A: Lambda calculus

This appendix briefly reviews lambda calculus. It is not a general or comprehensive introduction to the topic. The material covered relates to the main body of the text and they are used in it frequently.

Lambda terms (equivalently, λ-terms) are well-formed lambda expressions. They are recursively defined as follows.

λ-words are constructed from the alphabet

$x, y, z, \cdots$ for variables,

$1, 2, a', b'$ for invariables (constants),

λ for abstractor (lambda binding),

(,) for grouping (parentheses).

λ-terms are the set Λ such that

variables and invariables are in Λ,

if $M \in \Lambda$ then $(\lambda x.M) \in \Lambda$ where x is an arbitrary variable,

if $M, N \in \Lambda$ then $(MN) \in \Lambda$.

By convention, we write multiple lambda bindings with a single dot: $\lambda x.\lambda y.xy$ is written as $\lambda x\lambda y.xy$.

Also by convention, lambda bindings associate to the right, and juxtaposition associates to the left. $\lambda x\lambda y\lambda z.xyz$ is same as $\lambda x(\lambda y(\lambda z((xy)z)))$.

A variable is free if it is not in the scope of its lambda binding, bound otherwise. For example, x is free in $x+2$, $\lambda y.x$ and $(\lambda x(a'b'))x$. It is bound in $\lambda x.x$ and in $\lambda x\lambda y.xy$. Within the inner body of the last lambda term, xy, both variables are said to have free occurences because there is no lambda binding in the body.

Lambda conversions are operations that denote equivalences among lambda terms. When used in the direction of eliminating a lambda binding, they are called *reductions*. If a lambda is introduced they are called *abstractions*.

The conversions rely on the property of substitution for bound variables. Eta conversion shows the behavioral equivalence of the typed objects with and without variables. Beta conversion is the main mechanism to establish function application and function abstraction as two sides of the same coin. Alpha conversion shows the equivalence of bound variables under substitution. Together they define equivalence in the function treatment of lambda calculus.

substitution $M[a/x]$ stands for substituting a for free occurrences of x in M.

η-conversion $\lambda x.fx =_\eta f$, if x is not free in f.

β-conversion $\lambda x.M(a) =_\beta M[a/x]$

α-conversion $\lambda x.M =_\alpha \lambda y.(M[y/x])$, if scopes of variables in $\lambda x.M$ and $\lambda y.(M[y/x])$ are the same.

equivalence $M = N$ iff $Ma =_{\alpha,\beta,\eta} Na$, for all lambda-terms M, N, a.

Read '=' as 'behaves the same', not as 'identical'. From substitution and beta reduction, we get $\lambda x.fx(a) =_\beta fx[x/a]$, which is the same as fa, hence the association of beta with function application and abstraction. By equivalence, $\lambda x.fx = f$ too, hence the same behavior when f is supplied with a. The condition on eta conversion ensures that we do not change the behavior of objects; $\lambda x.(\lambda y.yx)x$, in which x is free in $\lambda y.yx$, is not equivalent to $\lambda y.yx$. Similarly, the condition on alpha conversion avoids an accidental capture of the same names, for example $\lambda x.y \neq_\alpha \lambda y.y$, and $\lambda x\lambda y.xy \neq_\alpha \lambda y\lambda y.yy$.

Normalization refers to the successive application of a conversion until it no longer applies. For example, the beta normalization of $(\lambda x\lambda y.f'yx)(a')(b')$ is two applications of beta reduction giving $f'b'a'$. The eta normalization of $\lambda x\lambda y.f'yx$ is f'. Some lambda terms have no normal forms because the process may not always terminate: $(\lambda x.xx)(\lambda x.xx)$ has no beta normal form.

Normal-order evaluation of a lambda term is the application of beta reduction to the leftmost outermost reducible expression (redex) first. In $(\lambda x.x)((\lambda y.y)a')$ there are two redexes, and normal-order chooses to reduce it to $(\lambda y.y)a'$, i.e. the application of the second one, without evaluation, to the leftmost redex. The Church-Rosser theorem establishes the result that two distinct sequences of reductions from the same lambda term will yield the same normal form if there is one. For the example above, it is $a.'$

B: Combinators

This appendix covers some mathematical aspects of combinators. Much of the book is about turning combinators into linguistic devices for explanation. These aspects are covered in the main body of the text.

Combinators are lambda terms with no free variables. As such they epitomize the compositional behavior of functional objects without a need for variables. By a convention going back to Curry and Feys (1958) they are written as single letters in bold. No extra notation is needed to describe their behavior. The ones considered most basic are defined below. The names were given by Curry and Feys.

