1 Introduction

For reasons such as discussed in Chapter 20, computer programming languages offer only a flawed and limited simulacrum of "natural" Language (i.e., human, spoken language). This does not rule out how structural features of programming languages could potentially shed light on human linguistics (shortly I will present in greater detail where I think the most promising or prominent structural correlations exist), but the tenuous connections between human and computer languages writ large invite questions related to the nature of languages that might be intermediate between them — more complex and/or context-dependent than computer code, but also more computationally tractable than human language itself.

This chapter is motivated in part by how these sorts of questions could be applied to robotics, which is an area of concern for several of this book's chapters. One foundational question when engineering robots is how humans should communicate with them. If scientists' goals are to fabricate "life-like" robots, perhaps taking a vaguely human-like physical form, then it would be natural to envision robots equipped with sophisticated Natural Language Processing capabilities. Such an aspiration, however, could eventually run up against the outer limits of computational NLP, in the sense that any inanimate machine — even ones with sophisticated AI capabilities — may simply be incapable of parsing and responding appropriate to human speech on a truly realistic level. Of course, even today NLP is sufficiently advanced to be a very useful tool, but not to the point where people in most cases would experience actual communication with AI agents as believably life-like, or a credible substitute for the emotional, gestural, and contextual nuances of linguistic interactions between people (which are almost never "purely" linguistic).

I think these issues are still open-ended. Philosophically, it is a reasonable hypothesis to assume that most of the "human" qualities of conversation are driven by cognitive activity that might be simulated at least in theory, so a truly advanced robotic or AI agent could — again, at least in theory — asymptotically approach a state of being considered almost human-like in how it communicates. A counter-argument to this kind of analysis might be that language-processing sophistication is necessary but not sufficient for being an effective discursive partner: witness how people from different cultures can have trouble understanding one another, even if baseline semantic and syntactic impediments are removed. Sports fans from London and New York, say, somehow eavesdropping on each others' conversations, could very well fail to comprehend their respective dialogs despite the two groups speaking essentially the same language. Likewise for comparing discussions about US versus UK politics, or navigating Brooklyn or Camden Town like a local — inevitably, in short, there must be some shared cultural and lifestyle familiarity between conversants in order for reasonable human dialog to take place, because so much of the semantic and referential details of our speech remain unstated, an implicit part of the discursive background rather than explicitly invoked lexically or descriptively. No one in a typical conversational context would need clarification on how "Spurs" in the US probably refers to a basketball team but a football club in the UK, or "Soho" to a part of Westminster in London but a part of Lower Manhattan in New York. These are matters of shared life-experience, not of linguistic competence.

How, in this sense, could a robot "share" life-experiences with people? On the face of it, such questions should lead us to question whether realistic NLP is ever truly feasible, because even in a hypothetical world where a machine with human-level abilities of raw linguistic processing could be built, that machine would still not be "living" in a human world and absorbing local culture and dialect. Could all of our enculturated background knowledge be adequately simulated on a computer? What about the emotive and empathic dimensions of language, without which it can be difficult to distinguish, for instance, sarcasm from sincerity?

Short of some empirical disproof of mind-body reductionism — i.e., the belief that all of our feelings and experiences ultimately derive from neurological phenomena, which are not known to have any physical properties that are truly beyond the scope of computational approximation — we certainly cannot rule out the possibility that truly human-like AI is theoretically possible. But absence of impossibility is a weak argument for something being realistic and feasible in practice, and cultural/interpersonal issues demonstrate that the hurdles for NLP are not only linguistic in nature.

In short, even if we cannot be certain that realistically human-like AI (at least in terms of linguistic competence) is *not* possible, there seems to be no reasonable argument to guarantee that it *is* possible, so any technological project which depends on (something approximating) human-like NLP is currently on shaky grounds. At the same time, we *should* aspire to robot/human communications which are more user-friendly than artificial constructs like programming languages. Note that this is an issue pertaining to robots' "users" — people who at some level guide a robot's behavior in the course of its assigned task, as compared to developers who engineer their capabilities in the first place. Robots' on-board operating systems can be programmed in a variety of languages, as is true for other embedded systems, and low-level computer code provides the minimal functionality allowing robots to operate (any physical movements or audio/visual processing of their surrounding environment). When a robot is performing a task on behalf of a person, however, we can assume that in most contexts the user would not be communicating with the robot through such low-level code, but would instead give instructions more conceptually. For a robot to move a box, say, the person should be able to designate the box in question and its destination, allowing the robot's low-level machinery to calculate the object's pose and location, and its own physical configuration and movements appropriate for that goal.

Here, then, is a good example of the "intermediate" language question I alluded to before: what forms of communication exist that are higher-level than artificial languages such as computer code, but simpler and more computationally deterministic than natural language? This actually encompasses two topics: how such "mid-level" languages should be *designed* (their semantic and syntactic properties — whether their vocabulary and/or grammar is based on actual human languages, albeit at a rudimentary level, or something more endemic to a robot's assigned tasks), first, and, second, how to translate expressions in such mid-languages to low-level operations which can be programmed as robots' baseline capabilities.

I will touch on these issues later in the chapter, but here I will also be focusing on the theme of language in that gap between humans and computers — what resources can help us think through

the nature of language outside the prototypes of mechanical computer languages and holistic human language? I argued at the start of Chapter 20 that calling implemented coding systems such as C++ or JavaScript "languages" is misleading, and in some ways the science of building and using programming languages (sometimes called Software Language Engineering) has very little connection to linguistics proper. Note, however, that some sophisticated lines of linguistic research are driven or inspired at least part by concepts that emerge, or are formalized, in the programming-language context ([2], [4], or [1], for instance). So there are at least some technical formations in software-language implementation which may arguably shed light on how people understand our own human languages (some of these I reviewed in Chapter 20, and will revisit later in this chapter as well). With that said, though, the kind of artificial languages employed to program computers and machines have at best only sporadic and limited structures that shed light on natural language, and as such are only an imperfect model for the kind of intermediate communication system which we might envision possessed by robots and other "Industry 4.0" machines.

I argued in Chapter 23 that intersubjective awareness serves as a central motivation that underlies other foundational dimensions of language — specifically, situational understanding and conceptual plasticity. Interpersonal understanding allows us to structure situational schemas around people (or, more abstractly, around epistemic agents and situational actors) so as to form coherent logical models of situations, and these models in turn are built up from concepts. Concepts, for their part, exist in more generic and more specific guises, and the manner in which broad concepts become particularly applicable to narrower situations provides one facet of situations' internal logic. Via this sort of architecture, intersubjectivity evolves, through a kind of dialectic, into a roughly effective logical and co-operative picture of our surroundings, which ensures the prelinguistic synergy of our communication.

Controversies over how to interpret results in Artificial Intelligence perhaps lend support to this point: enacting Computer Vision or processing natural language via statistical inferences (even if augmented by Convolutional Neural Networks or Machine Learning) may or may not qualify as legitimate "understanding." Computers' sometimes human-like performance in translation, or (for instance) image-tagging, could give the impression that AI modules are progressively approximating human reasoning. However, so-called "adversarial" examination of AI programs has led some researchers to question such assumptions. These analyses are able to produce images, for instance, that "trick" Computer Vision technology into radically incorrect or nonsensical results — tagging two almost identical images into completely different categories, one accurate and one not (e.g., a school bus and an ostrich), or applying the same tag to a realistic photograph and nonsensival picture looking like random noise [46, pages 110-113]. Similar effects have been produced against NLP engines [46, pages 211, 227-231].

The issue here is not merely occasional inaccuracies, but that AI tools can be *so* wrong on select kinds of input that we have to question whether the "intelligent behavior" they usually exhibit truly approximates what we would think of as (human-like) intelligence. After all, people too can mislabel images and misread sentences, but we would never confidently assert that incoherent mumblings or "nonrepresentational" graphics (with appearance akin to TV-screen "snow") were normal sentences or pictures of Milla Jovovich. One way to express these concerns is a doubt that AI agents — even if statistically accurate most of the time — truly share *situational understanding* with us.

Removing intersubjectivity (and subjective awareness) from language and communication does not necessarily disassemble the situational and conceptual formations which emerge on its basis, but they leave open-ended the process of how linguistic competence can take hold *without* interpersonal, collaborative foundations. My intent last chapter was not to argue that the triad of intersubjectivity, situations, and conceptualizations are constitutive of language per se, but rather of the prelinguistic firmament in which language can take root. Linguistic constructions encode situational details, but how they relate to contextual schemata remains a further question I did not address in any detail.

So, what exactly do we accomplish via speech-acts insofar as linguistic activity adds a further layer to our preliminary cognitions and co-operation? I propose a "foundations of language" story which says in effect that our common situations and "joint attention" are *pre*-linguistic, while language proper concerns *changes* to our joint environments (whether past, present/ongoing, or futurally planned/contemplated). This perspective fits nicely with "verb-centric" theories of semantics and grammar, according to which sentences' parse-graphs are organized around verbs, with nouns (and noun-phrases) slotting in verb-details and thematic roles expressed by "theta" relations (such as verbs' subject and object components) and by case/declension (where nominals in the locative, instrumentive, benefactive, and so forth add specificity — how, where or to where, by whom or for whom — to their verb's profiled event, process, or action) [43, e.g., page 39], [41], [56], [30], [7], [8], [9]. It is true that almost any theory of language will focus on verbs because verb-plus-noun combinations (VP+NP) are considered to be the minimal requirements of a complete phrase. But some paradigms reinforce the verb-centric perspective still further, with the idea that verb-meanings form the central organizing motif through which sentences are cognized — and that verbs dominate syntax in that all grammar elements (including categories such as nouns, adjectives, prepositions, or conjunctions, plus morphosyntactic marking rules/conventions) play roles ultimately determined by how they modify and add information to phrase- or sentence-level verbs.

Let's assume that in order to exchange speech-acts, people must have situation/environment models that are sufficiently in sync. Because this agreement *precedes* anyone saying anything in particular, we can assume that language itself is not usually *about* this shared awareness, albeit we sometimes verbally correlate our mutual understanding (along the lines of "do you see ...?" or "did you know that ...?"). Normally the role of language is to address what *deviates* from common understanding (or could cause respective models to become misaligned). Someone might propose an action that would alter the current situation (e.g., "Let's go to the park"), or inform others of a change they may not know about.

As a rule of thumb, then, we should look at a sentence and consider what details are expressed there that apprst to propose, observe, or describe some *deviation* in shared understanding to which the speaker wants to call attention (or to request/recommended). Sentences' meaning would then be structurally marked by the con-

trast or gap between existing shared situation/environment models and some belief idea held by the speaker and (potentially) no others. Insofar as verbs profile the difference between one environment-state and another, it makes sense that verbs would then be the central organizing anchors for syntax and semantics.[1]

This overall approach to sentence meaning highlights common understanding as a linguistic precondition, because it construes meaning as an effect of difference where that prior understanding is one differend — we measure significance *against* the common model, with the meaning proper consolidated in how a proposal or observation departs *from* that model. Linguistic competence would thereby be a two-part phenomenon which includes both sharing with others pre-linguistic joint understanding *and* being able to communicatively propose or effectuate changes to shared situation/environment models (including by changing the situation itself). In accord with my discussion in Chapter 23, I contend that we should credit dogs such as Stella with credible human-like language so long as we feel confident that they have cognitive resources to participate in our language at both of these levels. Data I summarized there certainly suggests both that Stella shares mental models with her caregivers, with some understanding of their context-specific beliefs, *and also* that she understands (even if at a rudimentary level) how she can *alter* others' beliefs or *suggest* changes to their shared situation by utilizing language (in Stella's case, by sounding out phrases via her talk buttons).

Because language proper expresses *deviations* from shared models — not withstanding how common understanding is a prerequisite — there must be other significatory facets to language that we can study via syntax and semantics. This, I would argue, qualifies as the origin of structures that can be applied to phenomena such as computer languages, which have some overlap with syntax or semantics in natural language proper but (for reasons I have outlined) do not, I believe, reflect the preliminary phenomena of joint attention and intersubjectivity. This chapter will consider some of these structures from a linguistic perspective, then pivot to discussing notions of "semantics" which may actually be appropriate for artificial settings such as robotics and software-programming.

