Computational Semantics: Computing Meanings on Finite Structures

Marcus Kracht
Department of Linguistics, UCLA
3125 Campbell Hall
PO Box 951543
Los Angeles, CA 90095-1543
kracht@humnet.ucla.edu

May 25, 2008
Introduction

This course is about the computing the meanings of expressions, be they of a formal language or of a natural language. Thus, using the same method we shall explain how find the meaning of $(\emptyset=0) \lor (\emptyset=1)$ and the meaning of /All dogs are asleep./. Before we enter the details of the investigation, however, it is necessary to clarify both the direction we are going into and the method we are using.

Language is seen here as a set of pairs $\langle e, m \rangle$, where $e$ is an expression and $m$ a meaning. Expressions are strings over an alphabet. The alphabet can be freely chosen. Once the alphabet $A$ is fixed, an expression is a member of $A^\ast$, the set of strings over $A$. We allow the empty string $\epsilon$ to have meaning. Thus, with $M$ the set of meanings, $L \subseteq A^\ast \times M$.

Now, given an expression, we may ask: what is its meaning? This question is somewhat misleading as it suggests that there is only one. But there may be several meanings to a given expression. Indeed, for natural languages this is the rule rather than the exception. Conversely, given a meaning we can ask: how do I express it? Again, there may be several solutions. We shall see that even in the case of formal languages there are endlessly many ways to express one and the same meaning.

Second, with respect to meanings the question of identity is a pressing one. How shall we picture the meaning of /All dogs are asleep./? In linguistics the answer runs as follows: depending on circumstances it is true or false. The circumstances are encoded into the notion of a world. Thus within Montague semantics the meaning of such a sentence is a function from worlds to truth values.

The model theorists answer is somewhat different. The model theorist does not use grand terminology and instead talks about model. A model is a structure in which every meaningful unit of the language is interpreted by a concrete object. In our particular example, a model may consist of a domain $D$, which comprises all objects that at all exist, and two subsets, $G \subseteq D$ and $A \subseteq D$. The triple $\langle D, G, A \rangle$ is our model. Next, we assign to the expression /dogs/ the interpretation $G$ and to /asleep/ the interpretation $A$. Finally, we choose to interpret the entire sentence as follows. A sentence of the form /All X are Y./ is true if whatever $X$ denotes is a subset of what $Y$ denotes. So, in our example it is true if $G$ is a subset of $A$. This is actually the end of story. The meaning, if you wish, is a function from models to
truth values. On the other hand, the latter answer is somewhat unsatisfactory. For if the domain turns out to be infinite there is no direct way to assess the truth of a sentence; it is impossible to run through all the members of $G$—unless $G$ is finite. It is in these circumstances that we envisage another solution to the problem. It consists in a principled answer if one can at all given. This principled answer can be given through a symbolic calculus. In the present circumstances, suppose by way of example that we know that all mammals are asleep. Suppose also that we know that dogs are mammals. Then /All dogs are asleep./ is true no matter what the model is, and no matter whether it is finite.
Chapter 1

Propositional Calculus

1.1 The Basic Case

Let our alphabet consist of the following letters:

\[(1.1) \quad A := \{\emptyset, 1, \cdots, 9, p, (, ), \land, \lor, \to, \neg\}\]

The set \(D\) is the set of digits (letter 0 to 9) from the above. A variable is a string of the form \(pD^*\). The set of variables is denoted by \(\text{Var}\). A valuation is a function from \(\text{Var}\) to \(\{0, 1\}\).

A well-formed expression is a string \(\vec{x} \in A^*\) such that either of the following holds.

1. \(\vec{x} \in \text{Var}\).
2. \(\vec{x} = (\neg \vec{y})\), where \(\vec{y}\) is a well-formed expression.
3. \(\vec{x} = (\vec{y} \land \vec{z})\), where \(\vec{x}\) and \(\vec{y}\) are well-formed expressions.
4. \(\vec{x} = (\vec{y} \lor \vec{z})\), where \(\vec{x}\) and \(\vec{y}\) are well-formed expressions.
5. \(\vec{x} = (\vec{y} \to \vec{z})\), where \(\vec{x}\) and \(\vec{y}\) are well-formed expressions.

Notice that in this definition the strings \(\vec{y}\) and \(\vec{z}\) are shorter than \(\vec{x}\), and so the definition is grounded. Let us pause here and prove an important property.
Theorem 1.1 (Unique Readability) Let \( \vec{x} \) be a well-formed expression. Then exactly one of 1.-5. is true of \( \vec{x} \). Moreover, if 2. holds, \( \vec{y} \) is unique, and if 3.–5. hold then \( \vec{y} \) and \( \vec{z} \) are unique.

Proof. Let \( \vec{x} \) be a string, and \( i \) a position in \( \vec{x} \) (this is a number from 0 to the length of \( \vec{x} \) minus 1). We define the *embedding depth* of \( i \) (actually, the symbol occurring at \( i \)) as follows.

\[
e(\vec{x})(i) := \sum_{j=0}^{i} b(j)
\]

where \( b(\epsilon) = 1 \) and \( b(\ell) = -1 \), and \( b(\ell) = 0 \) for all other members of \( A \). Then the following holds, as can be shown by induction on the length of \( \vec{x} \).

1. For all \( i < |\vec{x}| \): \( e(\vec{x})(i) \geq 0 \).
2. If \( e(\vec{x})(i) = 0 \) for all \( i < |\vec{x}| \) then \( \vec{x} \) is a variable (and conversely).
3. If \( \vec{x} \) is not a variable then there is exactly one operation symbol (\( \land \), \( \lor \), \( \rightarrow \), \( \neg \)) which has embedding depth 1. This is the main symbol.