$\mathbf{B} \stackrel{def}{=} \lambda f \lambda g \lambda x.f(gx)$ (compositor)

$\mathbf{S} \stackrel{def}{=} \lambda f \lambda g \lambda x.fx(gx)$ (substitutor)

$\mathbf{C} \stackrel{def}{=} \lambda f \lambda g \lambda x.fxg$ (elementary permutator)

$\mathbf{T} \stackrel{def}{=} \lambda f \lambda g.gf$ (commutator)

$\mathbf{W} \stackrel{def}{=} \lambda f \lambda x.fxx$ (duplicator)

$\mathbf{K} \stackrel{def}{=} \lambda f \lambda g.f$ (cancellator)

For example, $\lambda x.a'xx$ is equivalent to $\lambda x.\mathbf{W}a'x$, which is eta-normalizable to $\mathbf{W}a'$.

Combinators established computability about a decade before Turing machines. Their equivalent power can be seen without proof: **K** can delete any sequential material, **S** can expand and compose sequences, **C** can swap their order, which are the basic mechanisms that give the Turing machines their power. In this sense the Turing model is a formal specification of an algorithm in detail, and combinators are its global compositional view.

Normal-order evaluation of combinators evaluates the leftmost outermost combinator first. For example,

$\mathbf{BSC}fga = \mathbf{S}(\mathbf{C}f)ga = \mathbf{C}fa(ga) = f(ga)a$

As in the case of lambda calculus, the process may be nonterminating: **WWW** evaluates to itself indefinitely.

For the sake of completeness, I list the well-known combinators in Table 7. The names in the table are from Smullyan's (1985) tale of combinators as singing birds. They are in common use as well.

As Curry, Feys and Smullyan note, there are many equivalences between the combinators. This aspect opens way to linguistic theorizing about which must be included in the grammar or in the lexicon, therefore they belong to the main body of the book.

Table 7. Some well-known combinators

I	$\mathbf{I}x = x$	Identity bird
Y	$\mathbf{Y}x = y = xy$ for some y depending on x	Sage bird
U	$\mathbf{U}xy = y(xxy)$	Turing bird
K	$\mathbf{K}xy = x$	Kestrel
T	$\mathbf{T}xy = yx$	Thrush
W	$\mathbf{W}fx = fxx$	Warbler
B	$\mathbf{B}xyz = x(yz)$	Bluebird
C	$\mathbf{C}xyz = xzy$	Cardinal
S	$\mathbf{S}xyz = xz(yz)$	Starling
Φ	$\mathbf{\Phi}xyzw = x(yw)(zw)$	
Ψ	$\mathbf{\Psi}xyzw = x(yz)(yw)$	
J	$\mathbf{J}xyzw = xy(xwz)$	Jay

The power of a combinator is a generalization of its behavior. For example, $\mathbf{B}^n f$ composes f with n-argument functions, whereas **B** composes two one-argument functions. It is defined as follows:

$X^0 = \mathbf{I}$,

$X^1 = X$,

$X^n = \mathbf{B}XX^{n-1}$ for $n > 1$, for a combinatory object X.

Therefore, $\mathbf{B}^2 fgab = \mathbf{BBB}fgab = \mathbf{B}(\mathbf{B}f)gab = \mathbf{B}f(ga)b = f(gab)$. Powers are not distinct combinators, and they serve a crucial role in generalizing the linguistic notion of arity.

A supercombinator is a combinator in normal form in which all its argument-taking lambdas (its lambda bindings) can be grouped to the left, i.e. its behavior can be made fully transparent looking from the outside. The formal definition is as follows (from Hughes 1984):

Let $\mathscr{S} = \lambda x_1 \cdots \lambda x_n.E$ where E is not a lambda abstraction. $\mathscr{S}$ is a supercombinator of arity n if (a) $\mathscr{S}$ is a combinator, (b) any lambda abstraction in E is a supercombinator, and (c) $n \geq 0$.

In other words, if we can group all bindings before E, and leave no free variables inside E which must be remembered—bound—outside, then we have a supercombinator. Almost all the combinators we have seen so far are supercombinators, but not all combinators are supercombinators. The function $\lambda y.y(\lambda x.yx)$ is not a supercombinator because y occurs free in the inner lambda term. Supercombinators will directly relate to the argument-taking behavior of the linguistic notion of 'head of a construction'.

Fixpoint combinators stand out of supercombinators because they allow us to capture recursion without use of names or variables. One such combinator is **Y**. Its

definition is given below. Note that **Y** is not a supercombinator. It finds the fixpoint of any function h, as shown.

$\mathbf{Y} \stackrel{\text{def}}{=} \lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x))$

$\mathbf{Y}\,h = h\,(\mathbf{Y}\,h)$

It is truly remarkable that with the use of **Y**, recursion can be achieved without names. I borrow from a classic in the field of programming, Peyton Jones (1987: §2.4), to tell the story.[108]

Consider the following definition of the factorial function, where recursion is explicit due to naming (which is something we cannot do in lambda calculus).