2 Semantics and Situational Change

Most of the time, language reflects our pragmatic concerns. Situations provision a backdrop for goal-directed activity, and the circumstances obtaining at any moment serve as both a parameter on and, to the degree that we act deliberately, a byproduct of our actions. If we ride a bus to the dog park, we are on the bus *because* we have decided on that course of action for the afternoon, but the bus is also a constraint; it creates situations for us that we occupy rather than explicitly architect. If the bus is stuck in traffic, we have no way to get there faster. Rain keeps us indoors; cars cannot drive except on roads; locked doors must be opened with keys. The affordances of constructed objects are also djinns producing states of affairs that we must adapt to. In this sense our situational awareness, while partly present-tense, is always too projecting forward, protaining the immediate future, subtly modulating plans in response to the not-fully-anticipated turns that situations take.

Ultimately, I would argue, it is this enactive-protaining register that forms the predominant mode of linguistic semantics. What we *mean* is usually reflective of how we anticipate our actions in the environing context of our situations to proceed, moving forward, in the immediate future. Linguistic utterances are, in short, canonically, formulaic expressions of temporal intentions in this sense, i.e., "intentions" in the sense of intending-to-act or intending-to-do. Of course, we do not spend all our linguistic lives simply asserting plans, but such pragmatic intentionality forms the substratum of language proper, I would argue. Such intentionality is still an underlying semantic paradigm when we switch from raw expository statements to questions, conversation turns, prompts for others' opinions, and in general the techniques we use to form collaborative intentions via dialog.

In sum, then, language departs from underlying cognitive schemata largely through the gap between merely experiencing present situations and co-ordinating near-future actions that follow from present-moment situational episodes. Stella (referring back to Chapter 23) is accordingly working in a linguistic register when she uses words to project forward in time, via articulations that incorporate the present moment but are not wholly contained in it ("eat park outside", "want park/beach," "come bye outside", "play outside" after Christina's "what now," and so forth). The complexity of linguistic constructions, of course, will mirror the degree of detail and breadth of planning spanned by the current discourse. Stella's relatively simple phrases stand at one extreme; something like a political party's platform paper, sketching out years' worth of legislation and programs (however realistic or not) at the other. But it is reasonable to argue that the crucial dimension of language proper is not syntactic complexity itself; instead, complexity of form is a metaphenomenon reflecting the intricacy of the anticipations which language communicates. Simple plans are no less vehicles of language than elaborate ones.

On this account, it is impossible to theorize linguistic meaning without describing how hearers receive speakers' statements as asserting or formulating near-future plans. Addressees must understand how word-meanings, in the context of their proper grammatic forms, translate to anticipated actions. Such meaning depends heavily on prelinguistic knowledge, and on all parties to a conversation having succinct correlations in their respective beliefs to co-operate productively. If two people talk about taking a bus to the park, let's say, they need a prior understanding of what that would entail, such as the pragmatics of boarding a bus, where to get off, and so forth. This embodied knowledge is not a predicate that could be built via syntax and semantics alone, like a journalistic reportage ("today the new Prime Minister visited the King") stating a (relatively self-contained) fact. We might indeed say that the meaning of that Prime-Minister sentence is wholly expressed through its propositional content, but a more quotidian speech-act such as "let's take the bus to the park" is less illocutionary, its meaning deriving less from any proposition mimicked by its construction and more from how it solicits prelinguistic knowledge — prelinguistic "scripts," one might say — that would enable the actions proposed.

[1] Although "verb-centric" paradigms are tangentially related to Davidsonian "event semantics," I am thinking of verbs in the most general sense of *profiling* states, relations, or processes as well as events, following Langacker in particular [34, chapter 4], [38, page 24], [37, e.g., pages 21-25], [36]. Also, for reasons sketched below I am wary of semantic structures presented as *logical models* of sentences themselves, rather than as patterns organizing cognitive *responses* that may be *triggered* by linguistic constructions but are not prefigured or reciprocated within them.

From an enactive/pragmatist perspective, we (as language users) have a substantial repertoire of "scripts" which represent learned know-how, enabling our competent achievement of concrete goals (riding to the park, and so forth). Such scripts are not propositions, nor are they facts that language can convey in propositional form (like "The Prime Minister visited the King"). Rather, they are competences which we have at some level abstracted and represented such that they could be referenced indirectly through language. There is nothing in the phrase "ride the bus" that conveys the propriocentric patterns of climbing on board (and the geospatial acuity to know where to get off). Nonetheless, expressions like "take the bus" serve as proxies to these pragmatic packages. Linguistic meaning, in short, often takes the form of nominating various enactive-prelinguistic scripts with which an addressee (by assumption) is acquainted.

Not all expressions work by "proxy" to this degree, of course (as compared for instance to describing facts via logical constructions), but insofar as language such as "ride the bus" depends for its semantics on already-familiar enactive "scripts," we cannot treat meaning as something *internal* to language, something structurally replicated in linguistic form. I have attempted more detailed analyses to motivate what I see as an essential deviation between language and logic (see footnote 2), a line of argumentation which I won't pursue here for long, but a thumbnail version is that most language does not work the way "logical positivist" (among others) paradigms have claimed, where language serves in effect as an informal version of the kind of predicate compositionality modeled more rigorously by formal logic. Sometimes it does: "(Rishi) Sunak visited (King) Charles" individuates two actors in a logical relation and names this relation directly, all through straightforward lexical and referential means. But most linguistic constructions do not work by reciprocating propositional constructions.

A more typical speech-act is something like "let's take the bus to the park," whose apparent meaning does not have a direct propositional content in the first place (the intended idea is more involved than just, say, the expected fact of them soon *being at the park*) and, to the degree that predicates are implicit in the thought conveyed, the sentence does not work by structurally embodying them. Instead, the sentence's effect is, more, to inspire in the listeners imaginings of, and (to the degree that they endorse the plan) anticipations of the practical and experiential episodes that would comprise, in sequence, "going (by bus) to the park". The sentence works by invoking a script which the hearer (presumably) knows reasonably well.

Arguments to the effect that language "invokes scripts" more than "models propositions" are consistent, I think, with various theories of language that purport to offer an alternative to "analytic" philosophy of language arguably over-invested in language being a kind of symbolic (or symbolized) logic. These (countering-Analytic-paradigm) approaches would include Cognitive Grammar and Construction Grammar, probably alongside "embodied" notions in Cognitive Linguistics (George Lakoff and Mark Johnson, for example), or Jean Petitot/René Doursat "morphodynamics," speech-act or conversation-oriented models in the tradition of Austin, Grice, and Searle, or ethnolinguistic (and "eco-"linguistic) approaches to language as an anthropological phenomenon more than a "system" to be structurally analyzed [35], [35], [6], [26], [27], [32], [31], [69], [70], [52], [53], [28], [21], [24], [64], [63]. To this list I would add Orlin Vakarelov's "Interface Theory of Meaning," [65], [66] which I contend may be applied to artificial (programming) languages as well as natural language. In the formal-language context, the value of such "interface theories," as I intimate them, involves the formal-semantic paradigm (manifest to some degree in computer science, for example in the design of the "Semantic Web") according to which programming-language semantics should be evaluated in logical (and set-theoretic) terms. That is to say, expressions in programming languages are (within the paradigm being disputed) logical elements such as predicates/relations, sets, and set-members, and by sufficiently intricate aggregates of such logical building-blocks we can form computational simulacra of empirical objects. Logically constructing propositions about such objects (or their states and properties) enables objects to form the "semantics" of programming and database/data-modeling languages, so that such languages' semantics is constituted by the logical approximations relating computational models to their presumptive referents.

Against such a "logical" paradigm, I argued in Chapter 20 for a "procedural" perspective, in which the semantics of computational languages are constituted by the collection of procedures that can be invoked through (any given) language (in any given context). This yields what has been called a "procedural semantics" and applied both to linguistics and robotics [23], [13], [62] [60]. Computational (or, respectively, by analogy, cognitive) *procedures* semantically implicate empirical objects (by presenting information about them, showing them pictorially, simulating then virtually, and so forth), but we should not reason as if the structures of computer code somehow "describe" the objects in question. Instead, computer code engineers a space of procedural responses to user actions in which object-representation, to the degree that it is perceived as such by users, is an "emergent property" of the overall computational system. Outside the indeterminacy of such "emergence," the actual "semantics" of computer languages should be sought in the procedures that create representations of objects for a user's benefit.

Of course, computer code exists on multiple levels, and while we might agree that higher-level code depends for its meaning on procedures that are bound to code-symbols, those procedures are implemented via lower-level code, and so on down the compiler stack. Eventually we reach a machine language whose semantics is logic gates and memory addresses rather than procedure-calls as such. However, these lowest-level layers — where computations are actually manifest on a physical level — have less resemblance to real-world objects than computer code at higher levels. The layered nature of code-implementation motivates a model wherein any "natural" semantics for computational systems is an emergent property, not a logical construction. I defer to Vakarelov's "interface theory" in both contexts: the overt meaning of a linguistic expression (a speech-act in natural language, or a code-statement in programming) is an "interface" to some underlying script/procedure which resides not at the level of the explicit language, but on a deeper or hidden level (the repertoire of pragmatic scrips held by a person using language; the space of compiled routines known to a programming-language runtime and available to be called from application code). Natural language invokes "cognitive procedures" (e.g., "imagine riding the bus") much as high-level code notates instructions to call low-level functions.

I contend, therefore, that something like an "interface theory" yields a useful paradigm with payoffs in both natural language and Software Language Engineering. Granted, philosophical paradigms are not especially consequential in the latter domain unless they imply some specific tactic or norm for the design and/or implementation of coding languages (their compilers, static-analyzers, and/or runtimes). It may not be obvious how argumentation that the semantics of computers is basically one of "procedures" and not "logic" practically affects any of these latter topics. In earlier chapters I presented analyses hoping to show that there is at least some practical consequence that might compel new language-engineering ideas, for example in more rigorously formalizing non-set-theoretic type theories and potentially new strategies for code analysis (alluded to via "constructor channels" and my overall suggestions to the effect that we should formulate compiler design and code-verification via structures that can be translated to query-evaluations over hypergraphs). This is, however, I concede, a fairly narrow domain of practical application. I have not clarified how detecting a concordance between "interface theories" in natural and computer languages yields any substantive insights in either context. The following, as such, are brief comments addressing these questions.

2.1 The (provisional) semantics of syntactic disambiguation

The central idea of "interface" theories of meaning, as I see them, is that linguistic structures do not (at least fully) encode meanings in full detail, but rather defer to extra-linguistic cognitive processes — language proper defines parameters and specifications on such extra-linguistic faculties, but does not logically articulate, at least in full detail, whatever mental operations substantiate cognitive processes themselves. To be rigorous — given that *reciprocate* usually connotes something coming *after the fact* — we can imagine a variation on the reciprocation theme wherein the idea of a structural alignment, a systematized lining-up of structural elements, is lifted from the "reciprocate" notion but switched to that which is temporally prior, like an anticipation or foreshadowing. According to an implicit linguistic paradigm — probably shared by many branches of "Analytic" philosophy as well — language produces logically organized meanings because elements in langauge are patterned, during the course of enunciation, to model (in the sense of *illustrate* or *prototype*) the logical form of ideas which then get decoded by addressees. The prior logical structure then guides addressees to assemble a notional understanding of expressions by reconstructing their predicate outlines. Despite this temporal ordering, though, it seems best to say that the expression-structure "reciprocates" the later reconstruction — on this account — because addressees are grasping the idea that is anterior to and contextualizes the expression as a communicative artifact. The goal is to achieve a cognitive synergy between multiple parties' thought-processes; the linguistic form is merely a device to achieve that telos.

This is, at least, one way to gloss the "logistic" paradigm which some writers have critiqued ([33, Chapter 25], say). Cognitive Grammar and (at least as presented by Vakarelov) Interface Theories present an alternative paradigm; my gloss on this latter perspective is that whatever actually reciprocates logical form emerges from cognitive processing, not from linguistic structure (whether shallow or "deep") itself. The form of communicative expression has *some* correlation with logical structures, but this is not a mimicry in the sense of reciprocation or organizational alignment; instead, linguistic elements are patterned so that their structural relations, along with semantic and lexical cues themselves, serve as "prompts" or instigators molding processes of cognitive reception in different directions, so that the telic logical form emerges at the end of a cognitive arc. The resemblance of linguistic form to propositional structure is more like an embryo to a newborn than a closed term in symbolic logic to the proposition it encapsulates.

