But how can one formalise "expressions" as objects within mathematics (as contrasted against metamathematics)? In logic, the seminal tool for this is Gödel numbers, but with set theory in our toolbox we don’t have to be quite so obscure.

The first step is to formalise finite sequences of elements from a general set $X$; this is known as words on $X$. Technically we may encode a word $w$ as

$$w : \mathbb{Z}_{\geq 0} \rightarrow X,$$

where $\text{dom}(w)$ is finite and satisfies $k \in \text{dom}(w) \land j \in \mathbb{Z}_{\geq 0} \land j < k \Rightarrow j \not\in \text{dom}(w)$.

Practically, we use multiplicative notation for words, writing

$$w = x_1 x_2 \ldots x_n,$$

where $x_i \in X$ is the value of the function $w$ at $i$. The number $n$ of factors (elements of sequence) is called the length of the word, sometimes denoted $|w|$. There is a unique word of length $0$, the empty word, which we’ll denote $\lambda$ (other common notation is $\varepsilon$, especially in CS literature).

The set of all words on $X$ is denoted $X^*$ (for "$X$, any length") and is also known as the free monoid on $X$; it will be of great importance on its own later. As a monoid, the operation on $X^*$ is juxtaposition (concatenation):

If $u = x_1 \ldots x_m$ and $w = y_1 \ldots y_n$, then $uw = x_1 \ldots x_m y_1 \ldots y_n$.

Another useful operation on $X^*$ is reversal; with $w$ as above,

$$w^R = y_n \ldots y_1.$$

Implementation-wise, it is often convenient to let the elements of $X$ be individual characters, since $X^*$ is then
simply the set of strings in those characters — a datatype most languages support well. Even if multiple character sequences are necessary, it may still be preferable to encode words as strings with some custom decomposition into elements over encoding them as general lists (which would be the "pure" alternative).

In practice, the elements of \( X \) are identified with their length 1 counterparts in \( X^* \), by not making a notational distinction. This is usually trouble-free, but in implementations one may need to make the canonical injection \( X \rightarrow X^* \) explicit in the code.

The second (small) step is to form the disjoint union \( \sqcup \Omega \times X \) of a signature \( \Omega \) and a set of variables \( X \). Here we have three closely related concepts for which there actually are separate notations (as opposed to separate notations for the same thing):

- \( \sqcup \Omega \times X \): plain set-theoretic union of \( \Omega \) and \( X \).
  
  If \( \Omega \) is not disjoint from \( X \) then we’re in trouble.

- \( \sqcup \Omega \setminus X \): disjoint union; the same as \( \sqcup \Omega \times X \) value-wise, but also making the claim (side constraint) that \( \Omega \cap X = \emptyset \).

- \( \sqcup \Omega \sqcup X \): formally disjoint union, also known as coproduct in the categorical setting. \( \sqcup \Omega \sqcup X \) is technically not required to contain either of \( \Omega \) or \( X \) as subset, but there is a distinct element of \( \sqcup \Omega \sqcup X \) for each element of \( \Omega \), kifs for each element of \( X \), and each
Element of \( \Omega \cup X \) corresponds to either an element of \( \Omega \) or an element of \( X \). \( |\Omega \cup X| = |\Omega| + |X| \).

In using \( \Omega \cup X \), there is as for \( X^* \) technically a conversion of elements step which is usually skipped in notation by identifying elements of \( \Omega \cup X \) with their counterparts in \( \Omega \) and \( X \). We'll do the same.

The final step is to implement \( T(\Omega, X) \) as a subset of \( (\Omega \cup X)^* \). Cohn denotes this set \( W_\Omega(X) \) and all the set of \( \Omega \)-words on \( X \); more modern terminology would be right-Polish (or reverse Polish) notation \( \Omega \)-terms on \( X \).

(Cohn's intent is really to use \( W_\Omega(X) \) in the same way as Bader-Nipkow use \( T(\Omega, X) \), but since he gives a quite explicit construction of it, I choose to take \( W_\Omega(X) \) as an explicit encoding and \( T(\Omega, X) \) as the more abstract concept.)

The intuition behind right-Polish notation (RPN) is that expressions are sequences of tokens, read left-to-right by a stack-based interpreter. A token \( x \in X \) causes (the value of) \( x \) to be pushed onto the stack. A token \( f \in \Omega \) causes \( \alpha(f) \) operands \( t_1, \ldots, t_k \in B \) (where \( B \) is some \( \Omega \)-algebra) to be popped off the stack, after which the result \( f^\alpha(t_1, \ldots, t_k) \) is pushed back onto it. A token sequence is a well-formed term iff the required operands are always present on the stack and after reading the whole sequence the stack contains exactly one element.

This may be turned into compact formulae. Define \( \Delta : \Omega \cup X \to \mathbb{Z} \) by

\[
\Delta(s) = \begin{cases} 
1 & \text{if } s \in X, \\
-x(s) + 1 & \text{if } s \in \Omega 
\end{cases}
\]

Cohn calls this the "valency."
Then
\[ W_2(x) = \left\{ y_1 \ldots y_n \in (\Omega \cup \{x\})^n \left| \sum_{i=1}^k \Delta(y_i) > 0 \text{ for all } k = 1, \ldots, n \right. \right\}. \]

In particular,
\[ W_2(x) \supseteq x \cup \{ y \in \Omega \mid \alpha(y) = 0 \}. \]

Likewise if \( y \in \Omega \) and \( z_1, \ldots, z_n \in W_2(x) \) then \( z_1 \ldots z_n \in W_2(x) \).