$\text{FAC} = \lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (\text{FAC}\ (-\ n\ 1)))$

This recursive definition can be turned into self-application without recursion as below, because of beta conversion. Note that H is not recursive.

Let $\text{H} = \lambda f \lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (f\ (-\ n\ 1)))$
Then $\text{FAC} =\ \text{H FAC}$
because $\text{FAC}\,n =_\beta\ \text{H FAC}\,n$ for any natural number $n \geq 0$

The point of course is to able to recurse without names on any function, not just the factorial. This is where the combinator **Y** can help. The factorial can be defined without recursion or names. The steps below are borrowed from Peyton Jones (1987: 27). They show that it does the equivalent of the recursive factorial.

FAC = **Y** H, where H is as defined above.
$\text{FAC}\ 1 =$
$\mathbf{Y}\ \text{H}\ 1 =$
$\text{H}\ (\mathbf{Y}\ \text{H})\ 1 =$
$\lambda f \lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (f\ (-\ n\ 1)))\ (\mathbf{Y}\ \text{H})\ 1 =$
$\lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (\mathbf{Y}\ \text{H}\ (-\ n\ 1)))\ 1 =$
$\text{IF}\ (=\ 1\ 0)\ 1\ (\times\ 1\ (\mathbf{Y}\ \text{H}\ (-\ 1\ 1))) =$
$\times\ 1\ (\mathbf{Y}\ \text{H}\ 0) =$
$\times\ 1\ (\text{H}\ (\mathbf{Y}\ \text{H})\ 0) =$
$\times\ 1\ \ ((\lambda f \lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (f\ (-\ n\ 1))))\ (\mathbf{Y}\ \text{H})\ 0) =$
$\times\ 1\ \ ((\lambda n.\ \text{IF}\ (=\ n\ 0)\ 1\ (\times\ n\ (\mathbf{Y}\ \text{H}\ (-\ n\ 1))))\ 0) =$
$\times\ 1\ \ (\text{IF}\ (=\ 0\ 0)\ 1\ (\times\ 0\ (\mathbf{Y}\ \text{H}\ (-\ 0\ 1)))) =$
$\times\ 1\ 1 =$
1

The problematic property of **Y** is that it cannot be reduced to a form which cannot be reduced any further, thus the only way to stop recursion by **Y** is to reach nonrecursive (base) cases, such as reaching the '$\times$ 1 1' step above.[109] Below is **YK**'s infinite expansion. The base cases are an infinite supply of semantic objects following **YK**.

$\mathbf{YK} = \mathbf{K}(\mathbf{YK}) = \mathbf{K}(\mathbf{K}(\mathbf{YK})) = \mathbf{K}(\mathbf{K}(\mathbf{K}(\mathbf{YK}))) = \cdots$

In the book I follow the convention of writing a syntacticized combinator with its arity as a prefixed subscript. The subscript will be omitted when the arity is same as its combinatory definition, for example 2 for **T** and **K**, 3 for **B**, **S** etc. Curry and Feys (1958) use the notation $(X)_n$ for the same purpose where n is the arity, but the use of

parentheses for that purpose is somewhat unfortunate because they do so much work on the right-hand side of the definitions. Other options such as X_n, X^n, $X_{(n)}$, $X_{[n]}$ are used for other purposes by Curry and Feys (1958). I note the convention for easy reference:

For a combinatory object X,
its arity k in a particular use is denoted as ${}_kX$. (arity-in-use)
Arity is omitted when it is the same as in X's definition.
For example, ${}_2\mathbf{T}$ is same as $\mathbf{T}$. ${}_2\mathbf{B}$ is binary use of ternary $\mathbf{B}$.

C: Variable elimination

There is nothing any theory can do if a variable is free to vary. The process of variable elimination therefore relates to bound variables. It can be done in various way, as Frege, Schönfinkel, Geach and Quine have shown. This appendix is concerned only with the possibility of variable elimination. (The manner in which it is done bears on linguistic theory, and is dealt with in the main body.)

First we note that if all bound variables of a function symbolize the applicative behavior of the function, i.e. if they are used in the order they are lambda-bound, and only once, then eta conversion can do all the work, as follows.

$\lambda x_1 \cdots \lambda x_{n-1} \lambda x_n . f x_1 \cdots x_{n-1} x_n$ equals, by associativity, to

$\lambda x_1 \cdots \lambda x_{n-1} \lambda x_n ((f x_1 \cdots x_{n-1}) x_n) =_\eta$

$\lambda x_1 \cdots \lambda x_{n-1} . f x_1 \cdots x_{n-1} =_\eta$

$\vdots$

$\lambda x_1 . f x_1 =_\eta$

f

Therefore eta conversion is equivalent to saying that all semantic invariants are inherently typed. Once we know that f is say a three-argument function with applicative behavior, then writing f^3 or just f is sufficient.