Now, let \( \vec{x} \) be given. If it is not a variable it contains an operation symbol. Moreover, by 3. it contains exactly one main symbol occurrence, and this is the one that determines the decomposition. For example, if \( \vec{x} = (\vec{y} \land \vec{z}) \) then the shown occurrence of \( \land \) has depth 1. And it is the only one operation symbol occurrence. So, this decomposition of \( \vec{x} \) is unique. \( \square \)

Fix a valuation \( v : \text{Var} \rightarrow \{0, 1\} \). Then for every well-formed expression, the *truth value* \([\vec{x}]^v \) is defined as follows.

1. If \( \vec{x} \in \text{Var} \) then \([\vec{x}]^v = v(\vec{x}).
2. If \( \vec{x} = (\neg \vec{y}) \), then \([\vec{x}]^v = 1 - [\vec{y}]^v \).
3. \( \vec{x} = (\vec{y} \land \vec{z}) \), then \([\vec{x}]^v = \min([\vec{y}]^v, [\vec{z}]^v) \).
4. \( \vec{x} = (\vec{y} \lor \vec{z}) \), then \([\vec{x}]^v = \max([\vec{y}]^v, [\vec{z}]^v) \).
5. \( \vec{x} = (\vec{y} \rightarrow \vec{z}) \), then \([\vec{x}]^v = \max(1 - [\vec{y}]^v, [\vec{z}]^v) \).
1.1. The Basic Case

Using the Theorem above that establishes unique readability we can now be certain that the definition of $\vec{x}^v$ is unique: $\vec{x}$ has a unique decomposition into $\vec{y}$ (and $\vec{z}$) and if the values of the latter are unique, so is the value of $\vec{x}$. Thus, we are ready for the following definition.

**Definition 1.2** Based on a valuation $v$, the language $P_v$ is defined by $P_v := \{ (\vec{x}, [\vec{x}]^v) : \vec{x} \in \text{Wff} \}$.

We now implement a procedure that reads in a string $\vec{x}$ and raises failure if $\vec{x}$ is not well-formed, and returns $[\vec{x}]^v$, otherwise.

The procedure which is shown below implements a deterministic parser. When the string is completely read, it checks the end result, whether it is true. The parser as implemented is quite general; if we want to use a different valuation we just have to change the definition of $v1$. The procedure is as follows. We keep two stacks: one stack is for the truth values of the well formed expressions recognised so far; the other stack is for the operation symbols that have not been fully integrated. The parser reads in symbol and has to decide when a variable has been completely read. The diagnostic is that it hits an operation symbol or a closing bracket. When a variable is found, its value is put on the value stack. Also, when a bracket closes, an expression is found, and we look at the operator stack to see which operator symbol has latest been found. This is the one we shall use; if it is negation, we pop off 1 value, if it is a binary symbol we pop off 2 values and apply the function. It should be noted that we keep a balance sheet of opening brackets. If you do not do this, then $\text{p&p0}$ is accepted, and the result is true. Can you see why this is so?

I shall now discuss the distinction between a *constant* and a *variable*. The difference comes down to this. A constant, like *false* (or $\bot$ in logic textbooks) has a specific value. You cannot go ahead and give it a different value though you can give something the value *false*, which strictly speaking means that you give the value that *false* has. (In the context of this lecture it is of extreme importance to distinguish the expression from what it denotes. The constant *false* is a string, and it denotes a value. It is for this reason that I enclose strings in slashes, like this: /*false*/.) In OCaml, you cannot assign *false* a value, no matter what it may be. A variable on the other hand can freely be assigned a value. Strictly speaking, variables do not exist in semantics. This idea is misleading. Rather that thinking of a variable as something that can change in value you should think of it as an unknown value. For while the value of some statements, say /*(1=0)/ is known, the
exception Error of string
let vl x = match x with "p" -> true | "p0" -> false
| "p1" -> true

let parse s =
let b = Buffer.create 5
and var = ref true
and bc = ref 0
and vst = Stack.create ()
and ost = Stack.create ()
and strm = Stream.of_string s
in let f c =
  match c with
  | '(' -> (bc := !bc + 1)
  | 'p' -> (Buffer.add_char b 'p'; var := true)
  | '0' | '1' | '2' | '3' | '4' | '5' | '6'
  | '7' | '8' | '9' -> (Buffer.add_char b c)
  | '&' | 'v' | '>' -> (if !bc = 0 then
    raise (Error "Not a well formed expression")
  else bc := !bc - 1;
  if !var then
    Stack.push (vl (Buffer.contents b)) vst;
    Buffer.clear b; Stack.push c ost)
  | '¬' -> (Stack.push c ost)
  | ')' -> begin
    var := false;
    Stack.push (vl (Buffer.contents b)) vst;
    Buffer.clear b;
    match (Stack.pop ost) with
    | '¬' -> (Stack.push (not (Stack.pop vst)) vst)
    | '&' ->
      let p = Stack.pop vst and q = Stack.pop vst
      in (Stack.push (q && p) vst)
    | 'v' -> let p = Stack.pop vst and q = Stack.pop vst
      in (Stack.push (q || p) vst)
    | '>' -> let p = Stack.pop vst and q = Stack.pop vst
      in (Stack.push ((not q) || p) vst)
  end
in
Stream.iter f strm;
if !bc = 0 then
  Stack.top vst
else raise (Error "Not a well formed expression") ;;
1.1. The Basic Case

value of other is not or not necessarily (an example is Caesar conquered Gaul with 513657 soldiers./). This is not to say that they do not have a truth value (as opposed to what quantum mechanics says); it is to say that we do not know the value. In spite of that we can reason with such sentences. We know, for example, that the sentence (1=0) and Caesar conquered Gaul with 513657 soldiers./ is false. How do we know this? We know it because we can effectively decide what truth values proposition can have given a valuation. The first property is easy to see.