For lack of a better word, we might fall back on something like "temporally inverted reciprocation" to capture the notion that sentence signify propositional content by modeling its logical form (more or less directly — rather than prompting addressees to infer that logic via extra-linguistic cognition). There is no room here to go into detail as to what sorts of cognitive processes I have in mind, or linguistic evidence that I think argues for this perspective. A more expansive empirical and speculative defense of such arguments forms the thesis of a series of articles deposited as part of, and an analysis of, data sets published alongside [42] (I'll reference the paper-series in a footnote rather than citing them, because the supplemental material was developed as part of the data set, not submitted as journal articles)[2] — with a mashup of computer code, empirical data, and Cognitive-Grammar-style analyses, trying to achieve methodological plurality like I sketched in Chapter 23. The overarching theme here is a critique of "logicism" or symbolic-logic encoding of linguistic expression/signification, which of course rests in the shadow of similar counter-paradigm projects within the Cognitive Linguistic canon, represented on the philosophical side by Lakoff and Johnson in particular, and from the linguistic side by, e.g., Cognitive Construction Grammar (Langacker's Cognitive Grammar arguably being intermediary, part formal-linguistic and part-philosophical).

I claim however that these extant analyses intending to supplant "logistic" philosophy of language are still incomplete, and there is room to solicit more empirical data, more correspondences with formal and computational linguistic models, and more extensive investigation of cognitive processes *outside of* logicomathematic propositional models. That is to say, the Cognitive-Grammatic counter-arguments should be more thoroughly integrated with formalizable counter-theories of linguistic structure, perhaps from the perspective of "interface theories" and/or generalized Neo-Davidsonian event-semantics (e.g., [44], [25], [19], [20] [11], [10], [55]). Again, though, alas, the analyses and supporting data I have in mind are outside the scope of the current discussion. Here, I will focus on the narrow band within this territory that fits the points of contact between natural human discourse and artificial (e.g., programming) languages.

Having argued above that shared understanding is *pre*-linguistic, I would add that cognitive processing is *also* outside language, so that language proper is mostly a link or inter-connection from one to the other: syntax and semantics *starts with* joint situation/environment models and compels each conversant to supply new cognitions,

[2] From "Naturalizing Phenomenology" to Formalizing Cognitive Linguistics (I): Cognitive Transform Grammar.
From "Naturalizing Phenomenology" to Formalizing Cognitive Linguistics (II): Grounding and Center/Peripheral Relations.
From "Naturalizing Phenomenology" to Formalizing Cognitive Linguistics (III): Externalism and the Interface Theory of Meaning.

somehow altering or re-envision these surrounding, with language being the conduit between joint attention and a subsequent re-staging of the environment, both of which are predominantly *extra*-linguistic. Linguistic utterances then become figured as intermediate constructions that bridge but largely remain separate from these cognitive subsystems.

One potential counter-argument to this formulation is that fully constructing a linguistic expression (e.g., a sentence), so that it may serve such intermediary role, would therefore appear to be a mental operation outside of actual meaning-comprehension, because on this theory *extra*-linguistic cognitive processes flesh out meaning proper. This suggests that *as* linguistic, or during the "processing stage" where expressions are *only* linguistic artifacts (not yet subject to extra-linguistic cognitive scrutiny), expressions do not "have" meaning *per se*, and so are in effect meaningless. But, if so, how can we have coherent linguistic expressions in the first place? After all, what differentiates valid utterances from (logical or grammatical) nonsense is (apparently) that proper expressions are meaningful in some sense. For example, properly parsing a sentence would appear to be a necessary step toward aggregating linguistic content to serve as material for extra-linguistic cognitions. If there is no preliminary meaning, how can we actually distinguish correct parses than ones which yield gibberish, even if they satisfy certain criteria (such as pairing up words based on part of speech)? How can we rule out grammatically plausible but logically nonsensical parses, or filter interpretations that are syntactically reasonable but depend on mapping words to lexical options that are contextually unlikely?

In short, a further complication with regards to "interface" semantics, as I see it, concerns the role of meaning-anticipation in syntactic parsing. An idealized view of language might hold that we perceive parse-trees first, and fill in semantic content later; i.e., that parsing and lexical comprehension are two distinct processing stages. This might be how one would design an artificial language, but in practice there seems to be at least a circling between the two processes, where appeals to semantic coherence help resolve syntactic ambiguities. Assuming, with link and dependency grammars, that parse-structures can be modeled as graphs, most sentences have many candidate graphs which are not ruled out by syntactic desiderata alone (such as parts of speech). Barring outright "eats shoots and leaves" style ambiguity, only one parse-structure will typically emerge as a the most sensible reading *given* the resulting meaning, but assuming that a full semantic interpretation functions as a litmus test for rejecting less-likely parses would violate the model according to which we mentally form a parse-graph *first* and *then* deduce its intended meaning, by filling in the words' lexical (and, as needed, referential/anaphoric) content.

Of course, a simplistic way to address this problem is to envision our minds subconsciously testing many different parse-graphs, potentially "reading" thousands of different interpretations of a sentence — giving each a fully-specific meaning if possible — and only after all this work scoring one reading as the most likely. Such an explanation, which by stipulation we could never consciously verify anyhow, is not very satisfying (barring strong neurolinguistic evidence that processing really does branch out in parallel to this extent), because it basically just papers over an apparent theoretical gap by vague appeals to unknowable cognitive machinery. A more compelling analysis would provide some model of how the parse-graph space could be pruned *without* full semantic reconstruction, perhaps allowing *partial*, but noticeably limited, appeals to sentence-constituents' lexical (and semantic/referential/anaphoric) interpretations. Dependency grammar, for example, provides empirical data indicating how metrics of word-distance and "edge crossings" factor in to parses being more or less likely [45], [22], [14], [15], [16], [68], but this is still only a provisional analyses — many sentences' actual parse-graphs have configurations that would be deemed improbable based on such metrics in isolation, and even when they prune the parse-space correctly there can still be quite a few reasonable readings that must be further disambiguated.

I will not present empirical data supporting these proposals, but in light of my arguments in Chapter 20 I'd like to summarize at least the rudiments of a theory to further refine our models of parse-space. In general we seek to find criteria that would eliminate candidate graphs by considerations *other than* explicit semantic meaning, because with such restrictions in effect one can "quickly" narrow the scope of parse-structures that are legitimate contenders to be "correct" readings of each sentence. Part-of-speech data, for example, can play such an eliminative role, since only in specific contexts can a pair of nouns, say, or a pair of verbs, enter into head/dependent relationships. In practice, however, part of speech is fluid; quite a few constructions implicitly treat nouns as adjectives alongside other nouns (like "pet" in "pet store"), or treat nouns as verbs (e.g., "to primary" in politics), nouns-as-verbs-as-adjectives ("iced tea"), etc. Graph-pruning needs to ensure that valid part-of-speech migrations are not misclassified, which again begs the question of how to know what interpretations to reject *without* fully realizing the potential meanings that could be attributed to such interpretations.

A minimal criterion for parse-structures being acceptable is that they present sentences which build up to some (relatively) complete idea, although this still needs to account for partial expressions that addressees mentally complete (imagine someone holding out a glass and saying "could you please?," implying without stating a proposition related to someone filling the glass). This appears to be a fairly rigid system within language, which can without oversimplification be modeled via mathematical formalisms, such as Lambda Calculus. Actually, the role of Lambda Calculus in this context would lean toward type theory — even in untyped calculi there is a provisional distinction between functions and their arguments, and type-checking can be progressively refined as our type-models become more granular. So the essential detail, represented in Lambda Calculus via "beta"-reduction, would be how type-tuples under the influence of functional types collapse to single types, and iteratively such reduction can occur over multiple stages (for sub-expressions), yielding, in the linguistic context, the stipulation that (complete) sentences have "propositional" types.

In chapter 21 I argued that lambda calculi and their variations (Sigma Calculus, calculi with exceptions, and so forth) can be represented via "channels" and encoded through hypergraphs; this represents a small extension to the concept of type-reduction, because reductions are type-checked across multiple channels. Aside from that, we can further analyze to what extent semantic information can be encoded in type-systems; in the aforementioned [42] data-set analysis for example I distinguished between different levels of type-granularity ("macro-types," "meso-types," and "micro-

types"), with the idea that types of "intermediate" granularity could provide a mechanism for parse-pruning which rejects semantically implausible graphs on bases that are still heuristic and simpler than full-scale semantic interpretation.

Elaboration of my attempted extensions along these lines — via channels and meso-types — is outside the scope of the current discussion; I merely want to point out that type-theoretic approach may offer rigorous models of parse-pruning and that the effectiveness of pruning "algorithms" increases to the degree that type systems become finer-grained. This is true both for natural and programming languages, as evidenced by how strongly-typed languages support more rigorous modes of static analysis. Type systems which encompass "dependent types," typestates, and/or detailed "type-class" extensions progress further into the realm of encoding semantic information via type-attributions; these seem like good candidates for emulating how humans create provisional semantic models of sentences without fully completing all lexical/semantic/referential/anaphoric details. However fine- or coarse-grained our recognized type system, the key structural claim is that type-reductions cascade from terminal nodes to root nodes in tree-like (or directed-acyclic) subgraphs, and that this represents one universal constraint through which candidate parses can be eliminated.

Alongside such type-theoretic models, I also believe that interword relations along the lines investigated by Link Grammar supply a different and orthogonal class of constraints, because sentences appear to have predictable regularities in terms of contexts where specific forms of interword-links are permissible. In [48, chapters 6 and 9], I argued (inspired by work such as that of Lucas Champollion, as well as more generically from Cognitive Grammar and Link Grammar [57], [58], [47], [67] overall) that the most important framework for modeling pair-relations as parse-criteria is that of "thematic relations," focusing on the theta-roles and grammar-cases that link verbs to their profiled constituents (subjects and objects, and then via further detailing the various locative, benefactive, instrumentive, and etc. case-forms). Each pairing between a verb and its thematically related associated words or phrases adds a level of detail to the state or event profiled by the verb, as this is postulated by the speaker. The total collection of a verb's thematic relations represents the verb's state/event within an epistemic packages, mediated by the speaker's propositional and perspectival attitudes. Parse-graphs only make sense if thematic relations sum together in ways that are epistemically coherent. This, then, presents an additional layer of criteria that can be imposed on parse-spaces to distinguish credible from dubious parse-candidates. I defer to work such as Langacker [40], [39], [34, chapter 9] for actual descriptions of "epistemic coherence" (or see [3], [18], [12], [49], [50], [51], [29], [5], [59], [61], with respect to speaker-relative grounding and epistemic contingencies)[3]; here I will simply say that certain lineups of thematic relations are more coherent than others, and this can be grounds for accepting or rejecting candidate parse-graphs.

In sum, then, I propose that both type-reduction and thematic relations are systems that introduce parse-acceptability criteria, and both systems can operate at once over possible parse-graphs to eliminate spurious candidates (while still remaining uncommitted to full semantic realizations). Moreover, I believe these two systems are quite apparent in both natural and programming languages. Deciding amongst competing parse-graphs is not necessarily an issue for compilers in the latter case, because most computer dialects do not accept any sort of syntactic ambiguity comparable to natural language; the parse-structures of code-expressions should be fully determinate from rules that can be enforced mechanistically with compile-time information. However, parse-graphs for computer code only indicate how arguments are connected to procedures; there can still be ambiguity in the relation of procedure-symbols to the specific implementations which are intended to be called at the relevant code-points at runtime. In this context compilers and/or runtimes have to apply heuristic tests for symbol/procedural mapping, often involving "topological sort" algorithms and metrics grading type-coercions. Here a type-theoretic infrastructure plays a pruning role analogous to parse-graph multiplicities in natural language. Also, procedures' pre- and post-conditions serve as potential criteria on graph-edges structurally equivalent to thematic relations in natural language: consider the idea of overloading a procedure operating on two lists (or vectors, arrays, stacks, queues, etc.) by implementation-bodies differentiated via typestates: two empty lists, one empty list, two same-sized lists, and so forth.

With respect to "interface theories of meaning," the type-reduction/thematic-relation systems offer a plausible model of how parse-graphs can be selected even while parsing is understood to be functionally prior to cognitive "scripts" that yield the full-scale semantics for linguistic expressions. Although it would take more work to make this proposal rigorous, I believe that the co-existing structures of type-reduction and thematic-relations serve as structural criteria in both natural and programming languages, yielding constraints on syntactic and/or semantic processing which allow some decisions to be made — parse-graph pruning in the former case, function-symbol resolution in the latter — that foreshadow but do not depend on complete semantic information.

2.2 From Natural to Computer Languages

While similarities between programming and natural languages are in general pretty superficial — to reason otherwise would minimize the subtlety and indeterminacy of human speech, which is complete different than the mechanistic engineering of software code — I think these systems in particular (type-reduction and thematic-relations) are structurally complex, relatively speaking, and operationally non-trivial, revealing interesting overlaps between these two forms of language notwithstanding their ontological contrasts.