The rest of the dependencies, for example $\lambda x . f x x$ or $\lambda x . f(g x)$, are not eta-normalizable without the help of combinators. For example, the first one of these is eta-normalize as $\lambda f . x x = \lambda f . \mathbf{W} f x =_\eta \mathbf{W} f$ and the second $\mathbf{B} f g$. Schönfinkel's work showed that two suffice for this task, because $\mathbf{S}$ can be seen as a mechanism of pushing the lambda bindings inside, which will eventually reach a base case such as $\lambda x . x$, $\lambda x . y$ or $\lambda x . a'$, which are lambda terms with the simplest body of functions. These properties follow from the following equivalences.

$(\lambda x . M N) a =_\beta \mathbf{S} (\lambda x . M)(\lambda x . N) a$

hence $(\lambda x . M N) = \mathbf{S} (\lambda x . M)(\lambda x . N)$ from equivalence in lambda calculus.

The elimination is completed by the following equivalences:

$\lambda x . y = \mathbf{K} y \quad \lambda x . a' = \mathbf{K} a'$

$\lambda x . x = \mathbf{I}$

The equivalences are applicable to any lambda-definable (hence Turing computable) object. For example, $\lambda x . M N P$ is equivalent to $\lambda x . (M N) P$ because of left-associativity, thus any number of lambda terms can be handled by $\mathbf{S}$. In case of multiple abstractions such as $\lambda x \lambda y . love' x y$, we have to apply $\mathbf{S}$-pushing to the innermost lambda first.

Knowing that $\mathbf{I} = \mathbf{SKK}$, we can eliminate all bound variables and write everything in terms of $\mathbf{S}$, $\mathbf{K}$ and the invariables. For example, everything except hit' and $john'$ can be eliminated from the following formula.

$\lambda x . hit' x john' =$

$\mathbf{S}(\lambda x.hit'x)(\lambda x.john') =$
$\mathbf{S}(\mathbf{S}(\lambda x.hit')(\lambda x.x))(\mathbf{K}john') =$
$\mathbf{S}(\mathbf{S}(\mathbf{K}hit')\mathbf{I})(\mathbf{K}john')=$
$\mathbf{S}(\mathbf{S}(\mathbf{K}hit')(\mathbf{SKK}))(\mathbf{K}john')$

This is a dangerous practice because of **K**'s powers of deletion. The reader can verify that the following formula works endlessly to reproduce itself, due to having both **S** and **K**. Some steps are shown.

$\mathbf{SS}(\mathbf{KI})(\mathbf{SS}(\mathbf{KI}))(\mathbf{SS}(\mathbf{KI})) =$
$\mathbf{S}(\mathbf{SS}(\mathbf{KI}))(\mathbf{KI}(\mathbf{SS}(\mathbf{KI})))(\mathbf{SS}(\mathbf{KI})) =$
$\mathbf{SS}(\mathbf{KI})(\mathbf{SS}(\mathbf{KI}))(\mathbf{KI}(\mathbf{SS}(\mathbf{KI}))(\mathbf{SS}(\mathbf{KI}))) =$
$\vdots$
$\mathbf{SS}(\mathbf{KI})(\mathbf{SS}(\mathbf{KI}))(\mathbf{I}(\mathbf{SS}(\mathbf{KI}))) =$
$\mathbf{SS}(\mathbf{KI})(\mathbf{SS}(\mathbf{KI}))(\mathbf{SS}(\mathbf{KI}))$
$\vdots$

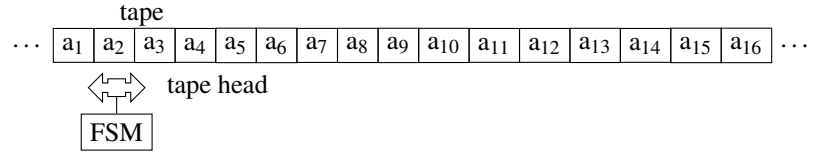
We might go one step further and derive **S** and **K** from Barendregt's (1984) combinator **X**, but not without some circularity. Take $\mathbf{X} = \lambda x.x\mathbf{KSK}$. Then $\mathbf{XXX} = \mathbf{K}$, and $\mathbf{X}(\mathbf{XX}) = \mathbf{S}$. The bottom line is, if we want complete elimination of variables, we need the **S** and **K** somehow; witness **KSK** in **X**.