Definition 1.3  Let $\overline{u}$ be a variable and $\overline{x}$ be a proposition. $\overline{u}$ occurs in $\overline{x}$ if $\overline{x} = \overline{y} \overline{u} \overline{z}$ for some $\overline{y}$ and $\overline{z}$ such that $\overline{z}$ does not start with a digit.

For example, $p$ occurs in $(p \land p \land 0)$ but not in $(p \land 0 \land p 1)$ even though there is a decomposition $(p \land 0 \land p 1) = (p \land \neg \overline{0} \land p 1)$. For in this decomposition, the third string starts with $0$.

Proposition 1.4 (Coincidence Lemma) Let $\overline{x}$ be a proposition. Let $v$ and $w$ be two valuations such that for every variable $\overline{u}$ occurring in $\overline{x}$, $[\overline{u}]^v = [\overline{u}]^w$. Then $[\overline{x}]^v = [\overline{x}]^w$.

Here is a proof sketch. Suppose $\overline{x}$ is a variable; then $\overline{u}$ can occur in $\overline{x}$ only if it is identical to $\overline{x}$. The claim then follows by definition. Or $\overline{x}$ is not a variable and then either $\overline{x} = (\neg \overline{y})$ or $\overline{x} = (\overline{y} \land \overline{z})$, $\overline{x} = (\overline{y} \lor \overline{z})$, or $\overline{x} = (\overline{y} \to \overline{z})$. In the first case, $\overline{u}$ occurs in $\overline{x}$ if and only if it also occurs in $\overline{y}$ and the claim follows by inductive hypothesis. In the remaining cases, $\overline{u}$ occurs in $\overline{x}$ if and only if it either occurs in $\overline{y}$ or in $\overline{z}$. Again, we can use the inductive hypothesis to establish the claim.

So let us be given a proposition $\overline{x}$. We can then give a procedure to establish, given a valuation $v$, whether or not $\overline{x}$ is true under $v$. Of course, one procedure is simply to evaluate $\overline{x}$. But we can be more specific; we can, on the basis of $\overline{x}$, say which variables must be made true together by $v$, and which ones must be made false. Before we begin, let us introduce the notion of a partial valuation. A partial valuation is a valuation that is defined only on part of the variables. It is a partial function from $\text{Var}$ to $\{0, 1\}$. We define truth under a partial valuation in the same way as truth under a valuation except that for a variable $\overline{x}$, $[\overline{x}]^v$ is undefined if $\overline{x}$ has no value under $v$. It then follows that $[\overline{x}]^v$ is undefined if $v$ is undefined under some of the variables occurring in $\overline{x}$. So, $\overline{x}$ receives a value under a partial
valuation if (and only if) v is defined on all variables occurring in \( \vec{x} \). Denote this set by var(\( \vec{x} \)). We can use the Coincidence Lemma to establish that \( \vec{x} \) is true under v if and only if it is true under the restriction \( \vec{x} \upharpoonright \text{var}(x) \). And so we only need to establish all possible choices of partial valuations on var(\( \vec{x} \)). There are exactly \( 2^{\text{var}(\vec{x})} \) many such valuations, and it is possible to get them just by simply checking them all out. There is a more efficient method, known as **tableau method**. A **signed formula** is a pair of the form \( V \vec{x} \), where \( V \) is truth value (we write T or F). A **line** is a sequence of sets of signed formulae, separated by \( | \). The conditions in a set are to be satisfied simultaneously, while each of the different sets is an alternative to the other. The tableau calculus is a set of procedures to go from line to the next. When no more rules apply, the tableau is finished. If it is empty, the tableau closes, otherwise it is said to be open.

The tableau starts with T\( \vec{x} \). The rules are:

\[
\begin{align*}
A &\mid \Delta; T\vec{x} \mid F\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid F\vec{x} \mid B \\
A &\mid \Delta; F\vec{x} \mid B \\
A &\mid \Delta; F\vec{x} \mid B \\
A &\mid \Delta; F\vec{x} \mid B \\
A &\mid \Delta; F\vec{x} \mid B \\
A &\mid \Delta; F\vec{x} \mid B \\
A &\mid \Delta; F\vec{x} \mid B
\end{align*}
\]

\begin{align*}
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B \\
A &\mid \Delta; T\vec{x} \mid B
\end{align*}

A few facts are easy to see. (1) If no rules can apply then all set consist of signed formulae of the form T\( \vec{x} \) or F\( \vec{x} \) with \( \vec{x} \) a variable. (2) Every step reduces the length of the formula occurring in the set. Hence it is not possible to run the procedure indefinitely. Also, every condition above the line is equivalent to the condition below the line. Thus, if the tableau closes the original formula cannot be made true, but if it is open, then sets indicate the conditions on the valuations that make the formula true.

The most efficient way to use a tableau is to always prefer a conjunctive step (one that does not increase alternatives) over a disjunctive step. This keeps the tableau small and reduces computation.