If these are shared structures across both natural and computer language, one might reason that similar formations would likewise apply to "robot" languages, to the degree that — as I hypothesized above — an optimal robot language would lie at some intermediate distance between human speech context-sensitivity/indeterminism and computer code's mechanism.

[3] With the caveat that I believe notions such as grounding and epistemics should be read more broadly than *merely*, say, semantics of modal verbs, or figure/ground relations — that indeed the entirety of a linguistic expression is patterned around speakers signaling through semantic and syntactic means their particular epistemic comportment vis-à-vis shared situations. Even if the purpose of a discourse-fragment is not to thematize speaker-relative epistemics directly (*I think that.... Surely/Probably...*, *It looks to me like...*) we cannot help but reconstruct the particularities of our perspectives through linguistic form and communicative pragmatics, because the overlap and possible contrasts between our situational beliefs and other people's are intrinsic to their cognitive reception of our speech-acts. Such cross-epistemic processing, I claim, factors directly into syntactic/semantic and serves as the overarching architecture of linguistic form to the degree that it does not just logistically simulate propositional structure; again the data-set article-trio linked in the prior footnote develops these claim in more detail.

If it is true that the proper semantics for computers and analogous computationally-engineered artifacts (such as robots) is *procedural*, so that the "meaning" of an expression in "language" (however rigid or artificial) is the procedures which it designates or which supply its implementation, then analyzing this semantic requires explaining the environment within which procedures operate. This does not mean only one single procedure — the context that obtains prior to its starting and subsequent to its ending — but from a broader vantage point the settings in which we can speak of groups of procedures executed in sequence, forming an aggregate unit of computational activity. As I pointed out at the start of Chapter 20, most software applications are designed to enter passive states awaiting user input. It is only after a user-driven *event* — due *gestures* such as clicking a mouse button or tapping a keyboard letter/symbol — that applications launch a series of procedures responding to the user's (presumed) intent. Typically the series concludes with some visible change to application state (typically something the user sees on-screen) and then the initial passive state is re-entered. As such, the canonical context for software procedures on aggregate is the bounded by the *initial* user action (which sets parameters on how the application reacts) and the eventual (usually on-screen) content presented to the user. To the degree that computers have "semantics," we should analyze such semantics in terms of whatever intermediate processing connects the start of these sequences (user gestures) to the end (user-visible content).

In the case of robotics, the details are similar, except that most robots do not primarily function merely by presenting information on-screen (unlike personal computers or even *en situ* computer-like devices with touch screens, e.g., interactive transit displays). A GUI console can be one part of a robot's equipment, but more broadly users expect robots to fulfill user intentions by moving themselves and/or external objects. Although robot actions might be preprogrammed (and they can be semi-autonomous, deciding some actions on their own), to sustain the analogy with other computing systems let's consider the case where a robot launches a series of actions in response to an explicit request by a human user ("lift the yellow box" or something similar). In that case, the robot should respond to the request by moving and/or entering a configuration where it may physically comply. (In some cases the user input might be provided in advance, with the expectation that the robot only responds to such information actively under certain circumstances; for example, robots might be instructed to perform some task only after they sense a particular kind of change in their environment). As with GUI front-ends to traditional applications, the robot's procedures propagate in response to user input, and have a well-defined end-point, but here the terminus of a procedure-sequence would (primarily) not be something visible on-screen, but rather a robot changing its location and/or configuration in physical space (possibly affecting some external object in the process).

The units of GUI presentations are "application windows" (visible screen areas), inside of which are "controls" such as buttons, labels, scroll bars, magnitude-indicators (consider how we drag an icon to partially fill or empty a bar, to zoom in or out on a text or graphic display) and dividing lines; within those controls, in turn, can be text (including numbers) or other displays of textual and/or quantitative information. Therefore, the "end state" of a series of procedures, at the application level, would be a change in visible GUI display through some of these constituent elements. Likewise, in the case of robots, the components of a robot's "configuration" would be angles of gears, extenders, wheels, and whatever mechanical devices robots utilize to move and reposition themselves. At the culmination of a procedure-set enacted in response to user input, then, the robot will in general be a different state with respect to their location and the orientation of their arms, graspers, mounted cameras, or whatever other mechanical equipment they have on-board. We can see these physical gadgets as analogous to windows, controls, and text/number displays in the GUI context.

Based on that analogy, deciphering the semantics of reactive GUI programming can be an indirect pathway to formulating semantic models appropriate for robots, because the constituent parts of GUIs provide structural analogs (with respect to the origination and telos of procedure-sequences reacting to user input) to robots' movements and configuration. In both contexts, the "semantics" of the computational system is determined by how procedures react to user actions so as to yield state-changes (to GUIs' visible contents and robots' orientations, respectively) that are "correct," or consistent with their initial intent, from the user's point of view ("correctness" in this sense taking the place of being "meaningful," for the purpose of semantic formalization).

Given the analogy of GUI state to robots' orientation, an analysis of GUI structures (intended to be included in an overall procedural semantics for computer code), which will be the focus of the following section, can perhaps serve as a starting point for further work in robotics and other areas that generalize and extend the familiar desktop-computing environments.

3 GUIs, Robots, and Environments

Current paradigms in desktop-style GUI engineering have stayed relatively consistent since roughly the start of the century. The heyday of GUI applications (often called "rich client" or "thick client" interfaces, to distinguish them from web applications — "thin" clients — which need to run in an internet browser) largely coincided with a period in programming methods where Object Oriented techniques were predominant. Neither Object-Orientation (OO) nor desktop-style software is canonical to the same degree as two decades ago, given the emergence of popular programming language which de-emphasize OO structures on the one hand, and web or smart-phone apps as an alternative to desktop environments on the other. However, for many categories of software — for instance, Scientific Computing, or for that matter the IDEs (Integrated Development Environments) that programmers use to write other programs (of all varieties) — desktop technology remains the only feasible option. It is impractical to engineer applications that work in web browsers or within the limited screen-size of a smart phone, once applications get past a certain threshold of functional and presentational complexity. For this discussion, then, I will refer to GUI applications as canonically native-compiled software components that run on a desktop computer or a similar environment (as opposed, for example, to web applications which are viewed in a browser and run by code interpreted through the browser rather than compiled to native machine instructions).

Object-Oriented techniques remain the predominant method for GUI development in this sense, largely withstanding the familiar criticisms leveled against the OO paradigm in other contexts (such as database or server-side programming). It is reasonable to conclude that OO is an effective modeling tool for GUI functionality, one which cannot readily be replicated by functional programming (FP) languages, for example (to prove the point, consider that FP bindings to GUI software libraries, particularly cross-platform ones — i.e., intermediate libraries that allow FP code to construct and manipulate GUIs — are notoriously buggy, poorly maintained, and/or difficult to use; for example, compiling and merging these libraries with an application's overall code base tends to be more difficult with FP bindings than using OO libaries written in native languages, such as C++, directly). So, again narrowing the focus of the current discussion, we may consider general-purpose models of GUI elements and structures to be preeminently Object-Oriented.

Accordingly, GUI elements are canonically objects with state (FP languages often are designed around "stateless" values, but in the GUI context recognizing mutable state appears to be the most natural digital encoding of the phenomenon being programmed, namely visible windows and parts thereof wherein applications present information to users). There are several different facets of GUI object-state, which may be grouped into three overall categories:

Visual Style Effective GUI design often depends on conscientious choices for colors and other visual details. This is particularly true because the visual appearance of GUI elements includes not only how they appear in their "normal" state, but also how their appearance is subtlely modified as a cue to users trying to orient themselves vis-à-vis applications' interactive features. For example, GUI buttons and clickable spans of text almost always provide some visual indication that these are not only static elements, but can respond to user events, such as clicking the mouse while the cursor hovers over a button. To help guide users, buttons (and text, when appropriate, plus other clickable GUI controls) take on a different style, usually with a variant color scheme, when the cursor "hovers" over the element (sometimes called a "mouseover" or "rollover" state) — this visual cue shows that the control is indeed clickable and that the mouse is currently positioned on the item, so that clicking the mouse (via a physical mouse-button) is understood to be an action for "clicking" the (virtual) GUI button/control. Once a button *is* clicked and while the response is progressing, a button may appear in a "down" state, which has a style distinct from both normal and rollover.

Alternatively, some buttons (or other controls, such as checkboxes) are not primarily designed for requesting a specific action, but rather to set some flag or indicator (e.g., a checkbox in a GUI area related to text-searching would be set when users want to do a case-sensitive match — in which a capitalized word for instance would not be considered a match of a lower-case search phrase — whereas the default setting, indicated by an unchecked box, would be case-insensitive). In these contexts there would be a visible contrast between "up" (or "unchecked") and "down""checked" states.

Moreover, buttons and other GUI elements can sometimes be "disabled," meaning that one cannot interact with them according to normal patterns because (given present application-state) their associated actions would be impossible or illogical. For instance, a button which saves a file (i.e., signals the application to enact that functionality upon being clicked) would not make sense if there were no file open. In that case, a disabled state and its style would indicate to the user that the button is not presently able to accept interactions (it is common to use gray colors to cue such states), but the styling of these states should not obscure the control's primary purpose when it *is* enabled. A disabled "save" button, say, should not be given a style where it is difficult to read the "save" label (or whatever icon replaces it).

As illustrated by button-states, most GUI controls have multiple different visual states and their changes are intended to cue users on how the controls may be utilized interactively. For this reason graphic styles have to be designed attentively, because each control-state should be visually distinguished from others, but also the styles should not interfere with the control's primary functions. For example, a button's descriptive label/text (and/or its icon) must remain visible even while its background and/or outline color (or other style-state effects, such as a color-gradient) changes. Style rules for GUI elements in general need to adopt color schemes in which coloring is relatively nonobtrusive (colors should not be overly dark or high-contrast) but still allows components to show a variety of different states.

Many applications do not use a single style-scheme, but rather allow their styles to be adapted for different users and/or different platforms. Most Operating Systems have preferred stylistic concepts, so that applications on a Mac tend to follow different conventions than those on Windows, or Linux. Consistent visual patterns from one application to another help users familiar with one program to learn how to use new software more quickly, so applications generally try to conform to Operating System standards; as such, cross-platform applications often employ multiple style schemas, depending on the kind of computer where the application happens to be running. In addition to these operating-system variations, applications may also recognize flexible styles that can be adapted for different users, in case someone wants to view larger text, say, or wants to switch between different (human) languages for GUI labels.

Information Content Most GUI elements exist to show data, so their state is obviously influenced by the nature of information they seek to present at any given time. In the simplest case, this data is in textual form, or is numeric data presented textually. For example, consider a label that shows the name and folder path of a file currently being viewed; when a new file is opened, the text shown on that label will be updated. Here the relevant data is presented just through a text sequence. Or, a digital map might have labels to represent the current latitude and longitude coordinates under the cursor. These are numeric values, but can be represented for users merely by showing the numbers in character form.

Other numeric displays, however, might represent magnitudes more pictorially. Temperature, for example, may be expressed by coloring a bar at a height proportionate to degrees fahrenheit or celsius; the bar filled higher to the top would indicate higher temperatures, and lower bars show colder temperature. Color could

also reinforce these visuals, with warmer temperatures represented via reds and yellows and near-freezing conditions cued by blue or purple. For some GUI controls, numeric values might be conveyed by visuals mimicking real-life physical gadgets, so the volume on an audio player could be shown via circular indicators mimicking the knobs on a stereo.

One relevant contrast between numeric and textual data is that the latter can be long and open-ended; GUI controls designed to show paragraph-like text can represent content spanning many lines. On the other hand, numerical indicators typically show only one or two quantities, particularly if they are gadget-like displays (relying on bar/line-heights or angles to connote magnitude, rather than printed numerals). Multi-part numeric data would then normally be represented via collections of different controls. Where appropriate, quantitative data sets could also be shown via charts or diagrams, either static images or aggregate components with multiple independent controls.

Aside from textual and numeric data, some GUI controls are engineered to show 2D images, or even interactive 3D graphics (in the latter case, users can modify 3D scenes — moving or rotating the display to show different angles onto the scene, which creates the illusion of three dimensions even though the graphic at any moment in time is confined to a 2D computer screen). Although we can speak of graphics displays as presenting "information," this is a more informal usage if a control shows, for example, a photograph. However, images can also be generated on-the-fly to visualize data structures, in which case we could say that image-graphics (at least when showing tools such as charts, graphs, or diagrams) represent one strategy (separate from text and indicators) for data-visualization. Image data, then, occupies a gray area where we can sometimes speak of controls simply showing users the contents of an image-file and sometimes instead treat image-displays as vehicles for presenting structured data analogous to other kinds of GUI controls.