In left-Polish (or just plain Polish) notation, the function symbol is instead in front of its operands (prefix notation), so we would have
\[ y \in \Omega \text{ and } z_1, \ldots, z_n \in W_2(x) \Rightarrow z_1 \ldots z_n \in W_2(x) \text{ (left-Polish notation)}. \]

Reversing a right-Polish notation term produces a left-Polish notation term (and vice versa), but does not in general preserve interpretation, since the operands end up in the opposite order. (One could make reversal an \( \Delta \)-algebra homomorphism by picking the opposite operations on one of \( W_2(x) \) and \( W_2^\text{left}(x) \).)

The name "Polish" for these things is because Łukasiewicz, who first published that this notation makes parentheses superfluous, was Polish.
Modularity of algebra construction

The generic construction of an algebra defined by generators and relations calls for gathering all sorts of equalities in one big bowl and then hoping to discover the congruence these generate—a huge undertaking. But most of those equalities have to do with the basic kind of algebra one is aiming for, so might it be possible to treat those equalities as a subsystem that can be prepared in advance—as half-finished goods to speed up production of the finished product? That is indeed the case. One kind of modularity is provided by the Third Isomorphism Theorem.

Third Isomorphism Theorem. Let \( A \) be an \( \Lambda \)-algebra and let \( Q, R \) with \( Q \subseteq R \) be two congruences on \( A \). Then

\[
\frac{A}{R} \cong \frac{A/\mathcal{Q}}{R/\mathcal{Q}} \quad \text{(isomorphism as \( \Lambda \)-algebras)}
\]

where

\[
R/\mathcal{Q} = \left\{ ([x]_\mathcal{Q}, [y]_\mathcal{Q}) \mid (x, y) \in R \right\}
\]

is the obvious \( \Theta([x]_\mathcal{R}) = [x]_\mathcal{Q} \mod{R/\mathcal{Q}} \).

Here \( A/R \) is the sought finished algebra, and \( A/\mathcal{Q} \) is the half-finished piece it will be produced from, at least up to isomorphism. The full picture is rather

\[
\frac{A}{R} \cong \frac{A/\mathcal{Q}}{R/\mathcal{Q}} \cong \frac{B}{R^1} \quad \text{where} \quad B \cong A/\mathcal{Q}.
\]
Depending on what one picks as $\varphi$, one can end up with very different things as $B$. The final quotient is where we apply the \textit{Diamond Lemma}, but technically, that is a family of results, each stated in terms of (and optimised for) one kind of $B$. In particular, the details of the algebra $B$ feature heavily in the conditions that one needs to verify, so we need to nail that down first, at least in cases of particular interest.

\textbf{Modularity of the signature}

In abstract algebra, there is often a separation of operations into “multiplication-like” and “addition-like”, or more rigorously: multilinear operations on the one hand, and operations that define the linear structure (addition and scaling) on the other. That separation can be turned into a fruitful modularisation of the signature, often to the point that it becomes effectively invisible.

Suppose $\varphi = \varphi_1 \cup \varphi_m$, where $\varphi_1$ consists of operations providing the linear structure and $\varphi_m$ consists of multilinear operations (such as binary multiplication, Lie bracket, Poisson bracket, or the like). Then more expansion (particularly distributivity) suffices for transforming any term in $T(\varphi, X)$ into an equivalent (with respect to laws of multilinearity) term in $T(\varphi_1, T(\varphi_m, X))$ (modulo the abuse of notation that we consider $Y$ to be a subset of $T(\varphi_1, Y)$). Moreover, the $\varphi_1$ part of these expressions can be simplified considerably — you don’t need to scale
sums, since that can always be expanded into a sum of
rescalings. You never need to rescale a rescaling, because
those can be combined into one rescaling. You don't
need more than one copy of any \( T(\Delta_m, X) \) term, because
those can be collected. We don't need to keep track of order
or brahcing of sums, because addition is commutative
and associative.

Taken together, these observations show that the
recursive \( telesc \) level of \( T(\Pi_0, T(\Delta_m, X)) \) can be
replaced by a flat, linear combination of \( T(\Delta_m, X) \) terms.

Construction. Let \( \mathbb{R} \) be an associative unital ring.
Let \( Y \) be any set. Then the set of formal \( \mathbb{R} \)-linear
combinations of \( Y \) is

\[
M = \left\{ \delta : Y \to \mathbb{R} \mid \delta'(\mathbb{R}\setminus\{0\}) \text{ is finite} \right\}.
\]

\( M \) is an \( \mathbb{R} \)-module under pointwise operations:

\[(\delta + \gamma)(\mu) = \delta(\mu) + \gamma(\mu) \text{ for all } \mu \in Y, \delta, \gamma \in M;\]

\[(r \delta)(\mu) = r \cdot \delta(\mu) \text{ to make it a left module and}\]

\[(f \delta)(\mu) = \delta(f(\mu)) \text{ to make it a right module. Either way, it is a free module with basis } \{1, \mu, x \alpha \text{ that consists of the characteristic functions}\}

\[1_{\mu}(x) = \begin{cases} 1 & \text{if } x = \mu, \\ 0 & \text{if } x \neq \mu \end{cases}\]

of the elements of \( Y \).

Notationally, the characteristic functions are often iden-
tified with their counterparts in \( Y \) when that does not
risk confusion, so general elements in \( M \) may be written as