I end this section with a few remarks. I shall use slashes to enclose a material string. Thus, whenever I speak of concrete strings, I shall enclose them in slashes.
1.2 Abstract Syntax

Thus, /false/ is a string consisting of five letters in sequence. It is customary to regard some parts of the string encoding as artifacts of their physical realisation. Such artifacts are, for example, different kinds of white space. White space includes blanks, tabs, and newlines. In a string as encoded in a computer, we consider blanks, tabs and newlines as equal. However, two newlines equal a new paragraph, but two blanks are the same as one. Often we do not however consider blanks as important. (Roman inscriptions, for example, do not use blanks for any purpose.) There is a style of writing (used with typewriters) where every letter is followed by a blank. (This is called spaced letters.) In this style, three blanks equal an ordinary blank, and five blanks equal three blanks. I will occasionally comment on white space when necessary.

1.2 Abstract Syntax

It is quite common in everyday usage to deviate from the official standard to some degree. We are used, for example, to dropping brackets from certain expressions. Outermost brackets are dropped, and so are brackets that subdivide a purely conjunctive or disjunctive proposition, as in /p0 ∧ p ∧ p1/ or /p0 ∨ p ∨ p2/. Yet, these simplifications are not part of the original definition. Typically, the answer to this is that such strings are just “shorthand” notations. The problem is: shorthand for what? Consider the string /p0 ∧ p ∧ p1/. If this is a shorthand, what is the original? Is it /((p0 ∧ p) ∧ p1)/ or is it /(p0 ∧ (p ∧ p1))/? The reply to this is, of course, that it does not matter. The truth value is the same anyway. So it seems that we depart from the original notation in a way that lets us recoved the original up to inessential variation; a variation is inessential if the meaning is the same. But just how much are we allowed to fiddle with the syntax?

Our version of that story is slightly different. We shall say that the two versions of propositional logic, the one introduced in the previous section, and the one described above, are different languages but have the same abstract syntax. It is this notion that I shall define in this section. First however some basic definitions. We shall say that language is defined to be a set of signs. A sign is a pair σ = ⟨e, m⟩, where e is an expression, and m a meaning. To define a language we first set out to define the set E of expressions, and the set M of meanings, and then say what signs may have what meaning. In many cases we simply put E = A* for some finite set A, called alphabet. While a language is a set, a grammar is a
description of that set; and we take this description here as a finite set of functions that produce each member of the language in finite time, and that produce nothing but members of that language. It is highly tempting to think that languages simply ‘must’ have a certain grammar (for example, that the language of propositional calculus as defined earlier, must have the grammar shown below). But this is mostly an illusion. There typically are endless different grammars one can propose for a given language; however, most of them will be rather pointless. But rather than looking at the question of what grammar to choose for a language, we shall in some more detail at variants of languages that share the same semantics component alias abstract syntax. It is the study of these variations that shows great promise in enhancing our understanding of language.

Before I deal with the general concept, let me explain it with the present case at hand. A language is a set of pairs; a grammar is a set of rules that generates them. Here is how.

Suppose we have two truth values, 0 and 1, and the following functions.

\[
\begin{array}{c|c|c|c|c|c|c|c}
\neg & \cap & \cup & \supset & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

It is then possible to rephrase the previous definitions for truth values as follows.

\[
\begin{align*}
\neg \bar{x} & = [\bar{x}]' \\
\land \bar{x} \land \bar{y} & = [\bar{x}]' \land [\bar{y}]' \\
\lor \bar{x} \lor \bar{y} & = [\bar{x}]' \lor [\bar{y}]' \\
\rightarrow \bar{x} \rightarrow \bar{y} & = [\bar{x}]' \rightarrow [\bar{y}]' \\
\end{align*}
\]

This sounds like the same thing, except that we shall make the above functions part of our semantics. The relevant notion is that of an algebra. Thus, rather than dealing with set of values we are now dealing with a set together with certain functions over that set.

**Definition 1.5** A **signature** is a pair \( \langle F, \Omega \rangle \) (denoted simply by \( \Omega \)) where \( F \) is a set (the set of function symbols) and \( \Omega : F \to \mathbb{N} \). An **\( \Omega \)-algebra** is a pair \( \langle A, I \rangle \) where \( A \) is a set and \( I \) a function with domain \( F \) such that \( I(f) : A^{\Omega(f)} \to A \).
1.2. Abstract Syntax

So, $I(f)$ is a function on $A$ with $\Omega(f)$ arguments. In our example, let $F := \{f_\land, f_\lor, f\to\}$; then $\Omega(f_\land) = 1$, $\Omega(f_\lor) = \Omega(f\to) = 2$. Furthermore, put $I(f_\land) := \land$, $I(f_\lor) := \lor$, $I(f\to) := \Rightarrow$. Then the pair $2 := \langle\{0, 1\}, I\rangle$ is an $\Omega$-algebra.

This formulation allows us to also consider the set of string as an algebra. Let $C$ be a function that assigns to $f_\neg$ a unary function on strings, and to $f_\land, f_\lor$ and $f\to$ a binary function in the following way:

1. $C(f_\neg)(\bar{x}) := (\neg \bar{x})$
2. $C(f_\land)(\bar{x}, \bar{y}) := (\bar{x} \land \bar{y})$
3. $C(f_\lor)(\bar{x}, \bar{y}) := (\bar{x} \lor \bar{y})$
4. $C(f\to)(\bar{x}, \bar{y}) := (\bar{x} \to \bar{y})$

The strings also form an algebra, $\langle A^*, C\rangle$. Notice that this set does not contain a lot of non well-formed expressions, too. On these, the functions are also defined. For example, $C(f_\neg)(\emptyset p) = (\neg \emptyset p))$. The well-formed expressions are in fact the strings that can be generated from the variables.

**Proposition 1.6** A string over $A$ is a well-formed expression if and only if it is generated from the variables by means of the functions $C(f_\neg), C(f_\land), C(f_\lor)$ and $C(f\to)$.