To some degree this can also be true of text displays, when for example a control is intended to show contents of an HTML or PDF file. The PDF format, in particular, is engineered so that each individual document-page becomes treated by applications as if it were an image or photograph (PDF text is not a data structure; pages instead are rendered and provided to GUI front-ends via mechanisms similar to photo-displays). On the other hand, PDF files also often include word-data so that applications can perform searches inside PDF text (although such data tends to be imperfect, with situations such as hyphenated words, ligatures, and foreign accents causing transcription errors; in short, PDF does not directly provide an accurate machine-readable transcript). Text in HTML is similarly largely static, but users can select character-ranges and activate hyperlinks in both formats (this is true for any HTML document and most PDFs). Controls showing full text files, then, analogous to image-displays, are in some sense intermediate between organized information-displays which directly map data structures to GUI aggregates and raw file-viewers which are only in an indirect sense representing "information," but are primarily rendering file-contents into a static visual form.

The more canonical case of GUI controls showing structured data is one where GUI screen-areas are subdivided into units that represent individual data fields on a fine-grained scale. For example, tabular data or CSV (Comma-Separated Value) files can be presented via spreadsheet-like controls, which are subdivided (by rows and columns) into cells that typically have one single numeric (or short textual) data. Information shown in these sorts of displays may originate from files, insofar as such GUI controls are used to access files they need to load data into computational structures first, and then map individual parts of these structures onto parts of the GUI display. The correlation between files and front-end controls is therefore granular and systematic, whereas image-viewers simply calculate how to render pictural data via on-screen pixel-areas and display the resulting graphs visually, without constructing distinct areas or sub-windows to show specific parts of a loaded file. Text displays can be between these two alternatives; they are largely unstructured (like pictures) but portions of text may be isolated as if they were interactive windows (such as hypertext links) and users may select character-ranges (e.g., for copy-and-paste) in which case the screen-area spanning selected text needs to be treated as an interactive window in some sense.

GUI controls as containers for other controls Because controls whose explicit purpose is to render individual data-units tend to handle relatively small pieces of information (e.g., single numeric values), multi-part data structures have to be displayed through higher-scale controls that group together such more narrowly-focused controls. Partly, containers along these lines serve to aggregate smaller controls in an orderly manner; for example, a display of several numeric fields could be arranged with a single key-value pair on each line of a multi-line list (tables and spreadsheets, grouped by both rows *and* columns, are a more complex organizing motif). GUI containers do not need a fully symmetrical layout; often smaller controls are arranged in table-like patterns but with varying numbers of elements on each line/row, depending on the amount of space each element needs. The important detail is that users can visually isolate and comprehend specific points of information, which is easier if GUIs are visually balanced and logically arranged. Grouping related controls — and leaving spaces (or drawing lines) between different such groups — is also a useful layout convention.

Apart from just organizing controls, GUI containers can also compensate for limited screen real-estate: GUI windows often have more data available than can be shown on-screen all at once. As such, containers can use various tactics to show some contents and hide others, such as scroll-areas (where users manipulate scroll bars to alter which controls are visible and which are hidden); tab/notebook windows where entire (sub)windows are "stacked" or lined up so that only one window is visible at a time; dialog boxes (which are typically temporary displays that become visible to show the user specific information and then are closed); "wizards" (in effect, multi-page dialog boxes); and "collapsible" displays, whose parts can be toggled between visible or hidden states. An canonical example of the latter is lists of files in folders, where individual folders may be seen in an "open" state (such that folder-contents, including other folders, are listed) or "closed," and likewise for subfolders inside (open) folders, and so on. These kinds of controls are sometimes compared to an accordion, because as folders and subfolders open or close their screen-areas expands and contract re-

spectively, creating a visual effect vaguely reminiscent of accordion leaves' folding patterns.

There are several notions of GUI controls' *state* covered by the above breakdown. Controls can have *functional* states, reflected in style changes, such as a button in "normal" or "hover" modes (or also, say, "down", "checked," and "disabled"). Most controls' state is also determined by the data they present (e.g., for single-value numeric indicators, the specific quantity expressed by the control at a given point in time). And, for containers (GUI controls which group together multiple smaller controls), state is also characterized in many cases by the interplay of which elements are currently visible, and which hidden.

The forms of GUI state just enumerated refer specifically to controls' features that are visible to users, so that state-changes convey information. This includes specific data-points which a control is designed to display, such as a numeric value; it also includes hints guiding the user to interoperate with a GUI most effectively. Mouseover effects, for instance, communicate which controls can be "clicked" and which not. There are other, more subtle forms of cues as well. For example, when the mouse hovers over the corner of a resizable control (so that "dragging" — moving the mouse with the left button held down — adjusts the control's size by pulling the corner inward or outward) typically the cursor, that marks the current mouse position, will switch to a different icon (one drawn to imply something getting larger or smaller); there might be extra coloration or thickened lines at that corner too. Similar patterns apply to other "drag" scenarios, such as moving a divider line between two parts of a "split" container, where different controls lie on either side of the line; moving the line resizes both parts of the container (one gets larger, the other smaller) potentially hiding or unhiding contained content and controls (depending on which side they lie on). Another pattern is "drag and drop," typically involving icons being moved (figuratively "dragged" by the mouse) from one container to another: in some contexts the receiving container takes on a colored background to convey that it is possible to drop an item on its interior.

Some visual hints are explicitly designed to inform users about applications' features, or the status of current operations. For example, a mature front-end will often have "tool tips" for many controls, where hovering a mouse over the control causes a box of explanatory text to temporarily appear, explaining the purpose of that GUI element. Tooltips serve both as didactic guides and as further visual markers that the relevant control accepts interactions. Other examples of indicative GUI features include progress bars (which are usually visible only while an operation is ongoing, such as downloading a file); message boxes (presenting users with brief informative text, such as declaring that a download has completed); and status bars (typically single-line controls with text related to recent activity, e.g., "download complete": unlike a message box, which might also inform the users of something like a download completing, status bars are usually permanently visible in the front end). For container-controls that alternately show and hide inner contents, visual cues sometimes clarify what material is currently visible: e.g., a "notebook" container, which has multiple tabs each showing a different window, will color and outline the tab corresponding to the currently open window with a different (more emphasized) style than the other tabs (whose windows are invisible at present). Details can also be conveyed by discreet changes to textual data: for example, if a file has been modified and not yet saved, a label or status bar showing the file name will often append a small marker, such as an asterisk, to the file name, reminding users that they may need to save the file.

Style conventions also inform how GUIs help users track the current mouse position. In general, the mouse is tracked with a cursor, which moves across the screen when users move the mouse (or run their fingers against a touchpad); a mouse click (or tap against a touchpad) is understood by the application to signal the user's intent to execute a "click" gesture at the screen position where the cursor currently hovers. The cursor will then take on different icons depending on how the click would be processed, given the control under the mouse position (an example mentioned above would be a "resize" icon when the mouse hovers over a corner; this works also for some controls' edges); cursors tend to be given an arrow-like icon in these settings. When hovering over text, on the other hand, cursors usually take on a shape reminiscent of a capital letter "I."

In some cases the on-screen mouse position can be correlated with some physical or empirical coordinate system — a good example is digital maps, where positions inside the map correspond to latitude and longitude numbers. Here it is common to print the geographic coordinates next to a corner of the map, so that the numbers change as users move the mouse around the map-view, which both aids users if they are trying to find a specific latitude/longitude pair and helps visually reinforce the idea that mouse-positions map directly to geospatial points. A similar pattern in one dimension would be using a mouse to advance or rewind an audio or video player, wherein a numeric control shows the time elapsed at different points relative to the file's start (the convention being to pair audio/video controls with a line-bar and position indicator such that moving the indicator right/left implies forward/backward in time, respectively). Finally, some controls display graphics or data charts/diagrams against a background of orthogonal gridlines (similar to "graph paper" used for drawings), or sometimes just the lattice-points at gridline intersections. In these contexts mouse position may be indicated by highlighting the gridlines which a mouse passes over as it moves.

One takeaway from this overview is that well-engineered GUIs are highly interactive; there are many conventions which govern how front-ends should present subtle visual cues and state-changes to assist users when interacting with the application. Users may not appreciate the full scope of these effects, becoming habituated to "responsive" features and perhaps absorbing them at only a subconscious level; however, front-ends which fail to implement expected interactive patterns, where GUIs are not sufficiently responsive in terms of giving users frequent feedback on current state (such as mouse position) and interactive possibilities, would almost certainly be experienced by users as erratic and confusing. Most of the front-end patterns mentioned here are probably familiar to people who work on desktop-style computers, but it is nonetheless useful to outline them explicitly, precisely because well-engineered responsiveness effects will be subtle and non-distracting. When we appreciate the full scope of interactive patterns, we can perceive the complexity

of standard GUI programming: a well-featured front-end application will typically have a large collection of associations and interconnections between GUI controls, and mappings between control-state and visual styles/interactive cues. Systematically engineered application will carefully document these interconnections and implement the associated front-end functionality via a consistent set of coding patterns.

As discussed in this chapter (and also touched on at the start of Chapter 20), application functionality can be analyzed in terms of operations and procedure-sequences that are, in effect, bookended by user actions (which invite responses from the application) and by visible GUI changes (changes in the state of one or more controls, communicating to users the results of the actions they requested). Reviewing forms of GUI state serves to explicate the second half of this equation. We can also examine user gestures, constituting the first half of the "equation," insofar as gestures signal user intent, which leads to application responses, and finally to GUI changes. Gestures are, indeed, closely connected to GUI state, because a user's actual (literal and physical) action, such as pressing a mouse button, needs to be interpreted in light of current GUI state to be seen as an "action" in the "virtual" context of a front-end display. We speak, for example, of "clicking" a GUI control; in reality, the "click" happens on the physical mouse, but the presence of cursors and mouse-tracking permits this physical movement to be read as an interactive gesture within the scope of a software component. In this sense the "semantics" of user actions builds off of GUI state analogously to the semantics of GUI visuals conveying information, so that the two may be analyzed together.

3.1 The semantics of GUI-control state

To briefly clarify terminology: a user *gesture* refers to some physical action which is *interpreted* as a specific kind of request, input, or instruction. The gesture is interpreted as an *action*, in the sense that gestures signal actions according to patterns which reappear in the logistics of how people interact with software (patterns that users recognize). As a canonical example, the physical gesture of tapping a mouse button is interpreted as the contextual action of "clicking" a GUI button or other GUI control. The relation of gestures to actions can perhaps be compared to that between audible words and their intended lexemes.

Mapping gestures to actions depends on GUI state, such as the current mouse-position. Accordingly, GUI state overall provides the structures against which individual actions are defined: an action of *clicking a* (GUI) *button*, for example, is meaningful in the context of a front-end environment where the specific button is among many controls that could be "clicked," and given that it is enabled and visible, etc.

Some GUI coding contexts will refer to "actions" as akin to "requests" that may be cued by more than one user gesture. For example, instructions to save a file could derive from clicking a "save" button, but also selecting a save option from a context menu, or hitting a combination on the keyboard, such as "control"-plus-"s." For more rigorous discussion, I suggest referring to actions in this multi-indication context as "requests," so that the various gestures signaling one common request (like "save file") each have distinct actions. In any case, all actions then get routed to interpretations of users' intentions; we can use the term "requests" to designate these interpretations in general, so that some requests are cued by one single action and some are targets of more than one possible action. Current GUI state determines both how *gestures* are interpreted as *actions*, and how *actions* are interpreted as *requests*.

By design, applications are engineered to *respond* to such requests, and GUI state (alongside application state overall) contributes to how software models the request to be handled. Consider a notebook control that contains multiple windows, each displaying an editable text document: when the user requests a save, the application would infer *which* file should be saved based on which document's window is currently visible. This is an example of GUI-specific state. Whether or not the document has been modified (affecting whether a save is necessary), and the file path that would be overwritten, would be an example of application-state proper (because these details are not determined by the visible appearance of GUI elements; although the GUI can convey to users such information as whether the currently viewed document is modified).

