asking, "What happens if Regan dies?" As the second diagrams in Figures 19 and 20 show, this event has three important effects. First, Regan's interest needs to be struck because its condition precedent (that Goneril not survive Regan) has definitively failed. Second, although Goneril's remainder is still a future interest, it becomes vested rather than contingent because its condition precedent (that Goneril survive Re-


To Lear for life, then if Goneril survives Regan to Goneril for life, otherwise to Regan, then to Cordelia and her heirs.

while Lear is alive
Lear has a possessory estate in life estate
if Goneril survives Regan
while Goneril is alive
Goneril has a contingent remainder in life estate
Cordelia has a contingent remainder in fee simple
otherwise
Regan has a contingent remainder in fee simple

Regan dies.

while Lear is alive
Lear has a possessory estate in life estate
while Goneril is alive
Goneril has a vested remainder in life estate
Cordelia has a vested remainder in fee simple

Lear dies.

while Goneril is alive
Goneril has a possessory estate in life estate
Cordelia has a vested remainder in fee simple

Goneril dies.

Cordelia has a possessory estate in fee simple

Figure 20: A more complicated example of successive and alternative interests (Littleton)

gan) has been satisfied. Third, Cordelia's remainder is now also vested because the contingency that could prevent it from becoming possessory (Regan surviving Goneril) is now impossible.

At this point, the consequences of further events are readily predictable. At Lear's death, Goneril's remainder becomes possessory () and at Goneril's death, Cordelia's remainder becomes possessory. As the third and fourth diagrams in Figures 19 and 20 show, Orlando and Littleton capture every one of these changes. Orlando's formalism provide the necessary framework for Littleton to correctly describe these changes.

## III. The Formal Details

The informal description of Orlando in II had four moving parts: (1) a ***conveyance*** like `to A for life, then to B` is (2) ***translated*** into (3) a ***title tree***, which is (4) ***updated*** in response to events. They correspond to four fundamental theoretical tools in the design and implementation of programming languages:

- A ***grammar*** that defines the syntax of a "program." In Orlando, this grammar resembles a subset of English, but with tightly restricted, formally specified syntax.[82]
- An abstract ***data type*** that models the current state of a program as it executes. In Orlando, this data type is a title tree, which is specified by another grammar that describes the contents of title trees.
- A ***translation*** function that specifies how to turn a conveyance into a title tree that describes the interests that conveyance creates. In Orlando, this translation function consists of a set of rules, each of which translates a small portion of a conveyance into a corresponding portion of a title tree.
- An ***operational semantics*** that specifies how a program is executed, step by step.[83] In Orlando, the operational semantics

[82] Orlando uses a ***context-free grammar***, in which there are no long-distance interdependencies between different parts of an expression. *See generally* MICHAEL SIPSER, INTRODUCTION TO THE THEORY OF COMPUTATION (3d ed. 2012) (describing context-free grammars).

[83] *See generally* GLYNN WINSKEL, THE FORMAL SEMANTICS OF PROGRAMMING LANGUAGES: AN INTRODUCTION (1993); HANS HÜTTEL, TRANSITIONS AND

consists of update rules that describe how a title tree changes in response to events.

This Section presents these four ideas, in detail but not quite in this order.

The formal presentation of Orlando follows some standard conventions from computer science. First, certain ***keywords***—like `to` and `heirs`—have specifically defined roles in Orlando.[84] For example, `heirs` is part of the standard phrase `and her heirs` used to create a fee simple. It is common for programming languages to have a few dozen reserved keywords—"reserved" in the same sense that a statutory section number is reserved and should not be used for something else. You can name your bardcore band "Adam And His Heirs" if you want, but using `Adam and his heirs` as the name of a person in an Orlando program is a good way to confuse Littleton and yourself. Littleton recognizes and responds to specific keywords, but it has no deeper understanding of their connotations. It responds to `and her heirs` and `and the heirs of his body`, but not `the heir of all the ages`.

Next, the equations that define Orlando will frequently use ***variables*** like *person* and $t_1$. Each variable has a specific ***type***, e.g. *person* always refers to a person's name, $t$ always refers to a title tree, and so on. So whenever the variable *person* appears in a definition it can be filled in with the name of an arbitrary person. The names of variables that represent expressions in Orlando, like a complete *conveyance* or a *limitation* on a grant, will be the name of the kind of thing they represent. The names of variables that represent parts of title trees will be individual letters, like $c$ for "condition". By convention, having introduced a variable like $c$, we can also put subscripts on it. Thus $c_1$, $c_2$, $c_3$, and so on all refer to conditions—possibly different conditions, possibly the same one.

---

TREES: AN INTRODUCTION TO STRUCTURAL OPERATIONAL SEMANTICS (2010). In Property Conveyances, *supra* note 7, we also presented a denotational semantics for Orlando and proved that its operational and denotational semantics are equivalent. The denotational formulation is more convenient for proving certain types of propositions about the behavior of programs. *See generally* WINSKEL, *supra*; DAVID A. SCHMIDT, DENOTATIONAL SEMANTICS: A METHODOLOGY FOR LANGUAGE DEVELOPMENT (1986).

[84] Keywords are written in the `monospaced typeface` used for the literal language of a conveyance.

The full set of rules that define the core subset of Orlando are collected in the Appendix.

## A. *Title Trees*

Title trees and conveyances in Orlando are terms in ***formal languages***. Unlike a natural language, which is whatever people speak to each other, the syntax and semantics of a formal language are precisely specified by a ***grammar***. Like the grammar for a natural language, it spells out how the pieces of one fit together, how larger units are made up of smaller ones, and what counts as a valid expression. The difference is that while natural-language grammar is flexible and context-dependent, a formal-language grammar is rigid and rigorous. These rules are constitutive; any expression allowed by them is meaningful in Orlando, while all other expressions are officially meaningless. The rules are symbolic; they describe Orlando's syntax using mathematical and logical notation. And the rules are recursive; they show how to build up more complicated expressions from simpler ones.

II described informally five types of title tree nodes. More formally, they are defined by a grammar with five ***rules***:

$$\begin{array}{lcl} t & \Rightarrow & \textsf{to}\ p \\ t & \Rightarrow & \bot \\ t & \Rightarrow & t_1\ \textsf{while}\ c \\ t & \Rightarrow & \textsf{if}\ c\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \\ t & \Rightarrow & t_1 \rightarrow t_2 \end{array}$$

Read the symbol ⇒ as "can consist of." Each line describes a different one of the title tree node types. Thus, a title tree can consist of

1. the keyword to and a person $p$ (an interest),
2. the symbol $\bot$ by itself (nothing),
3. the symbol while linking a smaller title tree $t_1$ and a condition $c$ (a temporal limit),
4. the symbols if, then, and else linking a condition $c$ and two smaller title trees $t_1$ and $t_2$ (a choice between two mutually exclusive possibilities), or
5. the symbol $\rightarrow$ linking two smaller title trees $t_1$ and $t_2$ (a sequential division of ownership across time).

There are two important types here. Title trees themselves are denoted with the letter $t$ (and the usual optional subscript).[85] Conditions, written with the letter $c$, are statements about the world being modeled that can be true or false. A logician would say that conditions are predicates; a mathematician would say that they are Boolean-valued functions. So far, we have seen two: "$p$ is alive" and "$p$ is married".[86]

These rules are obviously recursive. The third rule, for example, says that a if a title tree $t$ consists of a **while** node, then it contains *another* title tree $t_1$. That title tree, in turn, must be one of the five types. If it is a **to** or a $\perp$, then the recursion ends, because neither of these rules has another title tree on the right hand side. But if it is another **while**, the recursion continues: this new title tree must be expanded until every subtree has bottomed out with a **to** or a $\perp$. For example, here is the ***derivation*** of the title tree from Figure 13, which depicts a life estate followed by a remainder:

$$\begin{aligned} t &\Rightarrow t_1 \rightarrow t_2 \\ &\Rightarrow t_1 \rightarrow \textbf{to}\ p \\ &\Rightarrow t_1 \rightarrow \textbf{to}\ \mathsf{B} \\ &\Rightarrow (t_3\ \textbf{while}\ \mathsf{A\ is\ alive}) \rightarrow \textbf{to}\ \mathsf{B} \\ &\Rightarrow (\textbf{to}\ p\ \textbf{while}\ \mathsf{A\ is\ alive}) \rightarrow \textbf{to}\ \mathsf{B} \\ &\Rightarrow (\textbf{to}\ \mathsf{A}\ \textbf{while}\ \mathsf{A\ is\ alive}) \rightarrow \textbf{to}\ \mathsf{B} \end{aligned}$$

These symbolic description of title trees may not look much like the diagrams from II. But there is a straightforward, one-to-one correspondence between the textual and graphical depictions of title trees. Figure 21 shows the corresponding fragment of a title-tree diagram for each of the title-tree rules. To form a complete title tree, just glue together the appropriate pieces.

The visual description and the symbolic one are two different flavors of concrete syntax to describe the same abstract object, just like a graph and the equation $y = 3x + 4$ are two different descriptions of the same line. The advantage of the visual

[85] Title trees are an example of an *algebraic data type*. *See generally* BENJAMIN C. PIERCE, TYPES AND PROGRAMMING LANGUAGES (2002) (thorough presentation of type theory).

[86] Conditions are written in an ordinary serif typeface, to distinguish them from the words describing a condition in an Orlando program, which are written in a `monospaced terminal font`.

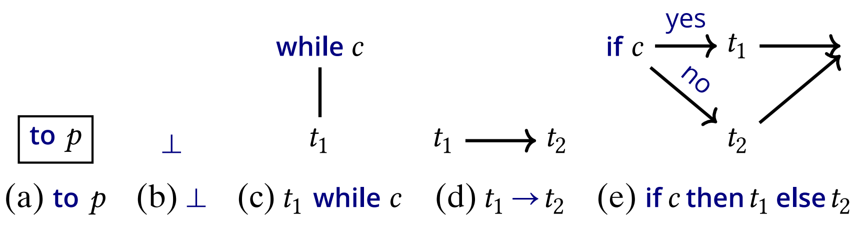


Figure 21: Graphical representation of title trees

version is that it is easier for people to grasp. The advantage of the symbolic version is that it is easier to specify precisely how to carry out formal mathematical operations on. To be clear, the diagrams are not just heuristic sketches; they are well-defined representations of well-defined mathematical objects, and can be freely converted to and from the symbolic representation. Littleton uses the symbolic version internally and displays diagrams to users. This Article will continue to use both.