So, we now have two algebras, one for the semantics and one for the strings (syntax). There is a third one, which plays an important theoretical role, and it is the algebra of terms, or the free algebra.

**Definition 1.7 (Terms I)** Let $V$ be a set. The set of $\Omega$-terms over $V$ is the least set $S$ such that $V \subseteq S$ and such that whenever $f \in F$ and we have $\Omega(f)$ many terms $t_0, \cdots, t_{\Omega(f)-1}$, then also $f(t_0, \cdots, t_{\Omega(f)-1})$ is a term.

In this formulation the definition makes terms abstract. I prefer the following definition.
1. Propositional Calculus

**Definition 1.8 (Terms II)** Let \( V \) be a set. The set of \( \Omega \)-terms over \( V \) is the least set \( S \subseteq (F \cup V)^+ \) such that \( V \subseteq S \) and such that whenever \( f \in F \) and we have \( \Omega(f) \) many terms \( t_0, \cdots, t_{\Omega(f)-1} \), then also \( f \tilde{t}_0 \cdots \tilde{t}_{\Omega(f)-1} \in S \). This set is denoted by \( \text{Tm}_\Omega(V) \).

Thus, the terms are strings over the alphabet of function symbols plus the set of variables. Although a proof will be given later, I announce the following result. Whenever we have a term \( \vec{u} \), it is either in \( V \), or it has the form \( f \vec{v}_0 \cdots \vec{v}_{\Omega(f)-1} \), and \( f \) as well as the \( \vec{v}_i, i < \Omega(f) \), are unique. This is necessary to ensure that the string faithfully represents the term. Once we know, of course, we definite the strings to be the terms.

Again, the terms form an algebra, as follows. The set is \( \text{Tm}_\Omega(V) \), and the operations are

\[
T(f) : \langle t_0, \cdots, t_{\Omega(f)-1} \rangle \mapsto f \tilde{t}_0 \cdots \tilde{t}_{\Omega(f)-1}
\]

In our scenario of propositional calculus, \( V = \text{Var} \), and the algebra of terms has the following operations:

\[
\begin{align*}
\circ & T(f_\neg) : \vec{x} \mapsto (\neg \vec{x}) \\
\circ & T(f_\land) : \langle \vec{x}, \vec{y} \rangle \mapsto (\vec{x} \land \vec{y}) \\
\circ & T(f_\lor) : \langle \vec{x}, \vec{y} \rangle \mapsto (\vec{x} \lor \vec{y}) \\
\circ & T(f_\rightarrow) : \langle \vec{x}, \vec{y} \rangle \mapsto (\vec{x} \rightarrow \vec{y})
\end{align*}
\]

**Definition 1.9 (Term Algebra)** The pair \( \Sigma\text{m}_\Omega(V) := \langle \text{Tm}_\Omega(V), T \rangle \) is called the term algebra generated by \( V \) over the signature \( \Omega \).

Given two \( \Omega \)-algebras \( \mathfrak{A} = \langle A, I \rangle \) and \( \mathfrak{B} = \langle B, J \rangle \), we call a map \( h : A \to B \) a homomorphism from \( \mathfrak{A} \) to \( \mathfrak{B} \) if the following holds: for all \( f \) and \( a_0, \cdots, a_{\Omega(f)-1} \in A \):

\[
h(I(f)(a_0, \cdots, a_{\Omega(f)-1})) = J(f)(h(a_0), \cdots, h(a_{\Omega(f)-1}))
\]
In our present case, this definition boils down to four requirements:

\[
\begin{align*}
    h(I(f_\lambda)(a_0)) &= J(f_\lambda)(h(a_0)) \\
    h(I(f_\land)(a_0, a_1)) &= J(f_\land)(h(a_0), h(a_1)) \\
    h(I(f_\lor)(a_0, a_1)) &= J(f_\lor)(h(a_0), h(a_1)) \\
    h(I(f_{\neg})(a_0, a_1)) &= J(f_{\neg})(h(a_0), h(a_1))
\end{align*}
\] (1.7)

Here is an example. Let \( A \) be the set of subsets of some set \( X \), and let 
\[ I(f_\land)(a_0) := X \setminus a_0, \quad I(f_\lor)(a_0, a_1) := a_0 \cap a_1, \quad I(f_{\neg})(a_0, a_1) := a_0 \cup a_1, \quad I(f_{\neg})(a_0) := (X \setminus a_0) \cup a_1. \] 
Then \((A, I)\) is an \( \Omega \)-algebra for our signature. Such algebras are called **boolean algebras**. Now consider the following map: Let \( x \in X \) be a specific element; and 
\[ h(a) := 1 \text{ if } x \in X, \quad h(a) := 0 \text{ otherwise.} \] This is a homomorphism to the algebra \( 2 \) defined above.

**Definition 1.10 (Grammar)** A grammar is a quintuple \( \langle \Omega, V, \mathcal{E}, \mathcal{M}, h \rangle \), where \( \Omega \) is a signature, \( \mathcal{E} \) an algebra of expressions, \( \mathcal{M} \) an algebra of meanings, and \( h \) a homomorphism from \( \text{Term}_\Omega V = \mathcal{E} \times \mathcal{M} \).

Actually, we shall not do things this way; rather, we shall just specify the set \( E \) (which will typically be \( A^* \) for some alphabet \( A \)), a set \( M \), and two maps \( h \) and \( k \) from the terms into the set \( E \) and \( M \) respectively. The functions that make this into a homomorphism are then simply defined to be the images of the maps \( f \) under \( h \) and \( k \). (We have done it exactly like that above.) However, the reason for providing the above definition is that it is not enough to just postulate the maps \( h \) and \( k \). For it may turn out that there are no functions that can be given inside \( E \) and \( M \) that turn them into algebras.