Some user requests may be handled in different ways depending on specific properties of application state or the surrounding computer environment. Consider a request to save a file: if the file already exists, saving the updated version involves replacing the contents on disk; in some cases, the current user is not authorized to make such a change, so the save action (with the current file name) cannot be completed. On the other hand, if this is a *new* file, the user needs to assign it a name. There are, then, at least three possibilities: the file can be overwritten with its current name; the file cannot be saved with its current name but may be saved with a new name, so that there will be two different versions on disk; and the file cannot be saved until a name is provided, which will then be one single version of the file. How the application fulfills the user's request will different in each of these scenarios. Some of these differences might be noticed by users: if the file needs a new name, users will generally be shown a "save as" dialog configured for choosing file/folder names, whereas if a file is overwriting a prior version users might see a message box confirming that they intend to do so (maybe with an option to make a backup of the older file).

The point of this example is that responding to user requests demands a rigorous calibration of GUI and application state. This is one reason why Object-Oriented methods predominate in front-end programming; the logic of such state-correlations is typically simpler to reason through via OO paradigms than alternatives. Indeed, GUI controls are best encapsulated as objects-with-state, and because GUI controls are functionally interconnected with many other application elements, similar OO concepts tend to be most effective on the application side as well. For example, if a notebook window shows one document, the window itself corresponds to one GUI object, from the front-end point of view; on the application side, the actual document visible *in* the window might be a different object, encompassing data fields such as file path, file type, length, edit history, and permissions (e.g., the identity of the user who created the file).[4] User requests relative to the *GUI* object would be carried

[4]Sometimes deciding whether to qualify state as GUI or application-level can be a gray area: consider undo/redo history. Should user actions that can be reversed be tracked at the GUI level, becauese these actions are signaled by GUI actions, or within the application logic, because requests are managed by application code? This question determined whether undo/redo history should be categorized as part of GUI or application state.

out, by the application, in the context of the corresponding application object. These functional interconnections are best managed by maintaining association between application and front-end objects (which implies an overall OO system unifying these two contexts).

3.1.1 Extending Object Orientation

One might argue, however, that there are details in application/GUI object systems that are inadequately addressed even by conventional OO models, and call for extensions to the OO framework (which is a different matter than replacing such a framework with something like FP methods oriented to stateless values).

Here is one example: in some contexts a front-end display will include two (or more) GUI controls that are styled and positioned so that the user views them as an integral unit. A simple case is a button which has both an icon and a descriptive text label; this arrangement can be implemented with a label control placed next to a button control, but configured so that the label appears to be a logical continuation of the button itself. In this setup, some facets of GUI state should be shared between the two controls as if they were one single element. For example, if the button is disabled (causing it to adopt a new style), the label should be restyled as well. Or, consider tooltips, as mentioned earlier — explanatory text that temporarily appears near controls when the mouse hovers over them. If users are to experience a button and a label as functionally integrated, this pattern is reinforced by them sharing a tooltip, which means that the same tooltip content should be applied to both controls.

Consider a procedure to define this tooltip text. Ideally the same procedure would act to both objects, because we want to reinforce in code the idea that two controls share some state-details, including their tooltips (we do not want this correlation to be dependent on two different procedure-calls, partly because programmers may forget one of them and partly to remind those maintaining the code that the two objects are interdependent). In OO systems, the basic pattern is that procedures which affect the state of a specific object are implemented via "methods," in which a specific object is singled out as a distinct form of input value, sometimes called the method "receiver" (in languages such as C++, C#, and Java, "receiver" values are designated with the special keyword **this**; elsewhere the most popular name is **self**). The problem with this convention in a context where procedures are intended to synchronize state between *two* objects is that it seems artificial to choose one or another as "the" receiver — a more fluent paradigm would be to allow *two* receivers (or more), in the same way that procedures can have two or more normal input arguments.

In Chapter 21 I mentioned the idea of grouping procedural parameters into "channels," including a "sigma" channel that can generalize receiver-objects to object-lists (derived from the "sigma calculus," which attempts to model OO methods by extending lambda calculi). This would be one way to tweak OO systems — permitting multiple receivers. There is no technical reason why programming languages could not support more than one **this** value; by analogy, most languages only recognize a single procedure *return* value, but there are outliers (e.g., Common Lisp), and generalizing returns to multi-value lists does not present special syntactic or implementational difficulties in those contexts.[5] By way of reference, the code accompanying this chapter includes some functionality which emulates "sigma channels" in ordinary (albeit unconventional) C++.

Apart from multiple method-receivers, a more substantial potential departure from conventional OO suggested by GUI state involves user actions and corresponding "signals." A common pattern in GUI programming is to generalize the idea of procedures calling one another to a model wherein procedures can post and/or respond to signals; such a system is usually referred to as a *signal/slot* mechanisms, where signals can be "connected" to slots. Rather than one procedure calling another directly, in this architecture the first procedure "emits" a signal, which the second then "receives," having the effect that the second procedure gets called, with parameters (if any) provided by the "sender." Unlike direct inter-procedure calls, these connections are routed through an intermediary component and they might be dynamically reconfigured: signal/slot connections can be newly created or dropped while an application is running. When a procedure emits a signal, there is no guarantee that any other procedure will implement a "slot" that receives it; but also *multiple* procedures might connect to the original slot). Compared to direct inter-procedure calls, the two (or more) procedures involved are thereby less tightly connected.

Programming environments that employ signal/slot connections often use this technique to handle responses to user gestures. After a user clicks a (GUI) button, for example, an object representing that button broadcasts a "click" signal, which in turn is handled by procedures to initiate the application response. For example, if the button in question represents requests to save a file, there will be some procedure whose purpose is to wait for "save" signals and then initiate the application-level process of actually saving the relevant file. The slot thereby connected to the "save" signal only is responsible for initiating save operations; it does not consider where the save button is located on screen, or its GUI state and state changes (mouseovers, tooltips, etc.). On the other hand, the button-object would track these GUI-specific details but not get involved with actual save operations. Signal/slot mechanisms are as such (as this case-study suggests) valuable tools for enforcing "separation of concerns," where each procedure and each object have a relatively narrow set of responsibilities. Signals and slots prevent objects from becoming tightly coupled, because signaling objects can be designed in isolation from procedures that receive (and act on) their signals.

Although a powerful pattern that fits organically within OO paradigms, signals and slots are not directly supported by the OO model, and few programming languages have intrinsic signal/slot features. In C++, several libraries emulate the signal/slot patten. The Qt framework, which is both a GUI library and a general-purpose technology for application development, supports a coding style wherein signals and slot function essentially as native language features — however, this is only possible because Qt implements an extended compiler workflow involving pre-compilation parsers and code-generators, in effect implementing an extension to the C++ language itself.

[5] Disambiguating methods called on "multiple receivers" may prove more complicated when two **this** objects are of different types — which class should be searched for corresponding methods? — but compilers could adopt similar rules to those used when matching procedure-names to functions wherein argument lists have multiple types.

Because most OO languages do not intrinsically support signal/slot connections — that is, they cannot be expressed directly through the language's syntax and semantics, but rather have to be approximated by code libraries (potentially including language extensions through code generators) — this patterns is for all intents and purposes an extension to OO which is not part of the basic OO model.

One further detail concerning signal/slot systems involves properly aligning multiple parameters of GUI state which should adhere to certain design regularities; for example, different lengths or magnitude in the front-display preserved in a constant ratio. Consider a scroll area showing an image: if the image's size (at its current zoom level) exceeds the dimensions of the scroll area, then the visible part of the image is "clipped" and users can scroll left/right and top/down to alter this visible area. On the other hand, if the image-size is *less* than the scroll area, it is conventional to center the image so that left and right margins, and likewise the top and bottom, are balanced. If the image is resized (by zooming in or out) its top-right coordinate therefore should be recalculated. Similarly, if the scroll area's viewport is resized because the user expands or contracts its surrounding window, the image itself would either have to be resized proportionately, or else moved to a different top-left position, to preserve the margin's desired left/right and top/bottom symmetry. In this example, there are several different numbers that need to be tracked in coordination: the overall window's size; the width and height of the scroll area's viewport (its visible screen-area); and the image's position, dimensions, and zoom factor. Changes to any one of these measures may need to trigger changes in some or all of the others, to ensure that the GUI continues to feel visually balanced.

Because multiple quantities are involved, orchestrating the proper symmetry in this example can require coordinating multiple signal/slot connections. For example, an object representing the main window might emit a signal when it expands or contracts. Most GUI systems employ "layout managers" which calculate how controls should be oriented relative to application windows, so that front-ends can have a roughly tabular, gridlike, or top-down organized layout, with controls that need a fixed amount of space (such as labels whose text does not change) remaining constant while others gain or lose space in sync with available screen-area (scrollable containers, for example, can be downsized because users expect to scroll around their interior anyhow). Layout managers will therefore tend to automatically resize certain controls, which can potentially trigger new signals: a screen area resizes because its surrounding window does, and both end up producing signals broadcasting their new state. Code responsible for the image display accordingly should keep track of either or both signals and modify the image's own size and/or on-screen placement accordingly.

Quantitative changes such as window-size may also trigger other kinds of GUI state-changes. For example, if an image is *smaller* than its scrollable viewport then scrolling per se is not possible; this situation is typically cued via scrollbars being visible but grayed out (users perceiving the scrollbars, even if they are disabled, reinforces that they *will* become active when necessary). Then, given either the image zooming in or the viewport shrinking, once internal scrolling becomes necessary the scrollbars should be enabled, taking on a new visible appearance. Resizing and zooming may therefore both affect the scrollbars' styling.

For a related example, when an image is *first* opened many picture-viewers resize the main window so that the image can be seen, in its entirety, in its native size. At this point, the scroll area and main window dimensions would be calculated on the basis of the image; the user can then resize the window if desired. However, *after* the image is loaded, ordinarily the user is free to expand or contract the window irrespective of the image size (scrolling being available to compensate for when the window get too small to show the whole image). There are, accordingly, two different scenarios for how the relation between window-size (and by extension a scroll area's viewport-size) and image dimensions should be calculated. Application code needs to keep track of GUI "history" to properly separate these different contexts. For example, if the main window is resized immediately after (and because of) a new image-file being opened, the window's resize-signal should be processed differently than a similar signal emitted due to users manually adjusting the window (typically by pulling at one corner or by clicking maximize/unmaximize buttons).

The point of these examples is that signal/slot connections often do not occur in isolation, but must be coordinated to maintain desired style and layout conventions in the front-end. Often these involve groups of interrelated parameters reflecting states from multiple different GUI controls. GUI state can be guided by sets of "rules," such as that an image top-left corner must be positioned and repositioned to ensure balanced margins, or that scrollbars should be visible but disabled when image-size is *less* than the available viewport area, but automatically enabled when the former expands to beyond the viewport dimensions.

In the image-view case there are at least four quantities or coordinate-pairs which are numerically interdependent: the image width/height; its top-left corner position; the viewport dimensions; and the image zoom. Most programming languages do not have an intrinsic feature for sustaining predefined mathematical relations amongst two or more independent variables along these lines — there is no way in code to declare interrelated variables such that changes to any one (or more) value will automatically propagate to the others. The closest approximation might be to group all these values into one data structure, and then make any changes to specific parameters via procedures specific to that structure, which can be engineered to update the other fields as needed. In that case, attributes such as an image's size would not be magnitudes (or pairs thereof) directly, but would instead be handles onto specific parameters held within the coordinated data structure (one added complication is that while that structure's fields are interrelated in the sense of demanding specific mathematical invariants, they are also properties of different controls' states, so they need to be properly classified vis-à-vis the control they are associated with).

Orchestrating these sorts of interrelationships among multiple objects and data-fields is a common problem but without a canonical design pattern, in particular in the OO context (this is one areas where Functional Programming has produced somewhat more concise formulation, that OO could perhaps borrow). Therefore, another kind of potential OO extension relevant to GUI state would be supporting groups of interrelated parameters, distributed across multiple objects but bound by stipulated invariance requirements. Such features would overlap with signal/slot systems because signal/slot connections are sometimes defined specifically to ensure

proper inter-parameter relationships (by receiving signals notifying change in one interlinked value and then updating the others proportionately).

From a practical point of view, these examples point to OO extensions that could potentially be incorporated into new language standards. More theoretically, we can also study how such extensions add representational features to OO models, enhancing the OO repertoire for capturing the state and interrelationships of complex systems. The "theory" in this case could provide a further perspective on OO "semantics" in the sense that Object-Oriented principles can inform data models, "knowledge system," and other technologies that are designed to have some explicit connection with real-world objects and phenomena. More direct applications of an expanded OO "theory" might also lie with tools such as code-analyzers, software-development tools, and compiler extensions, collectively serving to accelerate the implementation and quality of GUI front-ends.