A title tree represents the state of title at a single moment in time. It incorporates all of the information needed to keep track of who owns what, and who will own what in the future. If you have a title tree, you can discard the conveyance that generated it; the title tree captures everything you need to know. As events occur, the title tree can be updated to keep track of how they affect the state of title—at which point the new title tree will capture everything relevant and the old one can be discarded.

## *B. Semantics*

So far, a title tree is just an abstract data structure. We have been saying informally that the leftmost to node represents a possessory interest, and that other to nodes represent interests that can become possessory in response to events. Just as we formalized the intuitive description of the structure of title trees, we can also formalize the intuitive description of how they behave in response to events. This provides an *operational semantics* for the formal language of title trees.[87]

[87] The particular style of operational semantics we use is a ***derivative semantics***. *See* Janusz A. Brzozowski, *Derivatives of Regular Expressions*, 11 J. Ass'n for Computing Machinery 481 (1964) (defining derivative semantics). For a fuller treatment of the formalisms as used in Orlando see Property Con-

Orlando models the changes to a title tree over time with the concept of ***events***: discrete occurrences such as `A dies`, `5 years pass`, and `Mars becomes a state`. A sequence of zero or more events is a ***history***, e.g.:

```
B dies.
Mars becomes a state.
The property is used as a school.
```

The actual mechanics of appropriately modifying a title tree in response to events are handled by an update function $\delta()$ (so named because it computes the "delta" or change in the state of the title). To update a title tree $t$ in response to an event, replace it with $\delta(t)$.[88]

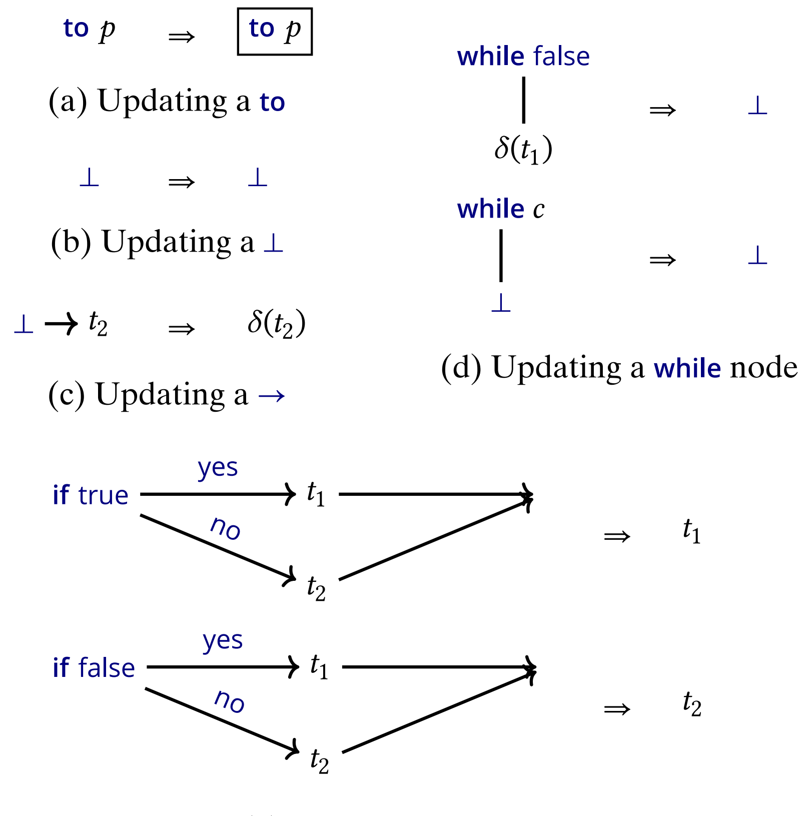
The basic idea of $\delta()$ is that the special value $\perp$ represents a portion of a title tree that has completely terminated. All of the interests it describes have ended. All it can do is pass possession onwards to the next interest ready to receive it. $\delta()$ identifies interests that have terminated, replaces them with $\perp$, and then deletes the $\perp$s from the title tree, pushing possession forward to the interests that follow them. **while** and **if** nodes do the terminating; $\rightarrow$ arrows do the pushing forward. (**to** nodes never terminate on their own; they can only be terminated by nodes above them in the tree.)

In particular, $\delta()$ is computed—and only computed—on the leftmost branch of a title tree. That is the branch leading to the currently possessory interest, and that is where the state of title could actually change.

The actual value of $\delta()$ is an example of ***definition by cases***. Since a title tree $t$ could take one of five forms, $\delta(t)$ must be defined for each of those forms. So there is a line in the definition corresponding to each of the rules in the grammar for title trees.

veyances, *supra* note 7. For general introductions to the theory of languages and automata, on which the derivative semantics draws, see SIPSER, *supra* note 82; DEXTER C. KOZEN, AUTOMATA AND COMPUTABILITY (1997); HARRY R. LEWIS & CHRISTOS H. PAPADIMITRIOU, ELEMENTS OF THE THEORY OF COMPUTATION (2d ed. 1997). A few papers have applied operational semantics to legal topics. *See* da Rocha Costa, *supra* note 25; Azzopardi, Pace, Schapachnik & Schneider, *supra* note 32.

[88] The event itself is not formally a parameter of $\delta()$. That is because $\delta()$ must be interleaved with a function that updates the state of conditions in a title tree. See Property Conveyances, *supra* note 7 for the details.



(a) Updating a **to**

(b) Updating a ⊥

(c) Updating a →

(d) Updating a **while** node

(e) Updating an **if** node

Figure 22: Update rules

The definitions here are the heart of Orlando, so it is worth going through them carefully.

$$\delta(\textbf{to } p) \;=\; \textbf{to } p$$

A **to** by itself represents a fee simple, which cannot be affected by events because it is always possessory. Other nodes can limit a **to** and cause it to terminate, but the **to** itself is unaffected. Thus $\delta()$ leaves it unchanged. Figure 22a illustrates.

$$\delta(\bot) \;=\; \bot$$

A ⊥ represents the opposite of a fee simple: a node that cannot be affected by events because it is *never* possessory. Other nodes cannot revive it. Thus it too is unchanged by $\delta()$. Figure 22b illustrates.

There are two possibilities for $t_1 \rightarrow t_2$, which are illustrated in Figure 22c.

$$\text{if} \quad \delta(t_1) = \bot \quad \text{then} \quad \delta(t_1 \rightarrow t_2) = \delta(t_2)$$

On the one hand, it might be the case that the first half of the $\rightarrow$, has terminated, i.e., that $\delta(t_1) = \bot$. If so, then possession should pass to the second half. In this case, the $\rightarrow$ itself is removed, leaving only the right-hand subtree $t_2$, which itself will now need to be updated to $\delta(t_2)$.

$$\text{if} \quad \delta(t_1) \neq \bot \quad \text{then} \quad \delta(t_1 \rightarrow t_2) = \delta(t_1) \rightarrow t_2$$

On the other hand, if the left subtree still exists, the $\rightarrow$ remains. In this case, the left subtree $t_1$ should be updated, but the right subtree $t_2$ should not. Only currently possessory interests need to be checked for termination by $\delta()$, and the right subtree has no such interests (they are all in the future).

The subcases for **while** and **if** are similar. A **while** terminates (i.e. is replaced with $\bot$) either if its associated condition has become false, or if the subtree beneath it has terminated. Otherwise, the subtree updates in place and the **while** remains. A **if** node is always removed when it is evaluated with $\delta()$. If the condition is true, the **if** is replaced with its first subtree (the "yes" branch), but if the condition is false, the **if** is replaced with its second subtree (the "no" branch).

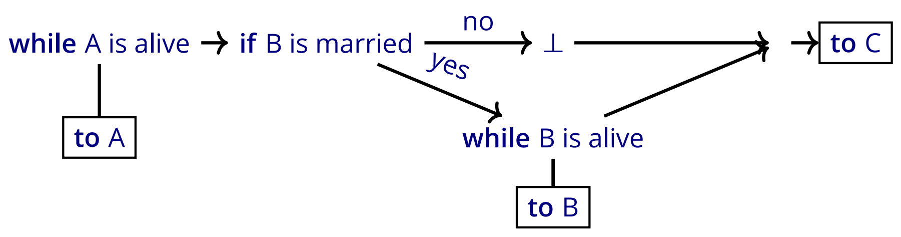
These definitions capture formally the idea of interests terminating in response to events and possession passing to subsequent interests. They may seem abstract, but that is what makes them so convenient to compute with. The formal definition of $\delta()$ can be applied mechanically in a way that natural-linguistic descriptions of the rules cannot.

A detailed example will show how the rules work together. Consider the following sequence of events:

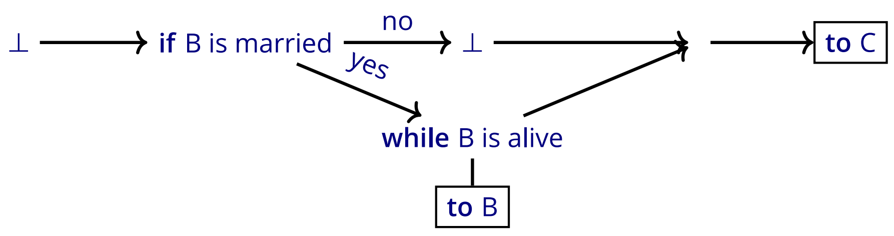
```
O conveys to A for life, then if B is
  married to B for life, then to C.
B marries.
A dies.
B dies.
```

First, consider this symbolically. The conveyance in the first line translates into the following title tree:

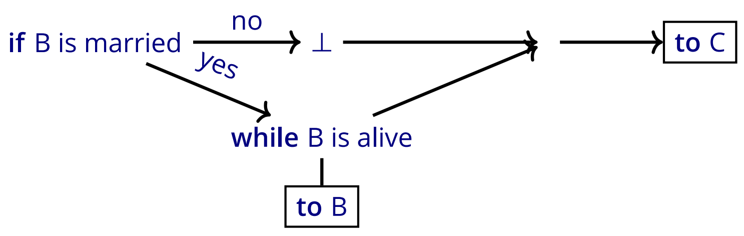
(**to** A **while** A is alive) $\rightarrow$
(**if** B is married **then** (**to** B **while** B is alive) **else** $\bot$) $\rightarrow$
**to** C



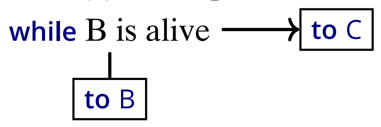
(a) to A for life, then if B is married to B for life, then to C



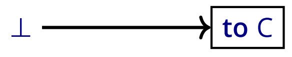
(b) A dies / **while** simplifies



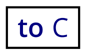
(c) → simplifies



(d) **if** simplifies



(e) B dies / **while** simplifies



(f) → simplifies

Figure 23: An updating example

A's interest is possessory. The condition A is alive is true, so therefore $\delta($**to** A$)$ = **to** A, which does not equal $\perp$. Thus, the **while** A is alive at the left does not simplifiy, i.e., the title tree does not immediately terminate A's interest.