Further optimizations in reducing the sequence of combinators in this process are possible, for example using **BCS** or **CDΦ** to eliminate the unnecessary proliferation of **S** abstractions; see Curry and Feys (1958: 188ff), Turner (1979).

D: Theory of computing

The theory of computing features quite often in the book because it has empirical and theoretical consequences for combinatory linguistics. In the first aspect, the children seem to be facing a computationally tractable problem in language acquisition and stagewise development. Granted that there have been some warnings about using the algorithmic complexity theory at face value for this task (e.g. Berwick and Weinberg 1982), the narrower claim of the theory covered in the book is that a performance grammar is competence grammar because it delivers the immediate assembly of all constituents and their meanings, partial or full. Theoretically, another aspect of algorithmic computation seems very relevant to natural language: discrete representability, without which complexity theory is meaningless. Because Turing (1936) was the first to give us a view of functions unheard of before, as a step-by-step computing over a representation, I will refer to it as Turing representability. The appendix covers these aspects very briefly from a mathematical perspective.

A Turing Machine (TM) is a finite-state abstract machine with an unlimited supply of sequential memory (usually called "tape") to which it can write, rewrite and scan one cell at a time:



A tape cell may contain a symbol a_i or it may be blank. The FSM is the finite-state machine component. Before computation starts, a TM is in the start state of the FSM with the tape head pointing to the beginning of the input, if any, and the remaining cells are assumed to be blank. It stops when it reaches a "halt state" in the FSM (there are many alternative definitions; this one suffices for our purpose of computing a function).

It has no notion of physical timing; its measure of a problem size is a combination of the number of states and the number of steps it takes to compute a function. We can assume that every basic step (read, write, rewrite or move left or right on the current tape head, and/or change the current state in the FSM) takes a constant time, but that is in theory unnecessary; it might as well happen simultaneously. What matters is that taking the next step requires the notion of "next", and that is either one cell, one symbol or one state, so that once we take the step we will have moved one step more than the earlier status in some regard above. These are the bases of complexity measures in the theory of algorithms.

A configuration of a TM is a collection of its current state, the current pointer to a cell in the sequential memory, and its memory content. Trailing blank cells are

not considered part of the content. Memory content can of course be indefinitely stretched, which we can capture as a regular expression.

A Deterministic Turing Machine (DTM) is a TM in which every configuration is uniquely determined by the previous one. This is Turing's capture of the notion of function in a step-by-step manner. If there is more than one way to take the steps of the function, a DTM can simulate these choices by making use of another tape to keep track of its moves while checking whether they all agree on the result, which we can keep on yet another tape. This is Turing's capture of the notion of relation, which is a function over powersets of inputs and outputs. We know that multi-tape Turing machines and other variations such as multiple tape heads, nondeterminism, random-access memory rather than the tape do not gives us more things to compute than a standard TM (see Hopcroft and Ullman 1979, Lewis and Papadimitriou 1998 for these results).

A TM is said to be **nondeterministic** (NDTM) if it can make a "guess" of the solution and check (as a DTM) whether it is indeed a solution. We can take the guessing stage to be equivalent to putting on another tape the precise sequence of steps to follow. In this regard we do not get a new class of computation but a new class of how to do computing, i.e. a complexity measure.

An algorithm is a DTM that always **decides**, i.e. if it can stop for any input to make a decision. A nondeterministic algorithm does the same with a NDTM. A **procedure** (or heuristic) is a DTM or NDTM that **semi-decides**, i.e. if they can stop on some input to make a decision.

Undecidable problems are "functions" for which there is no algorithm (deterministic or nondeterministic). The Halting Problem of the Turing machine in which a TM takes as input another TM and tries to decide whether it stops on all its inputs, is one such problem. The problem is at least formulable, but it is not solvable. Some problems are expressible but not formulable, for example: "what is the next number after π?"

In the book a problem will be called **Turing-representable** if it can be formulated as a Turing machine (but not necessarily solved by it). For example, the halting problem is Turing-representable as below (from Lewis and Papadimitriou 1998). It is the *diagonal(diagonal)* program. The π question is not Turing-respresentable.

diagonal(X):

a: if *halts(X,X)* then goto *a* else halt.

Turing-representability ties in with another line of development that gave us the understanding of limits of computability today: the recursion theory. **Primitive recursive functions** are those which can be defined by identity, succession, composition and recursion. The successor function $succ(n) = n + 1$ is crucial in this definition, which gives us the link to Turing-representability by providing a notion of "next".