Front-ends discussed here thus far have implicitly been focused on two-dimensional text and graphics, either viewers of 2D images or layouts presenting structured information, via text, printed numbers, or magnitude-indicators. In contexts such as robotics, GUIs also often work with 3D graphics, videos, and other multimedia assets which have properties and present complications different in kind from the more 2D contexts and issue raised to this point. Partly so as to circle back to the theme of robot environments, these multimedia concerns will be the topic of the rest of this section.

3.2 3D Graphics and Robotics Front-Ends

When displaying 3D graphics, applications need to support a further group of user actions because there are a larger set of possible view transformations, compared with 2D resources. Such graphics can be "panned" in three different directions (moving the virtual camera along **x**, **y**, and **z** axes), and could also be rotated (which is implemented as an effect akin to the virtual camera being spun along the outside of a sphere centered at a point along the camera's line of sight). Also, the scene may be zoomed in and out (preserving the camera's line of sight but moving its point of focus closer to or further from the camera). With a three-button mouse, pan and zoom in *two* dimensions is typically signaled by activating the central mouse wheel (for zoom) and pulling scroll bars (to pan, moving left/right or up/down) around the screed display). Sometimes panning can also be achieved by "dragging" inside the scene; moving the mouse while keeping its left-button down, with the cursor positioned inside the relevant view window (similar to drag gestures for dropping and resizing controls). When working in *three* dimensions, these kinds of drag actions are most commonly interpreted as rotations. Most 3D viewers support panning and zooming by recognizing "keyboard modifiers," wherein users hold down a key (often **shift**, **control**, or **alt**) while also moving the mouse.

In general, 3D displays are "responsive" in the context of users modifying the visible scene, but many of the interactive protocols which are commonplace in 2D structured front-ends do not apply to 3D graphics (nor, for that matter, to 2D images such as photographs, rather than organized data-visualization). Some 3D environments, tentatively, have tried to get beyond these limitations. The premise here is that users may have an interest in getting information about a location or object which is one *part* of a 3D scene. With 2D structured displays, it is easy to allocation a small screen area specifically to represent its own data structure, where the application may respond by presenting users with a more complete review of the relevant information (perhaps in a separate window). For example, users clicking a "print" button will in most programs see a "print dialog" which contains multiple points of information about the current page and printer setup. In general, when a front-end does not have enough space to show a complete data structure, a GUI control can show an icon or a short description, giving users a chance to click at that point for a more complete view of the associated information.

Similar effects are sometimes warranted in 3D. In a virtual "street" tour, for example, a user will virtually digitally "walk" past different streets and buildings, and platforms such as Google StreetView insert icons onto the view for users to click for location-specific information — with a restaurant, for example, a page showing menus and reviews; or, with a transportation center, a link to transit routes and maps. Via similar methods, virtual tours *inside* buildings are sometimes annotated with icons that lead to information about individual rooms or objects, or else with short textual descriptions (potentially including hypertext links that can lead to more detailed views). Interactive icons in these contexts are sometimes called "tag" or "place-marks" and allow 3D scenes to have a select group of interactive features.

The above comments apply mostly to graphics rendered via 360 photography, which creates the illusion of three dimensions by pieces together many individual "panoramic" pictures (often taken from special-purpose cameras that capture images from all directions, rather than having a single view angle). In the context of engineered 3D views, such as those rendered via mesh geometry, Computer Aided Design (CAD) programs will sometimes isolate specific object parts and surfaces and make these interactive. For example, a CAD file modeling an industrial part may permit users to select one piece of the object and examine information such as its dimensions or thickness — in the CAD setting, someone can use the application to actually create and modify industrial designs, so selecting specific segments of objects can lead to operations that actually edit their properties within the overall design (e.g., making industrial subparts larger or smaller relative to their contaning physical object).

Despite these familiar use-cases for interactive 3D, there are multiple technical limitations which prevent 3D graphics from being responsive to the same degree and with the same granularity as structured 2D front ends. Computationally, the geometry of 3D scenes — most often derived from triangular mesh complexes (point clouds where triangles are formed by connecting nearby vertices) or 360 images — is fundamentally different than the layout managers powering 2D front-ends. Layout manager ensure that GUI containers remain well-organized and usable even when a window is resized or scrolled. In general, GUI *controls* are not placed *within* 3D scenes the way that they are situated in GUI layouts. As one progresses through a 3D view, different parts and objects come into view or disappear — it is not clear how one would implement 3D analogs to layout managers, which would need to continuously update layouts while scenes are rotated, panned, and zoomed.

Apart from the mathematical computations involved in translating scene-transform to visible geometry (each rotation or camera-movement alters all angles and causes different parts of the scene to be hidden, or unhidden) — which would apply to GUI controls immersed in 3D scenes no less than mesh-triangles or other units of 3D geometry, and which are exponentially more complex than calculations related to 2D front-end — interacting with controls visible *inside* scenes could easily become cumbersome for users. For example, in order to click on controls users would have to rotate or move around within the scene, and the graphics engine would have to determine how to represent icons, text, or areas for controls at different icons or zoom levels. Taking commercial tools such as Matterport Tours or Google StreetView as examples, immersing even simple "tag" icons in Virtual Tours yields inconsistent and uncomfortable GUIs — it is common in these environments for tags to appear and disappear for no apparent reason, and having seen a tag (or a clickable hyperlink) it can be frustrating to maneuver the onscreen view so that the tag/link can be activated.

Such issues with 3D GUIs are directly applicable to robot's environments, because in most cases robots will be receiving visual imputs via mechanisms similar to panoramic photography. Humans users wishing to remotely monitor and/or guide robots by watching video feeds based on robots' own inputs will therefore be presented with displays similar to 360 tours. Unlike virtual tours, however, robots are not primarily designed simply to create visual feeds for human viewers (although that can indeed be one of their uses: consider robots deployed to gather real-time images from locations that would be dangerous to inspect in person). Because robots typically process and structurally analyze their environments, inferences they make concerning object-boundaries and affordances could be incorporated into graphics accompanying their visual feed. For example, robots may be programmed to categories objects in their visual field and/or identify ones which they can move or manipulate. Within their video feed, then, certain objects will be singled out through Computer Vision, making it possible either for views of such objects to be visually altered/highlighted or for a separate front-end display, presented alongside the video feed proper, which schematically diagrams the robot's environment as it is modeled through image analysis.

Perhaps the most effect paradigm for robotics GUIs, along these lines, would be a pair between 3D capture of their raw video feeds and 2D structured windows presenting their environments schematically, with an emphasis on diagramming inferred relationships between objects rather than spatial accuracy. If a robot travels through a room with multiple object that it could move, for example, a schematic GUI container might represent each of these physical objects with icons or screen-areas arranged so that users could interact with the 2D diagram as a proxy for controlling the robot through its 3D video feed. For example, GUI controls for each movable object could be programmed with interactive functionality analogous to controls tied to application features — users might, for instance, click on or select context-menu items for specific objects as a way of guiding robots to move, inspect, or otherwise manipulate them. Here the combined system merging the robot with the user-facing front-end would react to user actions similar to a desktop application, but instead of the reactions between restricted to digital operations localized in the computer itself (like moving a file from one folder to another), users could employ such a setup to mold robot's actions on external, physical things (e.g., moving a box from one room to another).

In such a scenario humans would be relying on robots' visual processing to convert a raw video feed to a structured diagram (one that could be rendered in two dimensions, making interactive gestures easier). For this reason, there is a sense that humans would be interacting with robots by virtue of shared environment-models: people would find it difficult to guide robots in this kind of system unless they analyzed their visual inputs similarly to how humans would if they were seeing the same objects around them. In short, robots' mapping of immersive visual inputs to schematic environment-summaries would have to feel natural and intuitive for humans. This kind of phenomenological correlation may be much less vivid and intersubjective that joint-attention between people themselves (or between a person and an animal, in the right circumstances), but it at least lays the foundation for something resembling human/robot communication — if one accept the premise, that I emphasized in Chapter 23, that shared situational understanding is the origin-point for language.

Note that in this kind of setup humans would be talking to robots indirectly, mediated by diagram of robots' environments. For example, if a display shows (as a summarial view onto the robot's visual input) outlines representing boxes that could be moved, the human user would request such an action by virtually moving the object within the diagram, or by some analogous gesture (like selecting a "move" option from a context menu). The front-end, in short, would be managed by a software component that would translate user intentions as expressed through a GUI into commands that the robot understands. There would, in effect, be two different languages or sign-systems linked together: the idioms of a GUI front-end with which humans would interact, and then a distinct robot-language engineered to convey instructions to robots at a tractable level, i.e., where there are deterministic algorithms to translate expressions in this language to low-level commands affecting robots' physical movements and orientation.

The idea of people "communicating with" robots probably calls to mind a more direct exchange where humans perhaps "talk directly" to them, their words parsed via Natural Language Processing (this is the kind of scenario discussed elsewhere in this book; see Chapter *ref*). However, within the relevant NLP pipeline there would presumably be a distinction between components that consume the spoken language directly and those that model specific robot-instruction suitable for low-level responses. In short, even a medium of exchange which users perceive as communicating "directly" to a robot would most likely involve multiple stages, organized analogously to the GUI-based scenario where people interact with a GUI display in lieu of "talking" to the robot explicitly. Because of these functional parallels, designing systems based on GUIs is a strategy for fine-tuning robotics communication to develop intermediate algorithms which might then also be applicable to interfaces based on natural language. With a working pipeline to map GUI operations onto robot-action code, engineers could swap out the GUI front-end for an NLP interface, perhaps preserving much of the lower-level protocols on the robot's side specifically. Working with GUI front-ends first permits the two different dimensions to the latter system to be engineered independently — the NLP parsers would have their own

set of requirements and complications (effect speech-to-text translation, for example) that could be worked through independently of how the robot's target languages (to which either NLP input or GUI actions would both have to be translated eventually).

None of this discussion in the robotics context represents new theories or proposals; the techniques I have summarized here appear to be relative mainstream approaches to robotic engineering. However, it is worth emphasizing that building GUIs to serve as intermediate communication devices may provide both practical benefits (in well-engineered robotics systems) and serve as one stage toward direct human/robot communication (for reasons just sketched out). While GUI front-ends similar to those I have summarized are sometimes used for robot interactions, there is relatively little literature on implementing robots GUIs as a distinct subject-area within both software engineering and robotic (in the intersection of these two disciplines). This is a topic that deserve further research, partly because effective GUI programming is itself not given due attention as far as a systemic approach to GUI engineering, to make front-end more user-friendly and improve development tools. Many details of robotics-specific front ends, such as how to pair 3D graphics with 2D schematic diagrams, and in general how to make 3D displays more interactive, are themes applicable to other software domains apart from robotics proper.

In particular, it is an open question how best to associated parts of 3D views with independent GUI controls: should controls be *immerced* in 3D scenes, so that they are subject to the same variation in view-angle, rotation, zoom, and other 3D effects along with all other facets of 3D geometry; or should they "float" above 3D views when their associate object/scene-locations are visible; or should they be arranged diagrammatically in a separate window? Of course, these options could also be combined. To my knowledge, rigorous answers to these questions have yet to be determined, let alone implemented — nothing in current WebGL, OpenGL, or panoramic-photography software appears to provide the level of interactive granularity that would be implied by direct associations between 3D scene-locations and 2D-style GUI controls, whether these are visible within the scene (immersed or floating) or separated out to a separate display.

4 Conclusion

This chapter has considered the semantics of programming languages and GUI displays alongside semantic and syntactic theses related to natural Language. Combining investigations into human language with computational environments produces an admittedly idiosyncratic style of exposition, cycling from a philosophical dialect (as much philosophy of language as linguistics proper) to the tone of a Stack Overflow discussion board (as much programmer's workshop as mathematically-inspired computer science). Despite the fact that these two subject-areas lend themselves to different discursive conventions, I contend that there are some structural parallels between human and computer languages, which such a juxtaposition may highlight. In particular, something like a "procedural semantics" can be theorized in both domains. Also, type-theory offers a strategy to work through notions of preliminary comprehension: in both human and computer languages, a "procedural semantics" paradigm needs to explain how parse-graphs (for sentences and computer-code statements, respectively) can be isolated without the full feedback loop of grammar cross-referenced against semantic interpretation (which, for reasons sketched in Section 1, I believe is deferred from the language-processing stage to some extra-linguistic cognitive or computational faculty). One way to approach "partial" interpretation (grasping enough semantic detail to disambiguate parse-structures) is via type-systems, modeling provisional but only approximate semantic data (e.g., detailed enough to recognize parts of speech but not specific word-meanings).