Next, `B marries`. This does not change the truth of the condition A is alive or terminate **to** A, so the tree does not change. The truth of the condition B is married in the **if** changes, as does the truth of B is alive in the second **while**, but since they are not currently at the left, these changes are irrelevant for now. They will become relevant as they reach the left of the title tree.

The real action starts when `A dies`. This *does* change the value of the condition A is alive, which is now false. Now it is the case that $\delta($**to** A **while** A is alive$)$ evaluates to $\perp$, so the overall title tree becomes:

$\perp \rightarrow$ (**if** B is married **then** (**to** $B$ **while** B is alive) **else** $\perp$) $\rightarrow$ **to** C

The simplifications are not yet done. Now the rule for $\rightarrow$ kicks in, because $\delta(\perp \rightarrow, t_2) = \delta(t_2)$. Thus, the first $\rightarrow$ should also be removed from the tree, resulting in:

(**if** B is married **then** (**to** B **while** B is alive) **else** $\perp$) $\rightarrow$ **to** C

But wait, there's more! Now that the **if** *is* now leftmost, the condition B is married must be checked. It is true, which means the **if** takes the "yes" branch, because $\delta($**if** true **then** $t_1$ **else** $t_2) = \delta(t_1)$. The title tree is now:

(**to** B **while** B is alive) $\rightarrow$ **to** C

Now the (formerly second) **while** is at the left. But in this case, its condition is true, as B is alive. Thus the title tree does not simplify further.

Finally, `B dies`. This is a replay of the updates at A's death. The condition B is alive is now false, so $\delta($**to** B **while** B is alive$) = \perp$. Thus the title tree becomes:

$\perp \rightarrow$ **to** C

The remaining $\rightarrow$ drops out, just like the first one did, leaving:

**to** C

Thus, after A's and B's deaths, C's interest is possessory.

These symbolic computations are not difficult, just tedious. But these computations are not meant to be carried out by hand. That's what Littleton and other computer implementations are for. The point of including them here is to show that there are no cards hidden up our sleeves. Every step of the analysis can be made precise, explicit, and mechanical.

What *is* useful to people is the visualization based on it. Figure 23 shows the same example, done visually rather than symbolically.[89]

## *C. Conveyances*

Orlando models the language of conveyances themselves with another grammar. Instead of generating abstract data structures, as the title-tree grammar does, the conveyance grammar generates outputs that look much more like natural language.

Title trees had one primary type *t*, because every title tree can be plugged into any title-tree-shaped hole. That's not the case for conveyances. The their grammatical and logical structure means that there are conveyances whose parts make sense on their own but not together, like `to A and his heirs for life, but if to B while B is married then to C`. Instead, conveyances have a few distinct types:

- A *conveyance* like `O conveys to A for life, then to B until Mars becomes a state` expresses a transfer from a grantor to various recipients.
- A *grant* like `to A and her heirs` creates an interest. One of the most striking things about the conveyance grammar is that individual granting clauses like `to B for life` and combinations of granting clauses like `to C for life, then to D for life` play the same grammatical role.
- A *quantum* like `for life` or `and his heirs` describes what estate an individual grant creates.
- A *condition* like `B is married` or `A survives B` expresses in words a logical condition.
- A *limitation* like `until Mars becomes a state` or `while B is married` uses a condition to terminate an interest.
- A few miscellaneous types round out the list. A conveyance can contain a *person* like `Tilda`, a *pronoun* like `her`, or a natural number *n* like `2` or `10`.

[89] The "yes" and "no" branches in the `if` have been swapped for clarity, but the semantics are the same whichever is drawn on top.

The first rule in the conveyance grammar is the only one for conveyances as a whole:

$$\textit{conveyance} \quad \Rightarrow \quad \textit{owner}\ \texttt{conveys}\ \textit{grant}$$

It says that a *conveyance* consists of a *person* followed by the keyword `conveys` followed by a *grant*. For example:

$$\textit{conveyance} \Rightarrow \overbrace{\texttt{Owner}}^{\textit{person}}\ \texttt{conveys}\ \overbrace{\texttt{to Alice}}^{\textit{grant}}$$

The first part is easy to fill in: *person* can be `Owner` or `O` or any valid name. And `conveys` is always just itself.

But a *grant* is another and different conveyance. For now, consider the first two:

$$\begin{aligned} \textit{grant} \quad &\Rightarrow \quad \texttt{to}\ \textit{person quantum} \\ \textit{grant} \quad &\Rightarrow \quad \textit{grant}\ \texttt{then}\ \textit{grant} \end{aligned}$$

The first option says that a *grant* can consists of the keyword `to` followed by a *person* and a *quantum*. In lawyers' lingo, `to` *person* is the "words of purchase" describing *who* receives the interest and *quantum* the "words of limitation" describing *what* estate they receive. So `to Alice and her heirs` is a valid granting clause; so is `to B for life`. For example:

$$\textit{grant} \Rightarrow \texttt{to}\ \overbrace{\texttt{Alice}}^{\textit{person}}\ \overbrace{\texttt{for life}}^{\textit{quantum}}$$

This is not the only option for a *grant*, which could also consist of two granting clauses, separated by the keyword `then`.[90] For example:

$$\textit{grant} \Rightarrow \overbrace{\texttt{to Alice for life}}^{\textit{grant}}\ \texttt{then}\ \overbrace{\texttt{to Bob}}^{\textit{grant}}$$

This option is the one that makes the grammar powerful enough to write indefinitely long conveyances creating arbitrarily large numbers of interests. Because this rule could be applied recursively, one of the two *grant*s could itself consist of two *grant*s, one or both of which could consist of two more, and so on. Anywhere that a conveyance could contain a single granting clause, it could contain two, three, four, or more of them.

Next consider the rules for a *quantum*, which specifies the duration of an interest.

$$\begin{aligned} \textit{quantum} \quad &\Rightarrow \quad \texttt{and}\ \textit{pronoun}\ \texttt{heirs} \\ \textit{quantum} \quad &\Rightarrow \quad \texttt{for life} \end{aligned}$$

[90] Littleton allows for optional commas between granting clauses, but for simplicity they are not listed in the formal grammar for Orlando.

These are familiar phrases. The language `and ... heirs` specifies a fee simple. The language `for life` specifies a life estate measured by the life of the grantee.

While we are at it, we can also say a bit more about valid names and pronouns.

$$\begin{array}{rcl} \textit{person} & \Rightarrow & \texttt{O}\,|\,\texttt{A}\,|\,\texttt{B}\,|\,\texttt{C}\,|\,\ldots\,|\,\texttt{Alice}\,|\,\texttt{Bob}\,|\,\ldots \\ \textit{pronoun} & \Rightarrow & \texttt{her}\,|\,\texttt{his}\,|\,\texttt{hir}\,|\,\texttt{their}\,|\,\texttt{zir}\,|\,\ldots \end{array}$$

A *person* consists of any single-letter pseudonym like `A`, `B`, `O`, and so on, or any given name like `Alice`, `Bob Terwilliger`, and many others. Littleton allows a *person* to be one or more words, each of which begins with a capital letter.[91] A *pronoun* can be any single word, like `her` or `zir`.[92]

Like the title-tree grammar, the conveyance grammar gives a procedure for generating valid expressions. Start with a variable *conveyance* that represents an arbitrary conveyance. Now apply one of the rules of the grammar to expand a variable. In this case, there is only one variable and exactly one applicable rule, so that *conveyance* expands to *owner* `conveys` *grant*. Now apply another rule to one of the remaining variables, e.g. expand *person* into `Owner`, yielding `Owner conveys` *grant*. Apply another rule to a variable, say expanding *grant* into *grant* `then` *grant*. Repeat, replacing a variable in the current conveyance by the right-hand-side of a rule applicable to it as as needed, until there are no variables left to expand. Here is an example, one that generates our by-now familiar friend: a life estate followed

[91] The specific set of valid names is not further defined here, since the details are unimportant. All that matters is that `Alice` is always `Alice`, and that `Alice` and `Bob` are distinct.

[92] *Cf.* Sprowl, *supra* note 54, at 48-49 (describing a program that asks about a testator's and spouse's preferred pronouns, but then discarding it in favor of one that asks whether the testator is male and assigns pronouns "assuming, of course, a heterosexual marriage"). We submit that Littleton's is the better approach; it is both more respectful and computationally simpler.

by a remainder.

  *conveyance*  
⇒ *person* `conveys` *grant*  
⇒ `Owner conveys` *grant*  
⇒ `Owner conveys` *grant* `then` *grant*  
⇒ `Owner conveys to` *person quantum* `then` *grant*  
⇒ `Owner conveys to Alice` *quantum* `then` *grant*  
⇒ `Owner conveys to Alice for life then` *grant*  
⇒ `Owner conveys to Alice for life then to` *person quantum*  
⇒ `Owner conveys to Alice for life then to Bob` *quantum*  
⇒ `Owner conveys to Alice for life then to Bob and his heirs`

Littleton runs this process in reverse; it starts from the text of a conveyance like the one in the last line and reconstructs the sequence of rules that yielded it. This process, called ***parsing***, tells Littleton what the linguistic structure of a conveyance is, and how its parts fit together.[93]

## *D. Translation*

The final piece of the formalization of Orlando is the conversion from conveyances to title trees. This is carried out by a translation function written using a new kind of notation:

$$[\![\texttt{to A for life}]\!]_O \quad = \quad \textbf{to}\ \mathsf{A}\ \textbf{while}\ \mathsf{A\ is\ alive}$$

[93] At an abstract level, Littleton does recursive descent parsing with backtracking. It reads a conveyance from left to right, using a list of of Orlando's grammar rules to test out different ways of generating the conveyance's language. Whenever its current hypothesis is contradicted by the next part of the conveyance, it backs up to the last point at which it had multiple options and tries the next available one it has not previously tried. Littleton uses the MParser library for OCaml, which is derived from the Parsec library for Haskell. *See* Daan Leijen & Erik Meijer, Parsec: Direct Style Monadic Parser Combinators For The Real World (2001) (unpublished manuscript), https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/parsec-paper-letter.pdf (describing Parsec); Graham Hutton & Erik Meijer, Monadic Parser Combinators (1996) (unpublished manuscript) (Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham) (describing approach taken by Parsec). *See generally* ALFRED V. AHO, MONICA S. LAM, RAVI SETHI & JEFFREY D. ULLMAN, COMPILERS: PRINCIPLES, TECHNIQUES, AND TOOLS (2d ed. 2006) (overview of parsing); DICK GRUNE & CERIEL J.H. JACOBS, PARSING TECHNIQUES: A PRACTICAL GUIDE (2007) (same, in more detail).