A computationally tractable problem is one for which there is an algorithm that works on a polynomial function of the size of the problem for a DTM (i.e. its number of states, the number of steps it must take and the space it must use, as a function of

the Turing-representable input). The complexity class $\mathscr{P}$ symbolizes such problems. Computing scientists sometimes use the term "polynomial time function" to talk about these problems, and care must be taken not to misunderstand the word *time*. It does not measure the physical time or space but abstract time and abstract space, which are the abstract measures of problem "size" from Turing representations. (In this sense computation as we know today cannot be a natural law as Chomsky once suggested.)

A computationally intractable problem is one for which there is a NDTM that can guess a solution and check its validity in polynomial time. This is a very important class of complexity, called $\mathscr{NP}$, for "nondeterministically polynomial". Intractable problems, then, have an exponential algorithmic solution, all of which can be checked in polynomial time individually.

The order of a function limits its behavior on the abstract size of the problem "from above." The order of f is g, written $f = O(g)$ by convention, if for some positive constants c and n_0, $f(n) \leq cg(n)$ for all $n > n_0$. If f is n^2 it is $O(n^3)$ and also $O(n^4)$ etc. It is $O(n^2)$ too, but n^4 is not $O(n^2)$ or $O(n^3)$. This notation allows us to equate $\mathscr{P}$ problems with $O(n^k)$ order, for some constant k, and the $\mathscr{NP}$ class with $O(k^n)$, where n is the problem size in the Turing sense.

Many interesting problems are $\mathscr{NP}$, e.g. finding the possibility of the truth of a set of disjunctive logical formulae such as $A_1 \vee A_2 \vee A_3$ and $\neg A_1 \vee A_2 \vee A_3$. If we are given the truth conditions of A_i, we can check in polynomial time whether the set is satisfiable (i.e. true in all its clauses). If not, we must check every truth assignment, which is exponential on the size of the set, therefore computationally intractable.

The fact that we do know this even if generating the entire solution space may wear us down relates the notion of Turing-representability, algorithms, competence and performance at the abstract level rather than concrete. This is the significance of the theory of computing for linguistics. It is an intensional body of knowledge.

In this regard a computational look at language cannot be understood just by looking at problem complexity, timing or space through the classes $\mathscr{P}$ and $\mathscr{NP}$. The approach and these complexity classes are intrinsically tied to abstract and discrete representability, which translate to scaling up of the knowledge of competence and identifying similarly characterizable problems of cognition.

We may compare a computational solution with a noncomputational one to see the nature of the argument. Consider sorting n quantities. A noncomputational solution in the sense of avoiding a Turing representation might be to conceive them as physical quantities, say as weights and solidity. Sorting can be done with a variant of Dewdney's (1984) method, which is itself algorithmic and linear. We can take a sheaf of spaghetti cut to different lengths, where length represents itself, i.e. an approximation of the quantity along which we sort. We bang the sheaf on the table and pick the ones that stick out progressively. This is in principle instantaneous if we leave the sorted spaghettis in place rather than separate them. In contrast, a computational solution would be to map the quantities to some representation, say numbers,

and solve the problem as a case of sorting anything that has a discrete representation, which is $O(n \log n)$. In the first case we can claim to have understood gravity, solidity and eye measurement. In the second case we understand the nature of the problem. The first solution would not scale up even if we assume to have devised a representation of weights through spaghetti and tables because it is not translatable, it will not work in outer space, or for gases. We might search for a mapping of any problem so that gravity can solve it by natural laws, but in doing so we would be turning gravity into a computer, crucially one that works over a representation, which is the mapping itself. We can compare this approach to the original analog algorithm of Dewdney for spaghetti sort, which is indeed an algorithm therefore a computational solution because although it makes use of gravity to sort the spaghetti rods, it iterates on the broken spaghetties for sorting, hence its complexity measure is not the physical time associated with gravity but the number of steps. (I am grateful to Mark Steedman for suggesting a look at Dewdney.)

E: Radical lexicalization and syntactic types

Radical lexicalization refers to the process of rewriting all the rules in a phrase-structure grammar which do not make reference to a lexical item on the righthand side, as rules for the lexical items. These rules collectively become the lexical item's combinatory category. Two kinds of phrase-structure rules, context-free rules and linear-indexed rules, can always be given such a treatment.

A linear-indexed grammar is a context-free grammar equipped with a stack such that the lefthand side of rules can push, pop or pass the stack to the righthand side, and only to one symbol on the right (hence the term "linear"). Such grammars can generate strictly noncontext-free languages. For example, the grammar below generates $\{a^n b^n c^n \mid n \geq 0\}$ ('..' denotes the remainder of the stack '[ ]').

$\mathrm{S}[..] \rightarrow a\mathrm{S}[..\mathrm{b}]c$
$\mathrm{S}[..] \rightarrow \mathrm{A}[..]$
$\mathrm{A}[..\mathrm{b}] \rightarrow \mathrm{A}[..]b$
$\mathrm{A}[\ ] \rightarrow \varepsilon$

This appendix shows the radical lexicalization of a context-free grammar. Linear-indexed grammars are related to CCG hence covered in the main text.