References

[1] Ash Asudeh and Gianluca Giorgolo, *Enriched Meanings: Natural Language Semantics with Category Theory*. Oxford, 2020.

[2] Nicholas Asher and Zhaohui Luo, "Formalization of Coercions in Lexical Semantics". https://semanticsarchive.net/sub2012/AsherZhaoHui.pdf

[3] Jerry T. Ball, "Double R Grammar: The Grammatical Encoding of Referential and Relational Meaning in English". https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=34c258564f72b6dc6455aeaac6e6be1a4a07d262

[4] Chris Barker and Chung-chieh Shan, *Continuations and Natural Language*. Oxford, 2014.

[5] Lawrence W. Barsalou, "Grounded Cognition". http://matt.colorado.edu/teaching/highcog/spr10/readings/b8.pdf

[6] Hans C. Boas, "Cognitive Construction Grammar". https://sites.la.utexas.edu/hcb/files/2011/02/CCxG-October22-2010.pdf

[7] Greg Carlson, "Thematic Roles and the Individuation of Events" https://www.sas.rochester.edu/lin/people/faculty/carlson_greg/assets/pdf/them-roles-events.pdf

[8] Lucas Champollion, "Covert Distributivity in Algebraic Event Semantics" *Semantics & Pragmatics*, Volume 9 (2016), pages 1-65. https://semprag.org/article/view/sp.9.15

[9] _____, "Parts of a Whole: Distributivity as a Bridge Between Aspect and Measurement" Dissertation, University of Pennsylvania, 2010. https://repository.upenn.edu/cgi/viewcontent.cgi?article=2117&context=edissertations

[10] Robin Cooper, "Perception, Types and Frames" https://link.springer.com/chapter/10.1007/978-3-030-50200-3_8

[11] Bridget Copley and Heidi Harley, "A Force-Theoretic Framework for Event Structure" http://heidiharley.com/heidiharley/wp-content/uploads/2016/09/CopleyAndHarley2015Published.pdf

[12] Astrid De Wit and Frank Brisard, "A Cognitive Grammar Account of the Semantics of the English Present Progressive" https://www.researchgate.net/publication/326960775_Nominal_Grounding_Elements_in_English_A_Domain-based_Account

[13] Marie Dužı́ and Michal Fait, "Integrating Special Rules Rooted in Natural Language Semantics into the System of Natural Deduction" https://www.scitepress.org/Papers/2020/93696/93696.pdf

[14] Jason Eisner and Noah A. Smith, "Favor Short Dependencies: Parsing with Soft and Hard Constraints on Dependency Length" https://www.cs.jhu.edu/~jason/papers/eisner+smith.iwptbook10.pdf

[15] Daniel Gildea and David Temperley, "Optimizing Grammars for Minimum Dependency Length" https://www.cs.rochester.edu/~gildea/pubs/gildea-temperley-acl07.pdf

[16] Juneki Hong and Jason Eisner, "Deriving Multi-Headed Planar Dependency Parses from Link Grammar Parses". https://www.cs.jhu.edu/~jason/papers/hong+eisner.tlt14.paper.pdf

[17] Stefan Müller and Stephen Wechsler, "Lexical Approaches to Argument Structure" https://hpsg.hu-berlin.de/~stefan/Pub/arg-st.pdf

[18] Kobeen Raouf Mustafa and Azad Hasan Fatah, "Nominal Grounding Elements in English: A Domain-based Account" https://www.researchgate.net/publication/326960775_Nominal_Grounding_Elements_in_English_A_Domain-based_Account

[19] Jean Mark Gawron, "Circumstances and Perspective: The Logic of Argument Structure" https://escholarship.org/content/qt7sd5987t/qt7sd5987t_noSplash_14d78590a8657a71ed9ad46dc6e69064.pdf

[20] _____, "Paths and the Language of Change" https://gawron.sdsu.edu/paths_change.pdf

[21] Chris Genovesi, "Metaphor and What is Meant: Metaphorical content, what is said, and contextualism" https://www.tcd.ie/slscs/assets/documents/Clinical-speech/genovesi_2019_metaphor.pdf

[22] Carlos Gómez-Rodríguez and Ramon Ferrer-i-Cancho "The Scarcity of Crossing Dependencies: a direct outcome of a specific constraint?" https://arxiv.org/pdf/1601.03210.pdf

[23] Robert F. Hadley, "A Default-Oriented Theory of Procedural Semantics" https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1301_4

[24] Daniel W. Harris, "Speech Act Theoretic Semantics". Dissertation, CUNY, 2014. https://academicworks.cuny.edu/cgi/viewcontent.cgi?article=1221&context=gc_etds

[25] Darrin Louis Hindsill, "Its a Process *and* an Event: Perspectives in event semantics". Dissertation, University of Amsterdam, 2007. https://eprints.illc.uva.nl/id/eprint/2058/1/DS-2007-03.text.pdf

[26] Kenneth Holmqvist, "Conceptual Engineering: Implementing cognitive semantics", in Jens Allwood and Peter , eds., *Cognitive Semantics*, pp 153 - 171, Amsterdam, Philadelphia: John Benjamins, 1999.

[27] Kenneth Holmqvist, "Implementing Cognitive Semantics: Image schemata, valence accommodation, and valence suggestion for AI and computational linguistics". PhD thesis, Dept.

[28] Laurence R. Horn and Istvan Keckes Peirce, "Pragmatics, Discourse, and Cognition" http://www.albany.edu/faculty/ikecskes/files/Horn%20and%20Kecskes%2065-45-R1026-Horn.pdf

[29] Werner Kallmeyer, "Verbal Practices of Perspective Grounding" https://d-nb.info/120819738X/34

[30] Kyle Johnson, "What Makes a Theta-role" https://www.qmul.ac.uk/sllf/media/sllf-new/department-of-linguistics/hagit-borer-celebration/Johnson.pdf

[31] Mark Johnson, "The Body in the Mind: The Bodily Basis of Meaning, Imagination, and Reason". University of Chicago Press, 1990

[32] George Lakoff and Mark Johnson, *Philosophy in the Flesh: the Embodied Mind and its Challenge to Western Thought*. New York: Basic Books, 1999.

[33] George Lakoff and Mark Johnson, *Metaphors We Live By*. University of Chicago, 1980.

[34] Ronald Langacker, *Cognitive Grammar: A Basic Introduction*. Oxford, 2008.

[35] Ronald Langacker, "Cognitive (Construction) Grammar". https://www.degruyter.com/document/doi/10.1515/COGL.2009.010/html?lang=en

[36] _____, "Constructions in Cognitive Grammar". https://www.jstage.jst.go.jp/article/elsj1984/20/1/20_1_41/_pdf/-char/en

[37] _____, *Grammar and Conceptualization* de Gruyter, 1999).

[38] _____, "How to Build and English Clause". https://oaji.net/articles/2016/3124-1458506162.pdf

[39] _____, "Evidentiality in Cognitive Grammar". https://benjamins.com/catalog/pbns.271.02lan

[40] _____, "The English Present: Temporal coincidence vs. epistemic immediacy". https://benjamins.com/catalog/hcp.29.06lan

[41] Gianluca E. Lebani and Alessandro Lenci, "A Distributional Model of Verb-Specific Semantic Roles Inferences" https://colinglab.humnet.unipi.it/wp-content/uploads/2012/12/Lebani-Lenci-2018.pdf

[42] Joseph Lehmann, *et. al.*, "Age-Related Hearing Loss, Speech Understanding and Cognitive Technologies". https://link.springer.com/article/10.1007/s10772-021-09817-z

[43] Sander Lestrade, "The Space of Case". https://repository.ubn.ru.nl/bitstream/handle/2066/82611/82611.pdf?sequence=1&isAllowed=y

[44] Claudia Maienborn, "Event Semantics". https://www.researchgate.net/publication/236898346_Event_semantics

[45] Ryan McDonald, *et. al.*, "Non-projective Dependency Parsing using Spanning Tree Algorithms". https://ryanmcd.github.io/papers/nonprojectiveHLT-EMNLP2005.pdf

[46] Melanie Mitchell, *Artificial Intelligence: A Guide for Thinking Humans*. Macmillan, 2019.

[47] Erwan Moreau, "From Link Grammars to Categorial Grammars" https://hal.archives-ouvertes.fr/hal-00487053/document

[48] Amy Neustein and Nathaniel Christen, *Covid, Cancer, and Cardiac Care*. Elsevier, 2022.

[49] Péter Pelyvás, "Epistemic Modality: A choice between alternative cognitive models". https://litere.uvt.ro/publicatii/BAS/pdf/no/bas_2011_articles/22%20247-259.pdf

[50] _____, "On Epistemic and Deontic Grounding". https://argumentum.unideb.hu/2019-anyagok/special_issue_I/pelyvasp.pdf

[51] _____, "The Development of the Grounding Predication: Epistemic Modals and Cognitive Predicates". https://brill.com/display/book/9780585474267/B9780585474267_s010.xml

[52] Jean Petitot, "The Morphodynamical Turn of Cognitive Linguistics". https://journals.openedition.org/signata/549

[53] Jean Petitot and René Doursat, *Cognitive Morphodynamics: Dynamical Morphological Models of Constituency in Perception and Syntax*. Peter Lang, 2011

[54] Emily Pitler, *Models for Improved Tractability and Accuracy in Dependency acy in Dependency Parsing*. Dissertation, University of Pennsylvania, 2013. https://repository.upenn.edu/cgi/viewcontent.cgi?article=2070&context=edissertations.

[55] James Pustejovsky, "The Syntax of Event Structure". https://www.cs.rochester.edu/u/james/Papers/Pustejovsky-event-structure.pdf.

[56] Jae-Young Shim and Samuel David Epstein, "Two Notes on Possible Approaches to the Unification of Theta Relations" https://jaeyoungshim.weebly.com/uploads/1/2/3/1/12312623/linguistic_analysis_two_notes_jae-young_shim.pdf

[57] Gerold Schneider, "A Linguistic Comparison of Constituency, Dependency and Link Grammar". Zurich University, Diploma, 2008. https://files.ifi.uzh.ch/cl/gschneid/papers/FINALSgeroldschneider-latl.pdf

[58] Daniel D. Sleator and Davy Tamperley, "Parsing English with a Link Grammar". https://www.link.cs.cmu.edu/link/ftp-site/link-grammar/LG-IWPT93.pdf

[59] Michael Spranger, "Incremental Grounded Language Learning in Robot-Robot Interactions". http://www2.ece.rochester.edu/projects/rail/mlhrc2015/papers/mlhrc-rss15-spranger.pdf

[60] Michael Spranger, "Procedural Semantics for Autonomous Robots -- A Case Study in Locative Spatial Language". https://www.researchgate.net/publication/280830547_Procedural_Semantics_for_Autonomous_Robots_-_A_Case_Study_in_Locative_Spatial_Language

[61] _____, "The Evolution of Grounded Spatial Language". https://langsci-press.org/catalog/book/53

[62] Michael Spranger, *et. al,*, "Open-ended Procedural Semantics". https://martin-loetzsch.de/publications/spranger12openended.pdf

[63] Sune Vork Steffensen and Alwin Fill, "Ecolinguistics: The state of the art and future horizons" https://findresearcher.sdu.dk/ws/files/87218550/Steffensen_2014_Language_Sciences.pdf

[64] Chukwualuka Michael Uyanne, *et. al.*, "Ecolinguistic Perspective: Dialectics of Language and Environment" https://ezenwaohaetorc.org/journals/index.php/ajells/article/viewFile/5-1-2014-010/61

[65] Orlin Vakarelov, "Pre-cognitive Semantic Information". https://link.springer.com/article/10.1007/s12130-010-9109-5

[66] _____, "The Cognitive Agent: Overcoming informational limits" https://philarchive.org/archive/VAKTCAv1

[67] Linas Vepstas and Ben Goertzel, "Learning Language from a Large (Unannotated) Corpus" https://arxiv.org/pdf/1401.3372.pdf

[68] Himanshu Yadav, *et. al.*, "Are Formal Restrictions on Crossing Dependencies Epiphenomenal?" http://socsci.uci.edu/~rfutrell/papers/yadav2019formal.pdf

[69] Jordan Zlatev, "Embodiment, Language, and Mimesis" https://lucris.lub.lu.se/ws/portalfiles/portal/4514085/1044802.pdf

[70] _____, "Phenomenology and Cognitive Linguistics" https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ebc99d9442497d36be7477f393dcc38d2d53ab5b