The double (or "denotation") brackets $[\![]\!]$ are computer-science notation to indicate the meaning or translation of the expression that appears between the brackets.[94] They describe a function from conveyances to title trees.[95] That is, the title tree fragment on the right hand side of the = is the translation of the conveyance fragment on the left hand side between the double brackets. (The subscript on the brackets keeps track of information abut the context—in this case, who the grantor is.)

Just as the update function $\delta()$ is defined by cases on the title tree grammar, the translation function $[\![]\!]$ is defined by cases on the conveyance grammar. Every rule in the conveyance grammar has a corresponding translation rule. In this approach, which is known as ***syntax-directed translation***, there is exactly one translation rule applicable for each syntactic option, so the translation is fully determined by the syntax of an expression.[96] For example, here are the grammar and translation rules for two successive granting clauses linked by `then`.

$$\begin{aligned} grant &\Rightarrow grant \texttt{ then } grant \\ [\![grant_1 \texttt{ then } grant_2]\!]_o &= [\![grant_1]\!]_o \rightarrow [\![grant_2]\!]_o \end{aligned}$$

Just as a *grant* is formed by recursively applying grammar rules, so too is $[\![grant]\!]$ recursively computed by recursively applying translation rules. And just as a *grant* can consist of *grant* `then` *grant*, when it does, the value of $[\![grant_1 \texttt{ then } grant_2]\!]$ is computed from $[\![grant_1]\!]$ and $[\![grant_2]\!]$. The process of translating a

[94] The notation is required to deal with a nuance of quotation. Using denotation brackets to describe a function indicates that keywords such as "`for life`" are to be read literally when the function is computed, but variables such as "*p*" are to be replaced with their values. The symbol was chosen because it was available on the typewriter the computer scientist Dana Scott was using at the time. Brian Rabern, *The History of the Use of* $[\![.]\!]$*-Notation in Natural Language Semantics*, 9 SEMANTICS & PRAGMATICS 12 (2016).

[95] We omit the details of Littleton parses the string of characters describing a conveyance to recognize which translation rules apply. The high-level version is that it first translates the language of a conveyance into a ***abstract syntax tree*** (AST) that describes which rules of the conveyance grammar were used to generate it. The AST is to the text of a conveyance as the visual diagram of a title tree is to its symbolic description. So to be more precise, the translation function is a function from ASTs to title trees. *See* Property Conveyances, *supra* note 7 (documenting Orlando ASTs and the translation function).

[96] *See* AHO, LAM, SETHI & ULLMAN, *supra* note 93 (discussing syntax-directed translation).

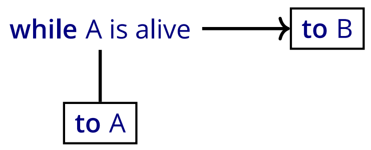


Figure 24: Translation of a conveyance

conveyance just consists of applying $[\![]\!]$ to successively smaller portions of a conveyance, as in the following example:

$$\begin{aligned}
& [\![\texttt{to A for life, then to B and her heirs}]\!]_{\mathsf{O}} \\
= \quad & [\![\texttt{to A for life}]\!]_{\mathsf{O}} \rightarrow [\![\texttt{to B and her heirs}]\!]_{\mathsf{O}} \\
= \quad & [\![\texttt{to A}\quad \textsf{A is alive}]\!]_{\mathsf{O}} \rightarrow [\![\texttt{to B and her heirs}]\!]_{\mathsf{O}} \\
= \quad & (\textsf{\textbf{to} A \textbf{while} A is alive}) \rightarrow [\![\texttt{to B and her heirs}]\!]_{\mathsf{O}} \\
= \quad & (\textsf{\textbf{to} A \textbf{while} A is alive}) \rightarrow [\![\texttt{to B}\quad \texttt{true}]\!]_{\mathsf{O}} \\
= \quad & (\textsf{\textbf{to}}\ A\ \textsf{\textbf{while} A is alive}) \rightarrow \textsf{\textbf{to} B}
\end{aligned}$$

The resulting title tree, shown in Figure 24, should look familiar.

This systematic dependence of the meaning of larger compound expressions on the meanings of smaller and simpler subexpressions contained within them is known as ***compositionality***, and it is an important part of what makes an approach based in programming-language theory so fruitful.[97]

## *E. Conclusion*

This completes the initial survey of Orlando. It is not a complete presentation; indeed, it is not even a complete presentation of the fragment of core Orlando documented in the Appendix. Instead, it is meant to give a clear sense of what Orlando does and how it does it. The rest is, more or less, more of the same.

How much more? The core subset of Orlando in the Appendix also includes rules to generate and process special limitations (e.g., `to A until A marries`), executory limitations (e.g., `to A, but if B graduates college to B`), conditions subsequent (e.g., `to A, but if the property is used as a school the grantor may reenter`), implied reversions (e.g., as in `O conveys to A for life`), and successive conveyances, (e.g., `O conveys to A for life, then to B.`

[97] *See* Scott & Strachey, *supra* note 81, at 12; Winskel, *supra* note 83, at 60. *See generally* Zoltán Gendler Szabó, *Compositionality*, Stan. Encyclopedia Phil. (2020), https://plato.stanford.edu/archives/fall2020/entries/compositionality/.

`A conveys to C.`). The full version of Orlando as implemented in Littleton handles far more, including:

- Estates for a term of years and in fee tail, including disentailment by conveyance of a fee simple.
- Simplification, which removes unreachable interests from the title tree so that it more closely reflects the state of title as a lawyer would describe it.
- Naming of interests (e.g., "remainder in fee simple subject to executory limitation").
- Vesting and the Rule Against Perpetuities.
- Miscellaneous doctrines, such as the Rule in Shelley's Case, the Doctrine of Worthier Title, and merger.
- Concurrent interests as tenancies in common, joint tenancies, and tenancies by the entireties.
- Wills, intestacy, and escheat.[98]
- Class gifts (e.g., `to the children of A`), vesting subject to open, and the rule of convenience.

The fragment discussed in this Part is just the tip of the iceberg. But it is made of the same frozen $H_2O$ as the rest; if you understand how Orlando and Littleton model a life estate, you understand how it is possible for them to model everything else.

## IV. Lessons for Property Law

*Why teach future interests to a computer?*, you may be wondering. *Hasn't the poor thing suffered enough already?*

Most obviously, we hope that Orlando and Littleton will help property scholars better understand the doctrinal rules of future interests. Orlando's rules are precise and concise; they make it easy to see how a special limitation differs from an executory limitation, how a reversion differs from a possibility of

[98] *See* Lilian Edwards, *Building an Intestate Succession Advisor: Compartmentalisation and Creativity in Decision Support Systems*, *in* INFORMATICS AND THE FOUNDATIONS OF LEGAL REASONING 311 (Zenon Bankowski, Ian White, & Ulrike Hahn ed., 1995) (describing expert system for Scots intestacy law).

reverter. Whether they read Orlando's rules or simply use Littleton to analyze conveyances, scholars may see a familiar subject in fresh ways. Future interests are the right hybrid of simple and complex to be illuminated by formalization.[99]

We also hope that Littleton will be useful as a teaching tool.[100] Teachers can generate expository diagrams in seconds and walk their students through the consequences of various contingencies. Some people are visual learners; having a good, consistent, visualization tool will help them understand how future interests fit together. In particular, because Littleton is interactive, it enables students to learn through self-directed exploration. The student wondering *What happens if A dies?* can type `A dies` and find out. Teachers and students can use Littleton without even knowing that Orlando exists.[101]

More profoundly, the fact that future interests can be formalized *as a programming language* provides deeper insights into property theory. For example, the fact that Orlando uses only a small and carefully defined set of title-tree node types validates the idea that property law follows the *numerus clausus* principle—that property interests come only in a set of specific predefined forms. This Part considers what Orlando has to say to Property scholars. It explains the design philosophy behind Orlando and Littleton, describes how they can illuminate specific property doctrines, and and discusses the broader systemic insights they offers into property-law theory.

## A. Design Principles

Before discussing what Orlando and Littleton have to say about property law, it will be helpful to say a bit about why they are the way they are.

[99] *See TAXMAN*, *supra* note 20, at 842-43.

[100] On teaching AI and law, see Kevin D. Ashley, *Teaching Law and Digital Age Legal Practice with an AI and Law Seminar*, 88 Chi.-Kent L. Rev. 783 (2012).

[101] For more interesting recent work on generating teaching problems automatically, see *Lawsky Practice Problems*, *supra* note 77.

## 1. Orlando

Orlando title trees have a number of overlapping design principles. First, they are ***minimal***. The core language of title trees is as small as possible. This makes it easier to get the semantics right. Compared with the dozens of distinct interests possible under the traditional naming system, a core set of five title-tree node types is clean and easy to reason about.

Relatedly, the semantics of title trees are ***simple***. The update function $\delta()$ can be defined in a handful of lines. It is also easy to define functions on title trees (like the path analyses of reachability and vesting) and be confident that they are correct. Simplicity makes it easier to teach how Orlando works, and easier to create implementations like Littleton.

The core title-tree node types are also ***orthogonal***.[102] Each node type implements a different concept: it does one thing and one thing only. A `to` node represents ownership, and $\perp$ represents its absence. A `while` represents termination: events can cause an interest to come to an end. An `if` represents choice: either A or B but not both. And a $\rightarrow$ represents sequencing: A follows B, in that order. They are good primitives because they don't mix concepts: a `while` node terminates without having to worry about what comes next.[103]

Simplicity and orthogonality help make title trees ***modular***.[104] The node types can be freely combined. $t_1$ and $t_2$ in $t_1 \rightarrow t_2$ can be anything; there are no hidden restrictions on what kinds of nodes can appear in them. This means that title trees are loosely coupled; there are no confusing or hard-to-model interactions between remote parts of one. Functions like $\delta()$ can treat subtrees as black boxes, without needing to look inside at their details.