Let us consider the following fragment of a context-free phrase-structure grammar to clarify this point. Exclusive terminals in the second column stand for the lexicon, and the grammar rules on the left refer to substantive categories S, NP, VP, V etc.

S	$\rightarrow$	NP VP	Det	$\rightarrow$	every
NP	$\rightarrow$	Name	N	$\rightarrow$	chemist
NP	$\rightarrow$	Det N	Name	$\rightarrow$	Kafka
VP	$\rightarrow$	V_{iv}	V_{iv}	$\rightarrow$	arrived
VP	$\rightarrow$	V_{tv} NP	V_{tv}	$\rightarrow$	adored

First, the information about arity is redundantly specified in this grammar. The rule VP $\rightarrow$ V_{tv} NP specifies that the verb is transitive because there must be an NP following the verb, and the lexical entry by the preterminal V_{tv} duplicates that information. We can take the rule to mean that a transitive verb, once it takes an NP to the right, yields a VP. That is, V_{tv}=*VP*/*NP* in present terms. We could also write *NP*=*VP*\V_{tv}=*(S\NP)\((S\NP)/NP)*, because from the S rule we can write *VP*=*S\NP*.

Similarly, V_{iv}=*VP*. Because the NP rules have lexical anchors in this grammar (name and determiner), we can follow the same strategy and arrive at *Det*=*NP*/*N* and *Name*=*NP*. We could also write *N*=*NP**Det* if we wished. The S rule has no lexical anchor, thus we must write it as both *NP*=*S*/*VP* and *VP*=*S**NP*. We have arrived at the following equivalences:

	V_{tv}=*VP*/*NP*	V_{iv}=*VP*	*NP*=*VP*\V_{tv}
	NP=*S*/*VP*	*VP*=*S**NP*	Det=*NP*/*N*
	Name=*NP*	*N*=*NP*\Det	
Hence	V_{tv}=*(S**NP)*/*NP*	V_{iv}=*S**NP*	
	NP=*(S**NP)**((S**NP)*/*NP)*		
	NP=*S*/*(S**NP)*		

We can eliminate the phrase-structure rules in the left column of the phrase-structure grammar above, and write only the lexical items with their new categories, to capture the same fragment of English surface syntax:

every	:=	Det = *NP*/*N* = *(S*/*(S**NP))*/*N*
chemist	:=	N = *NP*\Det = *NP**(NP*/*N)*
Kafka	:=	Name = *NP* = *S*/*VP*=*S*/*(S**NP)* and *(S**NP)**((S**NP)*/*NP)*
arrived	:=	*VP* = *S**NP*
adored	:=	*VP*/*NP* = *(S**NP)*/*NP*

What we cannot eliminate, of course, is the right column because that would change the empirical coverage of the grammar.

Any context-free phrase-structure grammar and linear-indexed grammar can be reduced to its lexicon if we are willing to translate the distributional categories such as N, V, A, P to combinatory categories as above. We can do this because any rule in these formalisms have one symbol on the left-hand side, with or without a stack, therefore a functional reading of the rule from right to left is always possible. (LIGs do not distribute a stack on the right, therefore the compositional reading of a LIG rule is straightforward too.) Notice also that the redundancy of V_{tv} specification has disappeared.

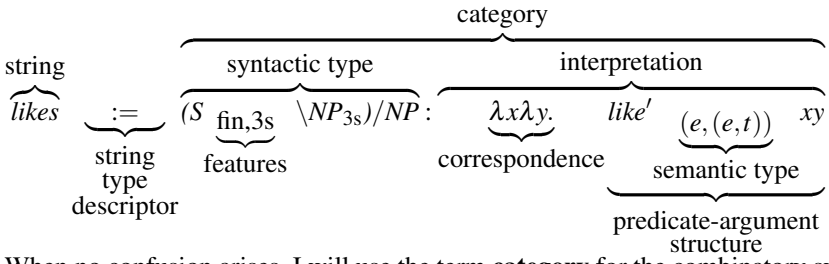
One can argue that the elimination of unwanted ambiguity leads to another ambiguity, viz. *NP=(S\NP)\((S\NP)/NP)* and *NP=S/(S\NP)*. We shall see in the text that the newly introduced ambiguity is not spurious; it relates to case marking.

A combinatory syntactic type can be thought of as a collection of the applicative translation of all phrase-structure rules as above, plus their combinatory derivatives. For example, from *S*/*(S**NP)* and *(S**NP)*/*NP* in this order we also get *S*/*NP* because of composition. They can be thought of as the possible landscape of all types derived from the lexical items as a closure of the lexicon on combinators. A linguistic theory will select a subset in some principled way.