I give some examples to show the difference between maps and homomorphisms. Suppose we define negation on the set \( \{0', u, 1'\} \) by \( -0' = 1', -u = u \) and \( -1' = 0' \). So we look at the algebra \( \langle \{0', u, 1'\}, K \rangle \) where \( K(f_{\neg}) = \neg \). Then there is no map \( h : \{0', 1', u\} \to \{0, 1\} \) that would turn this into a homomorphism into \( 2 \). To see this, let \( h(0') = 0 \). Then we must set \( h(1') = 1 \); and if we put \( h(0') = 1 \) then \( h(1') = 0 \). Now, \( h(u) = h(-u) \), so it is not hard to see that we cannot assign any value to \( u \). For we must have \( h(-u) = -h(u) \), which does not hold of either 0 or 1.

Now suppose we are given a signature, and some alphabet. Say, the signature consists of a single symbol \( s \) with \( \Omega(s) = 1 \). Let \( V := \{0\} \). The terms are of the form \( 0, s(0), s(s(0)) \), and so on, which can be equated with the natural numbers.
Namely, the term of the form $s^n(0)$ represents the number $n$. I choose the following map: $h(n) := 1$ if $n$ is divisible by 3, and $h(n) = 0$ otherwise. There is no unary map $m$ on $\{0, 1\}$ that can make this a homomorphism. For look at the equation

\[(1.8) \quad h(s(n)) = m(h(n))\]

For choose $n = 3k$. Then $h(s(n)) = h(3k + 1) = 0$, and so $m(h(n)) = m(h(3k)) = m(1) = 0$. On the other hand, if $n = 3k + 1$ then $h(s(n)) = 0$ and $m(h(3k + 1)) = m(0) = 0$ and if $n = 3k+2$ then $h(s(n)) = h((3+1)k) = 1$ and $m(h(3k+2)) = m(0) = 1$. Thus we must satisfy both $m(0) = 0$ and $m(1) = 1$m which is impossible.

A final detail needs to be settled. Given two algebras $\mathfrak{A} = \langle A, I \rangle$ and $\mathfrak{B} = \langle B, J \rangle$, the product is defined by $\langle A \times B, I \times J \rangle$, where

\[(1.9) \quad (I \times J)((a_0, b_0), \cdots, (a_{\Omega(f)}-1, b_{\Omega(f)}-1)) = (I(f)(a_0, \cdots, a_{\Omega(f)}-1), J(f)(b_0, \cdots, b_{\Omega(f)}-1))\]

This defines the symbol ‘×’ in the above definition. Moreover, the maps $\pi_0 : (a, b) \mapsto a$ and $\pi_1 : (a, b) \mapsto b$ are the first and second projection. (The projections are, by the way, homomorphisms.)

**Definition 1.11 (Analysis Term)** Let $G = \langle \Omega, V, \mathfrak{M}, h \rangle$ be a grammar. Then the unfolding of a term $t$ is defined to be $h(t)$. Given a sign $\sigma = \langle e, m \rangle$, we say that $t$ is an analysis term of $\sigma$ if $h(t) = \sigma$. $L(G)$ is the set of all signs that have an analysis term.

**Definition 1.12 (Ambiguity)** Let $G$ be a grammar, and $e$ a string. An analysis term of $e$ is any term $t$ that unfolds to a sign $\langle e, m \rangle$ for some $m$. A pair of terms $(t, t')$ is called an ambiguity if $h(t) = \langle e, m \rangle$, $h(t') = \langle e, m' \rangle$ for some $e$, $m$, $m'$. The ambiguity is spurious if $m = m'$.

The notion of a spurious ambiguity will come up later. It basically is the case when a string has two analysis terms that both result in the same meaning.

Finally we turn to the notion of abstract syntax. What is it that different notations have in common when we say that they share the same abstract syntax? Our answer is that what they have in common is the semantic operations.
1.3 Alternative Surface Syntax

**Definition 1.13 (Abstract Syntax I)** Let $G = \langle \Omega, V, \mathcal{M}, h \rangle$ be a grammar. The abstract syntax of $G$ is $\langle \Omega, V', \mathcal{M}, h' \rangle$, where $V' := \{ \pi_1(v) : v \in V \}$ and $h' : t \mapsto \pi_1(h(t))$.

If you look at this definition you see that what it does is cut the grammar ‘in half’. It simply takes only the semantics side; the variables are interpreted only by their meanings irrespective of their form. And the functions are interpreted by their actions on $M$ alone. In this abstract form, however, it is not possible to appreciate this definition. Therefore, in the next section we actually look at some examples.

For certain reasons this definition is not adequate. For it allows situations which are hardly of purely semantic nature. Let us assume, for example, that we have two function symbols $f$ and $g$ that are interpreted as the same semantic function (though their syntax may differ). Or consider an interpretation where the interpretations of $f$ and $g$ are exchanged. We do not wish to say that something essential has changed as far as the expressive side of the language is concerned. Thus, in actual fact, the added signature is of no significance and should be axed.

**Definition 1.14 (Abstract Syntax II)** Let $G = \langle \Omega, V, \mathcal{M}, h \rangle$ be a grammar, with $\mathcal{M} = \langle M, J \rangle$. The abstract syntax of $G$ is the triple $\langle M, V', F \rangle$, where $V' := \{ \pi_1(v) : v \in V \}$ and $F = \{ J(f) : f \in F \}$.