[102] *See* SEBESTA, *supra* note 16, at 9 ("Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.").

[103] An influential article on building languages out of orthogonal primitives is Peter J. Landin, *The Next 700 Programming Languages*, 9 COMM. ACM 157 (1966). *See also* Guy L. Steele, Jr.,, Growing a Language (1998) (unpublished manuscript), http://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf.

[104] Some authors treat modularity as a part of orthogonality. *See* SEBESTA, *supra* note 16, at 9 ("Furthermore, every possible combination of primitives is legal and meaningful.").

Despite this, title trees are ***expressive***: they suffice to model a wide variety of estates and future interests. A complicated property concept like "fee simple subject to condition subsequent" can be broken down into a suitable combination of title tree nodes. Expressivity is a design constraint; orthogonality and modularity are ways of achieving it despite minimality and simplicity.

And finally—and this is necessarily more subjective—title trees are ***faithful*** to the property-law concepts they model. A title tree bears a close resemblance to the conveyance it comes from—not perfect, but close. The translation rules are also minimal, simple, orthogonal, and modular. The translation of conveyances into title trees respects their structure.

Taken all together, these properties describe an elusive notion of ***elegance*** in programming-language design. An elegant language is mathematically clean but also easy to use; it enables a programmer to see at a glance how their code will work and what it will do.[105] The theoretical core of programming-language theory—the "lambda calculus" invented by logician Alonzo Church in the 1930s—has just three primitive operations and its semantics has only one rule, but it is expressive enough to model any program in any programming language, and clean enough that computer scientists use it to explain concepts to each other.[106] Orlando's title trees aren't on that level of exquisite elegance, but we have tried to make them as elegant as we could.

[105] On elegance in software, *see generally* BEAUTIFUL CODE (Andrew Oram & Greg Wilson eds., 2007), and in particular Yukihiro Matsumoto, *Treating Code as an Essay* (Nevin Thompson trans.), *in id.* at 477.

[106] *See generally* H.P. BARENDREGT, THE LAMBDA CALCULUS (rev. ed. 1985) (encyclopedic reference); Raul Rojas, A Tutorial Introduction to the Lambda Calculus (2015) (unpublished manuscript), https://arxiv.org/abs/1503.09060 (gentle technical introduction); RAYMOND SMULLYAN, TO MOCK A MOCKINGBIRD (1985) (non-technical introduction via puzzles about songbirds); *Alligator Eggs!*, WORRYDREAM, http://worrydream.com/AlligatorEggs/ (non-technical introduction via cute drawings of alligators).

2. Littleton

Littleton is primarily implemented in about 3,000 lines of code in the OCaml programming language.[107] It is designed as a web app; the user interface is a mixture of HTML, CSS, and JavaScript. The back-end OCaml code is compiled to JavaScript, which means that it runs entirely in the user's browser. The official site at https://conveyanc.es includes include three versions of Littleton: (1) a live online version that anyone can use, (2) the complete source code for anyone to copy and modify, and (3) a prebuilt version that anyone can copy to their own server and run.

Compared with more familiar languages such as C, Python, and Java, OCaml may seem like an unusual choice. Three features of OCaml, however, make it particularly well-suited for writing DSL interpreters like Littleton.

First, OCaml is ***functional***.[108] An OCaml program is not a sequence of steps to execute—*do this, then do that*. Instead, it consists almost entirely of functions to be evaluated. All of the important parts of Littleton are described and implemented as functions. This leads to a close correspondence between the the functional definitions of Orlando and their implementation in Littleton. The translation function and the update function are both functions in Littleton's source code. So are the functions that compute reachability and vesting along paths, the function that derives a human-readable name from an interest, and many more. Adding a new feature is generally a matter of defining a

[107] *See generally* ANDREW W. APPEL, MODERN COMPILER IMPLEMENTATION IN ML (1998) (describing the implementation of compilers and interpreters in the ML language, form which OCaml is derived).

[108] For introductions to functional programming languages and a functional approach to programming, see generally HAROLD ABELSON & GERALD JAY SUSSMAN WITH JULIE SUSSMAN, STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS (2d ed. 1996) (canonical); FRIEDMAN, DANIEL P & FELLEISEN, MATTHIAS, THE LITTLE SCHEMER (4th ed. 1995) (gentle); MIRAN LIPOVACA, LEARN YOU A HASKELL FOR GREAT GOOD!: A BEGINNER'S GUIDE (2011) (quirky); ALVIN ALEXANDER, FUNCTIONAL PROGRAMMING, SIMPLIFIED (2017) (verbose). A classic manifesto in favor of functional languages is John Backus, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, 21 COMM. ACM 613 (1978). McCarty's TAXMAN was written partly in the functional language LISP, from which OCaml derives. *TAXMAN*, *supra* note 20, at 850 n.54.

```
let rec delta (t:term) : term =
  match t with
  | Bottom   -> Bottom
  | Atom i   -> Atom i
  | Seq(t1,t2) ->
    if delta t1 = Bottom then delta t2
    else Seq(delta t1, t2)
  | If(c,t1,t2) ->
    if (C.eval c) then delta t1
    else delta t2
  | While(s,c,expr1) ->
    if (not (C.eval c)) || delta expr1
      = Bottom then Bottom
    else While (s, C.delta c, delta
     expr1)
```

Figure 25: Definition of $\delta()$ in Littleton source code

new mathematical function and then translating that function into OCaml code. This has helped immensely with rapid prototyping of different possible approaches.

Second, OCaml is ***strongly typed***. Data comes in many different types. In Orlando, types include conditions, title trees, and conveyances. In other programming languages, types often include integers, strings, and lists. Some languages allow programmers to store any kind of data in any variables and manipulate it freely. Sometimes, this leads to type conflicts, where the programmer tries to combine incompatible types of data, like trying to add the number `5` and the string `hello good sir`, which can cause bugs and crashes. OCaml checks the type of every variable and every piece of data when a program is compiled, and produces an error then, rather than later when the program is run. This means that many kinds of bugs are caught and prevented early. Strong typing helped us be careful, for example, to keep conveyances and title trees distinct. For programs (like Littleton) that interpret other programs (like conveyances), this strict separation between different types of data is quite helpful in preventing subtle design mistakes.

Third, OCaml uses a powerful form of ***pattern-matching***.[109] For a kind of data—like a title tree—that comes only in one of a finite number of forms, OCaml lets programmers define functions by cases on those forms. For example, the Orlando function $\delta()$ is defined by cases on a title tree. Compare its definition with the OCaml code implementing it in Figure 25. This simply *is* the definition of $\delta()$, transposed into slightly different notation. Other core parts of Littleton, like the translation and simplification code, are similarly transparent. Again, this is particularly useful when writing programming-language interpreters. It is easy to implement mathematical semantic definitions and to confirm that the implementation matches the definition. We took advantage of this simplicity to experiment with dozens of different definitions on our way to the final version of Orlando.

## *B. Insights into Property Doctrine*

Formalization has a disciplining effect; it forces implementors to be precise about the meaning and behavior of constructions, and this in turn helps users to be precise about what they intend. This subsection discusses three ways in which a formalized approach helps clarify specific doctrinal rules.

### 1. Defaults

There are several linguistic subtleties associated with parsing life estates. One of them is that a conveyance can create an interest with no explicitly specified quantum. The words `to A` by themselves are sufficient to give A an interest. So Orlando includes the grammar rule:

$$\textit{quantum} \rightarrow \epsilon$$

which says that a *quantum* can consist of $\epsilon$, a standard computer-science notation for the "empty string," i.e. no characters, not even a space. ($\epsilon$ is to conveyances as $\perp$ is to title trees.) The grammar uses it as a placeholder to represent the default quantum that an interest has when none is given.

But this begs the question: what *is* the default quantum? This is one of the classic doctrinal gotchas taught to Property

[109] *See* Minsky & Weeks, *supra* note 36 (explaining pattern-matching); *see also TAXMAN*, *supra* note 20, at 855 (discussing use of pattern-matching in TAXMAN).

$$\begin{aligned}
[\![\texttt{and}\ldots\texttt{heirs}]\!]_p &= \text{true}\\
[\![\epsilon]\!]_p &= p \text{ is alive}\\
[\![\texttt{for life}]\!]_p &= p \text{ is alive}\\
[\![\texttt{for the life of}\,q]\!]_p &= q \text{ is alive}
\end{aligned}$$

(a) Default quantum (older rule)

$$\begin{aligned}
[\![\texttt{and}\ldots\texttt{heirs}]\!]_p &= \text{true}\\
[\![\epsilon]\!]_p &= \text{true}\\
[\![\texttt{for life}]\!]_p &= p \text{ is alive}\\
[\![\texttt{for the life of}\,q]\!]_p &= q \text{ is alive}
\end{aligned}$$

(b) Default quantum (modern rule)

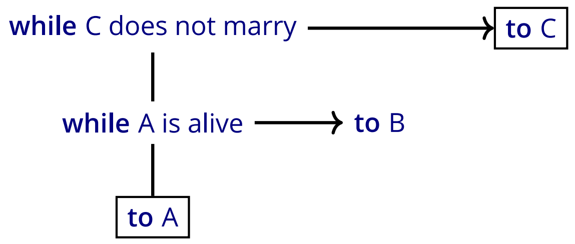
Figure 26: Two different translation rules

students. Well into the 20th century, some courts held that a bare grant of the form `to A` with no words explicitly describing the quantum of interest being granted was held to effectively create a life estate.[110] But over time that default flipped, and the modern rule is that it creates a fee simple. [111]

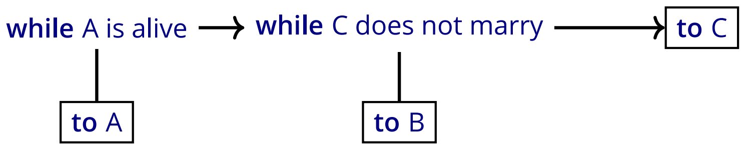
Orlando models this shift by using *two different translation rules*. Compare the two sets of translation rules in Figure 26. In the former, which corresponds to the earlier doctrine, the language `to A` creates a life estate, just like `to A for life` does. In the latter, which corresponds to the modern doctrine, the language `to A` creates a fee simple, just like `to A and their heirs` does. Neither definition is correct or incorrect in the abstract; each of them is useful for modeling a different set of legal rules. Formalization makes this doctrinal change over time easy to see and easy to describe. Note that the two sets of rules differ in exactly one line; this change is a nice example of modularity in that it does not affect the rest of Orlando or of property law.