An example type is shown below.

likes := $(S_{\text{fin,3s}} \backslash NP_{\text{3s}})/NP : \lambda x \lambda y.like' xy$

The breakdown of its constituents are as follows. Additionally, I use the common index *i* as a simple way to share the common features among syntactic types, for example *he* := $S_i/(S_i \backslash NP_{\text{3s}\in i})$. The *i* here is a set of features among which there is third-person singular agreement emanating from the *NP*. Feature abbreviations are also quite common in the book, to write NP_{3s} to mean $NP_{\text{AGR=3s}}$.



When no confusion arises, I will use the term **category** for the combinatory syntactic type.

A consequence of radical lexicalization is that one end of the rules for the lexical items is the syntactic type, and, since there is no other loci if lexicalization is strictly followed, then the other end has to be a predicate-argument structure, which bears the semantic types. I cover the consequences of this result in the main text.

A semantic type is a narrowing of a predicate-argument object in possible values. The type e is for things (Montague's *entity*), t is for propositions, and (e,t) is for predicates and properties, that is, for functions from things to propositions. For example the transitive verb *like*, with the semantics $\lambda x\lambda y.like'xy$, has the semantic type $(e,(e,t))$. The eta-normalized version *like*$'$ is assumed to carry this type along. Thus, *like*$'$ is not of type t, which its bare form might suggest. In that sense, every semantic object has a type.

F: Dependency structures

Dependency structures may be specified over words in a string in some theories and over predicate-argument structures in others. This topic belongs to an appendix because it can be done without combinators. With combinators, it is defined over a predicate-argument structure, and this narrow view is explained here.

A dependency structure is a relation between two semantic objects. For our purposes it can be defined as follows.

A function depends on its arguments. (dependency)

Juxtaposition *xy* means '*x* depends on *y*' (juxtaposition)

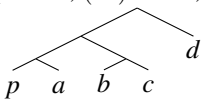
It arises from a functional interpretation of the concept. (I use a distinction, more commonly made in computer science and computational linguistics, between functions, predicates and algorithms. The term *function* refers to opaque properties, such as arity, dependence and output, whereas the term *predicate* refers to transparent properties such as the event class, argument structure and their obliqueness, although, formally speaking, they are both functions or relations. *Algorithms* are functions that do something. I will use this term when we are interested in the task of the function rather than its dependencies or structures.)

A predicate-argument dependency structure, abbreviated to **PADS** in the book, is a predicate-argument structure of dependencies where the leftmost element is a predicate. For example, $john'mary'$ is a dependency structure but not a PADS, and $sleep'john'$ is a dependency structure and a PADS. As we shall see throughout the book, PADS is different from the logician's logical form, from the transformational linguist's logical form, from the dependency structures between words of a string, and from model-theoretic objects. It is lexically determined and projected. It depends on the syntactic type in crucial ways, codetermines it in crucial ways, and it is indeed a nonassociative structure: $(hurt'love')mary'$ is different than $hurt'(love'mary')$. In the first case, $mary'$ cannot be construed to have an individual relation to the other elements. Likewise for $hurt'$ in the second case. The first one might arise from an expression such as *Mary thinks that love hurts*, and the second one from *Mary's love hurt John.*

For example, in $sleep'kafka,'$ $sleep'$ depends on $kafka.'$ In $like'milena'kafka,'$ there are two dependency relations: $like'$ depends on $milena,'$ and $like'milena'$ depends on $kafka.'$ These embeddings follow from the left-associativity of juxtaposition, which we can show as:

The relation can be abstracted over. For example $\lambda x.sleep'x$ abstracts over the argument of a dependency, and $\lambda f.fkafka'$ over the function.

We can take the tree above to signify the obliqueness of the arguments of the predicate in the prefix. We can say that a leaf node that c-commands another in the PADS is less oblique.[110] That is one of the reasons why we consider PADS to be a structure rather than a flat list. There would be no obliqueness relation for the leftmost element of a PADS. For example $slept'\,john'$ does not manifest obliqueness. We can also say that a predicate "sees" its arguments one at a time in PADS: the elements that c-command the leftmost element in its PADS are its arguments. In the example below, the arguments of p are a, (bc) and d, not a, b, c and d.



We shall see in the book a combinatory equivalent of arity and argument structure specification, without the need of another primitive such as c-command. Order and its semantics will be doing the work, rather than auxiliary assumptions. Notice that we have already obtained the result from juxtaposition that obliqueness relations are asymmetries; no two arguments can c-command each other in the notation $pred'\,arg'_1 arg'_2 \cdots arg'_n$.