Notice that the pair $\langle M, F \rangle$ is not an algebra in the technical sense. For it does not state by which concrete semantic function we wish to interpret the symbols.

**1.3 Alternative Surface Syntax**

Let us look at a few variants of boolean logic.

**Example 1.** **Alphabetic Variants.** The simplest change is the change of the alphabetic symbols. For example, we might choose to write ‘&’ in place of ‘∧’, or ‘x’ in place of ‘p’. This changes the look and feel of the strings but not the constitution of the formulae. So, under these replacements the formula $/(p\bigland p)/$ becomes $/(x\bigland x)/$, for example.
Example 2. Dot Notation. Define the following string homomorphism: let $B := D \cup \{., p, \neg, \land, \lor, \to\}$. For symbols of $A$, $s(\epsilon) := ., s(.) := ., \text{and } s(x) := x$ for other symbols. If this is a string homomorphism, this means that it is applied to all letters in turn. So, we have

$$s((p \land p \to (\neg p))) = p \land p . \to \neg \ldots$$

I can give the following grammar for it. The formation rules are as follows (with the final period being part of the string written).

1. $D(f\neg)(\vec{x}) := \neg \vec{x}$.
2. $D(f\land)(\vec{x}, \vec{y}) := \vec{x} \land \vec{y}$.
3. $D(f\lor)(\vec{x}, \vec{y}) := \vec{x} \lor \vec{y}$.
4. $D(f\to)(\vec{x}, \vec{y}) := \vec{x} \to \vec{y}$.

This gives you

$$s(C(f\neg)(\vec{x})) = D(f\neg)(\vec{x})$$

and similarly for the other functions. It can be checked that this defines a grammar in the technical sense. Moreover, this grammar has the same semantical component as our original grammar. So, they share the abstract syntax.

In one of the assignments I set the task to show that this notation is unambiguous.

Example 3. Polish Notation. The alphabet is $\{p, \neg, \land, \lor, \to\} \cup D$.

1. $P(f\neg)(\vec{x}) := \neg \vec{x}.$
2. $P(f\land)(\vec{x}, \vec{y}) := \vec{x} \land \vec{y}.$
3. $P(f\lor)(\vec{x}, \vec{y}) := \vec{x} \lor \vec{y}.$
4. $P(f\to)(\vec{x}, \vec{y}) := \vec{x} \to \vec{y}.$
1.3. Alternative Surface Syntax

We have used this notation to write terms. The following is an important tool in recognising strings in Polish Notation. Let the weight of a symbol be defined as follows. (1) A variable has weight \(-1\) (thus, ‘p’ has weight \(-1\), digits have weight 0), (2) an n-ary function has weight \(n - 1\). A string \(\vec{x}\) is a well formed expression if and only if the sum of the weights is \(-1\), and for no proper prefix the sum is \(< 0\) (except if the prefix omits some digits). Thus, the string \(/\land\neg p0\lor p2p13/\). The cumulative weight is written below each symbol.

\[
\begin{array}{cccccc}
\land & \neg & p & 0 & \lor & p & 2 & p & 1 & 3 \\
1 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & -1 & -1 & -1 \\
\end{array}
\]

(1.12)

This is a term since the first time we hit \(-1\) is when only digits remain.

It follows that Polish Notation is uniquely readable. Instead of formulating this in terms of string decompositions, we shall express this simply by saying that Polish Notation is unambiguous. This means that every well formed expression has a unique analysis term. This justifies in retrospect the choice of strings in Polish Notation to represent terms.

Example 4. Dropping Brackets. It is customary to drop brackets in certain circumstances. Outer brackets are often dropped. Also, conjunctive terms or disjunctive terms are often written without the use of brackets. Thus, we write \(/p0\land p21\land p3\land p/\), doing entirely without brackets. However, in \(/p0\land (p\lor p2)/\) the brackets remain. The definition of the grammar is somewhat intricate. For example, we can no longer write a simple definition for the interpretation of \(f_\land\). For it sometimes is realised as the map \(\langle \vec{x}, \vec{y} \rangle \mapsto (\vec{x}\land \vec{y})\) and sometimes as the map \(\langle \vec{x}, \vec{y} \rangle \mapsto \vec{x}\land \vec{y}\). And the exact characterisation of when we can afford to use the latter is not easy. There are two different notations to be looked. The first—and easiest—is when brackets must be dropped. In that case we write the following definitions. Here, a conjunctive term is one which is of the form \(\vec{x}\land \vec{y}\) with \(\vec{x}\) and \(\vec{y}\)
1. Propositional Calculus

terms.

\[
C(f_A)(\vec{x}, \vec{y}) := \vec{x} \land \vec{y}
\]

\[
C(f_\neg)(\vec{x}, \vec{y}) := \begin{cases} 
(\neg \vec{x}) & \text{if } \vec{x} \text{ is not conjunctive} \\
(\neg (\vec{x})) & \text{if } \vec{x} \text{ is conjunctive}
\end{cases}
\]

(1.13)

\[
C(f_\lor)(\vec{x}, \vec{y}) := \begin{cases} 
(\vec{x} \lor \vec{y}) & \text{if neither term is conjunctive} \\
((\vec{x}) \lor \vec{y}) & \text{if only } \vec{x} \text{ is conjunctive} \\
(\vec{x} \lor (\vec{y})) & \text{if only } \vec{y} \text{ is conjunctive} \\
((\vec{x}) \lor (\vec{y})) & \text{if both terms are conjunctive}
\end{cases}
\]

The definition of \( \rightarrow \) proceeds analogously. Now take the term \( /p0\land p21\land p3\land p/ \). It has various analysis terms:

(1.14)

\[
\begin{align*}
f_{\land f_{\land f_{\land p0p21p3p}} f_{\land f_{p0f_{p21p3p}}} f_{\land f_{p0p21f_{p3p}}} f_{p0f_{p21f_{p3p}}} \\
f_{p0f_{p21p3p}} f_{p0f_{p21f_{3p}}} f_{p0f_{p21f_{p3p}}}
\end{align*}
\]

It turns out, though, that the ambiguity is totally spurious: the terms all unfold to the same sign.