110 *See, e.g.,* Cole v. Steinlauf, 144 Conn. 629, 631-32 (1957). *See generally* THOMAS F. BERGIN & PAUL G. HASKELL, PREFACE TO ESTATES IN LAND AND FUTURE INTERESTS at 24-27 (2nd ed. 1984) (discussing origin and construction of ”and his heirs” language).

111 *See, e.g.,* Dennen v. Searle, 149 Conn. 126, 135-39 (1961) (discussing the history of the default and overruling *Cole*).



(a) `(to A for life, then to B), but if C marries to C`



(b) `to A for life, then (to B, but if C marries to C)`

Figure 27: Two alternative title trees

Concretely, Littleton implements *both* sets of translation rules for a default quantum. The modern rule is enabled by default. But a configuration setting in Littleton allows the user to toggle between the earlier preference for a life estate and the modern preference for a fee simple.

## 2. Syntactic Ambiguity

For another kind of ambiguity lurking beneath the surface of the usual informal presentation of future interests, consider the grant

```
to A for life, then to B, but if C marries to C
```

If A dies and then C marries, it is clear that C's interest should divest B's. But what if C marries while A is still alive? Is A or C entitled to possession? Is A's interest also subject to this executory limitation in favor of C? This is a *scope* problem. The limitation could have *broad* scope and apply to both A's interest and B's, i.e.:

```
(to A for life, then to B), but if C marries to C
```

or it could have *narrow* scope and apply only to B's, i.e:

```
to A for life, then (to B, but if C marries to C)
```

The two diagrams in Figure 27 depict the two possibilities.

$$\begin{aligned} \textit{grant} &\Rightarrow \textit{grant}\ \texttt{then}\ \textit{grant} \\ \textit{grant} &\Rightarrow \textit{grant}\ \texttt{then}\ (\textit{grant}) \\ \textit{grant} &\Rightarrow \textit{grant}\ \texttt{then}\ (\textit{grant}\ \texttt{but if}\ \textit{condition grant}) \end{aligned}$$

(a) Narrow scope

$$\begin{aligned} \textit{grant} &\Rightarrow \textit{grant}\ \texttt{but if}\ \textit{condition grant} \\ \textit{grant} &\Rightarrow (\textit{grant})\ \texttt{but if}\ \textit{condition grant} \\ \textit{grant} &\Rightarrow (\textit{grant}\ \texttt{then}\ \textit{grant})\ \texttt{but if}\ \textit{condition grant} \end{aligned}$$

(b) Broad scope

Figure 28: Derivations with different scope

It is not the case that one of these diagrams is right and the other is wrong. There are conveyances in which a broad scope is appropriate and intended; there are conveyances in which a narrow scope is appropriate and intended. A language for modeling conveyances should not force once choice or the other. It should support both, just as it supports both fees simple and life estates.

Thus, the grammatical rule for executory limitations makes explicit which clauses are subject to them.

$$\textit{grant} \Rightarrow \textit{grant}_1\ \texttt{but if}\ \textit{condition}\ \textit{grant}_2$$

This rule looks like the `then` rule, and it seems like it adopts broad scope: every interest before the executory limitation is subject to it. But that is not quite what it says. Instead, whatever granting clauses are generated by $\textit{grant}_1$ are subject to the executory limitation *condition*. Consider the two derivations in Figures 28, in which parentheses have been added between the two steps in the derivation. The parentheses, like the parentheses in $(10 * 2) + 3$ and $10 * (2 + 3)$, clarify which of two possible interpretations is correct. Note that in the first derivation, with narrow scope, only one of the two interests before the `but if` was generated from $\textit{grant}_1$ in this rule.

From a formal perspective, the scope ambiguity matters because the semantics of `but if` are not *associative*. Compare the semantics of `then`, which are associative. Consider the conveyance

```
to A for life, then to B for life, then to C
```

which can be understood as

```
        to A for life, then (to B for life, then to C)
```

or as

```
        (to A for life, then to B for life), then to C
```

In this case, the syntactic ambiguity does not matter and the resulting title trees will be semantically equivalent. The keyword `then` and its translation, operator $\rightarrow$, are associative in the same way that addition is: $2 + (3 + 4) = (2 + 3) + 4$. It does not matter where the parentheses go. This is not an accident. The relationship "$x$ comes before $y$" is naturally captured by an associative operator, and the definition of $\delta()$ has been carefully chosen so that it is associative: $x \rightarrow (y \rightarrow z)$ behaves identically to $(x \rightarrow y) \rightarrow z$. But the relationship of one interest divesting a previous one is not naturally associative, which produces a recurring scope ambiguity, which Orlando deals with using explicit parentheses.

## 3. "Theorems" of Property Law

Orlando's definitions are mathematical. The are written using the dialect of mathematical notation used by programming-language theorists, and they describe Orlando's syntax and semantics in terms of abstract mathematical structures. This isn't just a notational convenience; it opens up new ways to reason about Orlando programs, and thus about property law.

For example, consider the proposition that a fee simple is perpetual. More precisely, after any possible sequence of events, the owner of a fee simple will still hold a fee simple. Let us write this out formally in mathematical notation. Let $h$ refer to a history: a sequence of events. And let $\Delta_h(t)$ be the result of using $\delta()$ to update the title tree $t$ with the events in the history $h$, one at a time.[112] Then the proposition that a fee simple is perpetual is the proposition that the following equation holds for all possible histories $h$:

$$\Delta_h(\textbf{to } p) = \textbf{to } p$$

The proof is by mathematical induction: if the equation holds for the history with no events and it holds on a history with $n+1$

[112] $\Delta()$ is the "capitalized" version of $\delta()$: it is $\delta()$ applied to multiple events. For the details of how $\Delta_h()$ is defined, see Property Conveyances, *supra* note 7.

events whenever it holds on a history with $n$ events, then it holds for all histories, regardless of how many events they contain.[113]

Start with the case where $h$ contains no events. Then it is trivially true that $\Delta_h(\texttt{to}\ p) = \texttt{to}\ p$ because the title tree does not need to be updated when there are no events to update it with.

Now consider the case where $h$ contains $n + 1$ events. Then we can rewrite $\Delta_h(\texttt{to}\ p)$ as $\Delta_e(\Delta_{h'}(\texttt{to}\ p))$, where $h'$ is the first $n$ events in $h$ and $e$ is the most recent event. I.e., first update the title tree with $h'$ and then update it with $e$. But we are allowed to assume that $\Delta_{h'}(\texttt{to}\ p)) = \texttt{to}\ p$, because $h'$ has only $n$ events in it. So $\Delta_e(\Delta_{h'}(\texttt{to}\ p)) = \Delta_e(\texttt{to}\ p)$. To update a title tree by a single event $e$, we evaluate $\delta()$ on the tree after the event takes place. But by the definition of $\delta()$ for `to` nodes, $\delta(\texttt{to}\ p) = \texttt{to}\ p$ regardless of what $e$ is. Thus we have established

$$\Delta_h(\texttt{to}\ p) = \Delta_e(\Delta_{h'}(\texttt{to}\ p)) = \Delta_e(\texttt{to}\ p) = \texttt{to}\ p$$

which completes the proof that the equation holds where $h$ contains $n + 1$ events.

This is a comparatively simple proof. Other proofs about programming languages use more sophisticated forms of induction, other types of semantics, and even computer assistance. In previous work directed at computer scientists, we formalized four other "theorems" of property law: that ownership is unambiguous; first in time, *nemo dat quod non habet*; first in right; and conservation of estates.[114]

This is a novel way of thinking about familiar heuristics, which gives old claims about property law a new kind of theoretical content. These proofs do not mean that judges would find that a particular person has a right to possession. Mathematical proofs do not establish legal propositions.[115] But they can help to show that Orlando is a faithful model of property law. A legal proposition—*a fee simple is perpetual*—can be given a precise formulation as a claim about the behavior of an Orlando program, and that formulation can be proven true given Orlando's definitions. So we have rigorous assurance that Orlando satisfies important desiderata of a model of property law.

More ambitiously, restating claims about doctrine in terms of claims about formal programming-linguistic models can help

113 *See* Sipser, *supra* note 82, at 23 (explaining mathematical induction).
114 *See* Property Conveyances, *supra* note 7.
115 *See TAXMAN*, *supra* note 20, at 841.

clarify the content of those claims and make the points of disagreement among scholars more evident. The leading future-interests scholars of the first half of the twentieth century conducted a lengthy and arid debate over how to classify the interest held by the residuary devisee of a will `to A for life, then to those of A's children who survive A`.[116] The issue might have been joined more clearly and conclusively if they had made their arguments with dueling semantics that reduced their various positions to a common (programming) language.

## C. *Insights into Property Theory*

Orlando holds a mirror up to the common-law system of estates and future interests. Some things are easier to see from another angle. Modeling conveyances as a programming language helps to explain property law. In particular, three features of property law noted by Thomas W. Merrill and Henry E. Smith are thrown into sharp relief.

### 1. The *Numerus Clausus*

First, property law is subject to the *numerus clausus* (Latin for "closed number"): interests in property can exist only in a finite number of forms.[117] Property lawyers and their clients must work within the existing forms. Consider the case of *Johnson v. Whiton*, where Royal Whiton's will purported to create an estate "to . . . Sarah A. Whiton and her heirs on her father's side." As Oliver Wendell Holmes, Jr. put it, "A man cannot create a new kind of inheritance. . . . [I]f the words 'on her father's

[116] *See* Wythe Holt, *The Testator Who Gave Away Less Than All He or She Had: Perversions in the Law of Future Interests*, 32 ALA. L. REV. 69, 84 (1980) (reviewing and criticizing their answers, and proposing that it be called a "perversion").

[117] Thomas W. Merrill & Henry E. Smith, *Optimal Standarization in the Law of Property: the Numerus Clausus Principle*, 110 YALE L.J. 1, 3 (2000) [hereinafter *Optimal Standardization*]; *see also* Henry E. Smith, *Standardization in Property Law* [hereinafter Standardization], *in* RESEARCH HANDBOOK ON THE ECONOMICS OF PROPERTY LAW 148 (Kenneth Ayotte & Henry E. Smith eds., 2011); Christina Mulligan, *A Numerus Clausus Principle for Intellectual Property*, 80 TENN. L. REV. 235, 237-42 (2012).

side' do not effect the purpose intended, they are to be rejected, leaving the estate a fee simple . . . ."[118]

Orlando embraces the *numerus clausus* with a vengeance. The acceptable forms of conveyances are limited to those generated by Orlando's conveyance grammar. Unless the language of a conveyance can be fit into one of the allowable patterns, it is not valid Orlando syntax, and Littleton will not attempt to interpret it. Similarly, the acceptable forms of interests are limited to those that can exist in a title tree generated by its title-tree grammar. Here is the code in Littleton that defines title trees:

```
type t =
| Bottom
| Atom  of interest
| Seq   of t * t
| While of source * condition * t
| If    of condition * t * t
```

In programming-language terms, this code defines the type of title trees; it says that a title tree can have one of these five forms, and no others. The *numerus clausus*, in other words, is a statement about the types of property law.[119]

The restriction on allowable wording is a feature of programming languages, not a feature of property law. A perfectly plausible English-language conveyance like "I hereby convey to my beloved nephew Geoffrey forever and ever, and to his heirs successors and assigns, to have and to hold, free and clear, as their own property" is not part of the fragment of conveyancing Orlando and Littleton attempt to model. To be clear, lawyers and judges might recognize this as creating a fee simple in Geoffrey, and computer scientists might attempt to write a natural-language-processing system that could infer that this language creates a fee simple. But that is not part of this project; we have focused on the underlying legal structure, rather than all possible nuances of natural language.

In contrast, the restriction on allowable interests very much is a feature of property law.[120] Littleton never reports that a title tree corresponds to a new or unknown kind of interest; it can

[118] Johnson v. Whiton, 159 Mass. 424, 426 (1893).

[119] *See generally* PIERCE, *supra* note 85 (discussing type theory).

[120] The reason for the *numerus clausus* principle is more disputed than its existence. *Compare Optimal Standardization*, *supra* note 117 (information

fit any possible title tree into the finite framework of property law. There is no node or combination of nodes that corresponds to Royal Whiton's desire to restrict Sarah Whiton's interest to her heirs "on her father's side." That Orlando and Littleton shoehorn every interest into the few forms of title trees they recognizes is a feature, not a bug, because that is how property law works.

Orlando also shows that the *numerus clausus* is far less restrictive than it seems. The forms of title trees are extraordinarily general. An interest can be subjected to arbitrary conditions on how it starts and ends, and interests can be combined in arbitrarily long chains. The common-law catalog of allowable interests (herein of "remainder in fee simple subject to executory limitation") is misleading because it is a catalog of names. By modeling the functional behavior of interests, rather than starting from their familiar names, Orlando shows how it is possible to work with great flexibility using only the basic elements provided by property law. Some kinds of ownership—like Royal Whiton's patriarchal folly—are still unrepresentable. But more is possible within the common-law system than it seems at first.

## 2. Recursivity

Merrill and Smith observe that the system of future interests is *recursive*:

> These rules can feed into themselves. For example, a fee simple can be physically divided and divided yet again, or a lessee can create a sublease and the sublessee a (sub)sublease, etc.[121]

They note that recursivity is a feature of natural languages,[122] and in their casebook they observe that it is a feature of programming languages as well.[123] So it is with Orlando. The conveyance grammar is recursive: a granting clause be expanded

costs), *with* Chad J. Pomeroy, *The Shape of Property*, 44 Seton Hall L. Rev. 797 (2014) (historical evolution).

[121] *Optimal Standardization*, *supra* note 117, at 36-37; *see also* Henry E. Smith, *Property as the Law of Things*, 125 Harv. L. Rev. 1691, 1707-08 (2012) [hereinafter Law of Things].

[122] *Optimal Standardization*, *supra* note 117, at 36-37.

[123] Thomas W. Merrill & Henry E. Smith, Property: Principles and Policies 529 (3rd ed. 2017).

into pair of granting clauses joined by `then`, and these granting clauses can be expanded again, and so on an indefinitely large number of times. And the title tree grammar is recursive, too: every **while**, **if**, and → can be expanded into any title tree. Correctly capturing the recursivity of future interests was a key design principle for Orlando.

More than that, Orlando sheds light on how property law is recursive. The legal language of conveyances is recursive in the same way way that Orlando's conveyance grammar is recursive: arbitrarily long sequences of granting clauses can be strung together. And the legal relations of future interests are recursive in the same way that Orlando's title trees are: title can be carved up into arbitrarily large numbers of successive interests. Orlando shows that these two kinds of recursivity are closely related, because its syntax-directed translation of conveyances into title trees maps recursive rules of the one into recursive rules of the other. Property *law* can accommodate recursively divided ownership, but the actual work of division is done by property *language*, which partakes of natural language's recursivity. Orlando foregrounds and formalizes that relationship.

As Merrill and Smith point out, the analogy to natural language breaks down because natural language is far more flexible.[124] But this is precisely where a formal treatment based in programming language theory can be *more* faithful to the domain being modeled than the usual linguistic characterizations lawyers trade in. Students who have spent months learning to look for ambiguities in the language of a legal test and to challenge the application of every rule are often frustrated to encounter a legal domain with bright-line rules that leave little room for argument. In this respect, the notorious inflexibility of computer programming may be an accurate reflection of the law of future interests.

### 3. Modularity

Smith has argued at length that private law in general and property law in particular make extensive use of modularity to economize on information costs.[125] By decomposing legal re-

[124] *Optimal Standardization*, *supra* note 117, at 37-38.

[125] *See* Law of Things, *supra* note 121; Standardization, *supra* note 117; *Optimal Standardization*, *supra* note 117; Henry E. Smith, *ModularIty in*

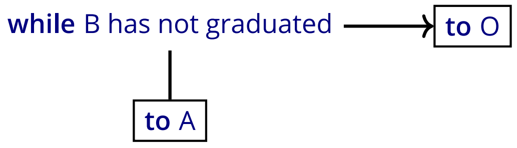
lationships into weakly coupled units, modularity makes it possible for actors to focus on the small number of modules that directly affect them, without having to worry about the legal consequences of other more remote modules.[126] Where the *numerus clausus* emphasizes the standardization of individual modules, modularity emphasizes the separability of modules from each other. In property law, Smith focuses on the way in which discrete and distinguishable " legal things" mediate property rights. Some of these things are pre-legal, socially defined things like chairs, cars, and plots of land. Some of them are legally distinguishable interests in the same thing, like a life estate and remainder in the same plot of land.

Orlando emphasizes the existing modularity of property law by demonstrating how loosely coupled the multiple interests in the same piece of property truly are. The statement that a life estate must be followed by a remainder or reversion expresses a non-modularity of future interests: distinct modules can interact only in certain ways. But Orlando's title-tree node types are fully modular and orthogonal; they are not so restricted. The rule about life estates and remainders is a rule of how we talk about property interests, not a rule about how property interests function.

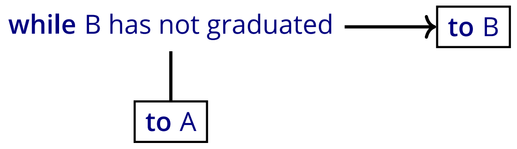
The same is true of the conveyance grammar. If one starts from the cumbersome common-law names, then tries to create them using appropriate granting language while respecting the rules about what can follow what, the result is a linguistic dumpster fire. But Orlando's conveyance grammar rules are simple and orthogonal; there are no long-distance dependencies between different parts of a conveyance. The only informa-

*Contracts: Boilerplate and Information Flow*, 104 MICH. L. REV. 1175 (2006). *See generally* Christopher S. Yoo, *Modularity Theory and Internet Regulation*, 2016 U. ILL. L. REV. 1 (detailed analytical breakdown of concept of modularity); BARBARA VAN SCHEWICK, INTERNET ARCHITECTURE AND INNOVATION (2012) (precise discussion of modularity); CARLISS Y. BALDWIN & KIM B. CLARK, DESIGN RULES: THE PWER OF MODULARITY (2000) (*locus classicus* of modularity theory).

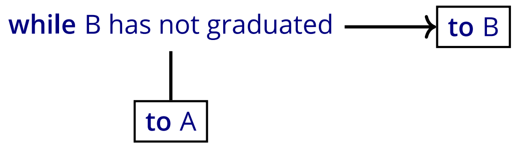
[126] A related literature analyzes the modularity of the constituent elements of contracts. *See* Smith, *supra* note 125; Gerding, *supra* note 30; Cathy Hwang, *Unbundled Bargains: Multi-Agreement Dealmaking in Complex Mergers and Acquisitions*, 164 U. PA. L. REV. 1403 (2015); Cathy Hwang & Matthew Jennejohn, *Deal Structure*, 113 NW. U. L. REV. 279 (2018).



(a) `to A until B graduates college, then to O` gives A a fee simple determinable



(b) `to A until B graduates college, then to B` gives A a fee simple with executory limitation



(c) `to A, but if B graduates college to B` gives A a fee simple subject to executory limitation

Figure 29: One interest with three different names.

tion that must be remembered during translation is the identity of the grantor, for classifying interests as being retained or non-retained, and for inserting implied reversions. The legal rules for parsing conveyances are about as modular as anything expressed by natural language can be; they have merely been obscured under accumulated layers of overly fussy description. Much of the confusion about the workings of the system of estates and future interests arises not because the substantive rules are complicated and arbitrary, but because the *naming* rules are complicated and arbitrary.

For example, compare the conveyances (1) `to A until B graduates college, then to O`, (2) `to A until B graduates college, then to B`, and (3) `to A, but if B graduates college to B`. As depicted in Figure 29, they generate structurally identical title trees. The only thing that varies is who takes possession after B graduates, and the specific language of the grant. And yet A's fee simple is ”determinable” in (1) because it is followed by an interest in the grantor, and "subject to an executory limitation" in (3) because it is followed by an interest in a transferee. The Restatement complicates things even further by insisting that A's fee simple in (2) is "with" an executory limitation because the limitation is stated in the grant creating A's interest, while it is "subject to" an executory limitation in (3) because the limitation is stated in a subsequent grant.[127] There is a useful terminological distinction between O's reversion in (1) and B's executory interest in (2) and (3), as executory interests are subject to the Rule Against Perpetuities and reversions are not. But there is nothing useful to be gained by describing A's functionally identical interest three different ways depending on the context.

## Conclusion

We have turned future interests into a programming language. Or perhaps they were one already. One interpretation of the notorious driness of the subject is that generations of courts and scholars had distilled its doctrines into a nearly pure form, one closer to logic than to experience. Since before the Restatement (First) was published, reformers have been calling for a dramatic simplification of the system of estates and future interests.[128] Maybe the task is smaller than it seems.

[127] *Compare* RESTATEMENT (FIRST) OF PROP. § 46 (1936) [hereinafter RESTATEMENT (FIRST)] ("subject to"), *with* RESTATEMENT (FIRST), *supra*, § 47 ("with").

[128] Myres S. McDougal, *Future Interests Restated: Tradition Versus Clarification and Reform*, 55 HARV. L. REV. 1077, 1115 (1941) ("To make a superb inventory of Augean stables is not to cleanse them."); J.J. Duleminier, *Contingent Remainders and Executory Interests: A Requiem for the Distinction*, 43 MINN. L. REV. 13 (1958); William F. Fratcher, *A Modest Proposal for Trimming the Claws of Legal Future Interests*, 21 DUKE L.J. 517 (1972); Lawrence W. Waggoner, *Reformulating the Structure of Estates: A Proposal for Legislative Action*, 85 HARV. L. REV. 729 (1972); Gerald Korngold, *For Unifying Servitudes and Defeasible Fees: Property Law's Functional Equivalents*, 66 TEX. L. REV. 533 (1987); Thomas P. Gallanis, *The Future of Future Interests*, 60 WASH. & LEE L. REV. 513 (2003); D. Benjamin Barros, *Toward a Model Law of Estates and Future Interests*, 66 WASH. & LEE L. REV. 3 (2009). In fairness, the Restatement (Third) made a start. *See* Lawrence W. Waggoner, *What's in the Third and Final Volume of the New Restatement of Property that Estate Planners Should Know About*, 38 ACTEC L.J. 23 (2012). And the Restatement (Fourth) is underway. *See* Thomas W. Merrill & Henry E. Smith,

Future interests are just the start. Formalizing legal rules as a programming language can clarify their conceptual structure in a way that other approaches do not. Elegant algorithms are often not just correct, but self-evidently correct, and the process of finding them "adds a strong dose of precision and rigor" to legal analysis.[129] Fresh insights await as more parts of law are subjected to this new type of scholarly scrutiny.

We believe that Orlando offers a firm theoretical foundation for future research in formalizing property law. Orlando's title trees can easily be extended with new node types—for example, perhaps with a `common` node to indicate concurrent ownership in a tenancy in common—without requiring any changes to existing node types. And Orlando's conveyance grammar can easily be extended with new rules to allow conveyances to create these new nodes. The following is just a partial list of topics that strike us as ripe for formalization in an extension of Orlando:

- The feudal system that preceded the one Orlando currently formalizes, with moving parts like seisin, subinfeudation and substitution, homage, and feudal incidents.
- Equitable interests, such as historical uses and modern trusts.
- Nonpossesory interests, such as easements, servitudes, and liens.
- Dower, curtesy, and spousal shares.
- Modern RAP reforms, such as wait-and-see.
- Priority among conflicting transfers and the effects of recording acts.
- Involuntary transfers, such as adverse possession.

And formalizing property law is just one small corner of what programming languages have to offer. Legal scholars Paul Ohm and Houman Shadab have argued that writing programs can be a form of legal scholarship.[130] Sometimes the best program for the job will be an interpreter for a new legal programming language. Just as legal scholars use the tools of critical

*Why Restate the Bundle? The Disintegration of the Restatement of Property*, 78 Brook. L. Rev. 681 (2014).

[129] *TAXMAN*, *supra* note 20, at 839.

[130] Ohm, *supra* note 49; Shadab, *supra* note 49.

race theory and economic theory to illuminate law, they can use the tools of programming-language theory too. Law and linguistics is an established subfield;[131] law and programming linguistics could be one, too. Most legal scholars will not work with programming languages, but some of them should. If they can learn to perform regressions and run collocation queries, they can write context-free grammars and operational semantics.

To quote the computer scientist Donald Knuth, "Science is what we understand well enough to explain to a computer. Art is everything else we do."[132] For centuries, future interests have been an arcane art. Now they are a science.

[131] *See, e.g.,* Jill C. Anderson, *Just Semantics: The Lost Readings of the Americans with Disabilities Act*, 117 YALE L.J. 992 (2007); Thomas R. Lee & Stephen C. Mouritsen, *Judging Ordinary Meaning*, 127 YALE L.J. 788 (2017); BRIAN G. SLOCUM, ORDINARY MEANING: A THEORY OF THE MOST FUNDAMENTAL PRINCIPLE OF LEGAL INTERPRETATION (2015).
[132] MARKO PETKOVSEK, HERBERT S. WILF & DORON ZEILBERGER, A=B vii (1997).

**Orlando Reference**

*conveyance* ⇒ *owner* `conveys` *grant*

*grant* ⇒ `to` *person quantum*  
*grant* ⇒ *grant limitation*  
*grant* ⇒ `if` *condition grant*  
*grant* ⇒ `if` *condition grant* `otherwise` *grant*  
*grant* ⇒ *grant* `but if` *condition grant*  
*grant* ⇒ *grant* `but if` *condition* … `reenter`  
*grant* ⇒ *grant* `then` *grant*  
*grant* ⇒ ( *grant* )

*quantum* ⇒ $\epsilon$  
*quantum* ⇒ `and` *pronoun* `heirs`  
*quantum* ⇒ `and the heirs of` *pronoun* `body`  
*quantum* ⇒ `for life`  
*quantum* ⇒ `for the life of` *person*  
*quantum* ⇒ `for` *n* `years`

*limitation* ⇒ `while` *condition*  
*limitation* ⇒ `until` *condition*

*person* ⇒ `O` | `A` | `B` | `C` | … | `Alice` | `Bob` | …  
*pronoun* ⇒ `her` | `his` | `hir` | `their` | `zir` | …

Figure 30: Conveyance grammar

$$[\![ \mathit{owner}\ \texttt{conveys}\ \mathit{grant} ]\!](t) = \quad t[([\![\mathit{grant}]\!]_{\mathit{owner}} \rightarrow \textbf{to}\ \mathit{owner})\ /\ \textbf{to}\ \mathit{owner}]$$

$$[\![\texttt{to}\ \mathit{person}\ \texttt{true}]\!]_o = \textbf{to}\ \mathit{person}$$
$$[\![\texttt{to}\ \mathit{person}\ \mathit{quantum}]\!]_o = \textbf{to}\ \mathit{person}\ \textbf{while}\ [\![\mathit{quantum}]\!]_{\mathit{person}}$$
$$[\![\mathit{grant}\ \mathit{limitation}]\!]_o = [\![\mathit{grant}]\!]_o\ \textbf{while}\ [\![\mathit{limitation}]\!]$$
$$[\![\texttt{if}\ \mathit{condition}\ \mathit{grant}]\!]_o = \textbf{if}\ [\![\mathit{condition}]\!]\ \textbf{then}\ [\![\mathit{grant}]\!]_o\ \textbf{else}\ \bot$$
$$[\![\texttt{if}\ \mathit{condition}\ \mathit{grant}_1\ \texttt{otherwise}\ \mathit{grant}_2]\!]_o = \quad \textbf{if}\ [\![\mathit{condition}]\!]\ \textbf{then}\ [\![\mathit{grant}_1]\!]_o\ \textbf{else}\ [\![\mathit{grant}_2]\!]_o$$
$$[\![\mathit{grant}_1\ \texttt{but if}\ \mathit{condition}\ \mathit{grant}_2]\!]_o = \quad (([\![\mathit{grant}_1]\!]_o \rightarrow \textbf{to}\ p)\ \textbf{while}\ [\![\mathit{condition}]\!]) \rightarrow [\![\mathit{grant}_2]\!]_o$$
$$[\![\mathit{grant}\ \texttt{but if}\ \mathit{condition}\ \ldots\ \texttt{reenter}]\!]_o = \quad ([\![\mathit{grant}_1]\!]_o\ \textbf{while}\ [\![\mathit{condition}]\!]\ \text{and}\ o\ \text{does not reenter}) \rightarrow \textbf{to}\ o$$
$$[\![\mathit{grant}\ \texttt{then}\ \mathit{grant}_2]\!]_o = [\![\mathit{grant}_1]\!]_o \rightarrow [\![\mathit{grant}_2]\!]_o$$
$$[\![(\ \mathit{grant}\ )]\!]_o = [\![\mathit{grant}]\!]_o$$

$$[\![\epsilon]\!]_p = \text{true}$$
$$[\![\texttt{and}\ \ldots\ \texttt{heirs}]\!]_p = \text{true}$$
$$[\![\texttt{and the heirs}\ \ldots\ \texttt{body}]\!]_p = p\ \text{has issue}$$
$$[\![\texttt{for life}]\!]_p = p\ \text{is alive}$$
$$[\![\texttt{for the life of}\ q]\!]_p = q\ \text{is alive}$$
$$[\![\texttt{for}\ n\ \texttt{years}]\!]_p = n\ \text{years have not yet passed}$$

$$[\![\texttt{while}\ \mathit{condition}]\!] = [\![\mathit{condition}]\!]$$
$$[\![\texttt{until}\ \mathit{condition}]\!] = \neg[\![\mathit{condition}]\!]$$

Figure 31: Translation function

$$\begin{aligned}
t &\Rightarrow \textsf{to}\ p\\
t &\Rightarrow \bot\\
t &\Rightarrow t_1\ \textsf{while}\ c\\
t &\Rightarrow \textsf{if}\ c\ \textsf{then}\ t_1\ \textsf{else}\ t_2\\
t &\Rightarrow t_1 \rightarrow t_2
\end{aligned}$$

Figure 32: Title tree grammar

$$\begin{aligned}
\delta(\textsf{to}\ p) &= \textsf{to}\ p\\
\delta(\bot) &= \bot\\
\delta(t\ \textsf{while}\ c) &= \begin{cases}\delta(t)\ \textsf{while}\ c & \text{if } \vDash c \text{ and } \delta(t) \neq \bot\\ \bot & \text{if } \nvDash c\\ \bot & \text{if } \delta(t) = \bot\end{cases}\\
\delta(\textsf{if}\ c\ \textsf{then}\ t_1\ \textsf{else}\ t_2) &= \begin{cases}\delta(t_1) & \text{if } \vDash c\\ \delta(t_2) & \text{if } \nvDash c\end{cases}\\
\delta(t_1 \rightarrow t_2) &= \begin{cases}\delta(t_1) \rightarrow t_2 & \text{if } \delta(t_1) \neq \bot\\ \delta(t_2) & \text{if } \delta(t_1) = \bot\end{cases}
\end{aligned}$$

Figure 33: Update function