Let us return to the more realistic scenario where dropping brackets is optional. Then we cannot use our original signature any more. The reason is that the formation of strings is no longer unique. The strings \( p \land p0 \) and \( (p \land p0) \) have the same analysis term—which cannot be the case. The solution to this is to introduce another binary function symbol \( b_\land \) and define

\[
C(f_A)(\vec{x}, \vec{y}) := \vec{x} \land \vec{y}
\]

\[
C(b_A)(\vec{x}, \vec{y}) := (\vec{x} \land \vec{y})
\]

(1.15)

The definition of the other functions is the same as before.

The previous example show that the view of surface syntax being a shorthand for something else actually is problematic. Programming languages for example do allow to drop brackets in conjunctive terms. However, what the original (deep) notation is for which this one is proxy, is unclear. For an equally simple analysis would be to say that a term of the form \( \vec{x} \land \vec{y} \land \vec{z} \) has only one analysis: the left branching one. (In principle, you can pick also the right branching analysis or something more exotic, it does not matter.) This requires careful execution on the part of a grammar writer, but it may be clear that to claim a left branching analysis as the only option is not so far off the mark; computers really scan strings left to right and commit early to a constituent when they see one. (OCaml definitely does that.)
Example 5. **Obligatory Differentiation.** I shall present a syntax that is highly unusual, but interesting in its own right. Consider the formula \( p \land p \). It is (semantically) equivalent to \( p \). Why therefore allow the luxury to write the former? Just as we previously dropped the brackets to eliminate spurious ambiguity, we may now propose a grammar that will prevent us from writing a conjunction of identical formulae. Essentially, this can be done in two ways. One is to make the interpretation of \( f_\lor \) partial. It will simply refuse to be applied to the pair \( \langle \vec{x}, \vec{y} \rangle \) (or some other similarly redundant pairs). Another solution is to invent an intelligent string mechanism that drops from \( \vec{y} \), say, any conjuncts that already appear in \( \vec{x} \). This strategy occurs actually in real life. Consider using lists to represent sets. You may want lists simply to behave like sets. So you must define operations that avoid adding the same element twice (and, if you read the OCaml manual on the Set module, you also need to define an ordering of the elements so that the elements will always appear naturally ordered). The approach via partial functions has the advantage that it keeps the ambiguity to a minimum (in the nonpartial version \( p \) has infinitely many analysis terms). Moreover, it allows to use straightforward concatenation as string functions, while the nonpartial solution will have to decompose strings.

I shall now give some examples from natural languages. In English, we have the expression \( \text{and} \) which is used in infix notation. That is to say, it is put in between its arguments. For implication we have the pair \( \text{if} \cdots \text{then} \). The syntactic interpretation of \( f_\lor \) in English is therefore (among other things) the function

\[
(1.16) \quad \langle \vec{x}, \vec{y} \rangle \mapsto \text{if} \ \vec{x} \ \text{then} \ \vec{y}
\]

In morphology this is known as a **transfix**. More examples are \( \text{both} \cdots \text{and} \), \( \text{either} \cdots \text{or} \) and \( \text{neither} \cdots \text{nor} \). In Latin, there is in addition to the word \( \text{et} \) also the suffix \( \text{que} \), which is added after the second conjunct (so is technically in postfix notation, only that morphologically it is an affix rather than a word). For example, the letters SPQR stand for \( \text{senatus populusque Romanus} \) “senate and the people of Rome”. Note that \( \text{que} \) is put after the head noun. Another famous expression is \( \text{aut Caesar aut nihil} \) “Either Caesar or nothing”, where \( \text{aut} \) appears before each disjunct. All these variations are rather changes in surface syntax, they concern the outward appearance of expressions, not their meaning.
1. Propositional Calculus

1.4 Ambiguity

In the previous section I have shown some variant of propositional logic which were ambiguous, though the ambiguity was spurious, that is semantically irrelevant. In this section I shall look at variants that come closer to natural language.

Example 6. BRACKET FREE INFIX NOTATION. Consider the following string homomorphism. \( r(\cdot) := r(\cdot) := \varepsilon; r(x) := x \) otherwise. This homomorphism erases all brackets from strings. Thus, applied to the string \( /p0 \land (p \lor \neg p2)/ \) it gives \( /p0 \land p \lor \neg p2/. \) This means we define the following grammar. The meaning algebra is the same (so, same abstract syntax!), but the string algebra is now this. \( \langle A^*, E \rangle \) with

\[
\begin{align*}
\circ \ E(f_\neg)(\vec{x}) & := \neg \vec{x} \\
\circ \ E(f_\land)(\vec{x}, \vec{y}) & := \vec{x} \land \vec{y} \\
\circ \ E(f_\lor)(\vec{x}, \vec{y}) & := \vec{x} \lor \vec{y} \\
\circ \ E(f_\rightarrow)(\vec{x}, \vec{y}) & := \vec{x} \rightarrow \vec{y}
\end{align*}
\]

In the example above there are plenty of ambiguities, many of them not spurious. The example above is of that kind. Suppose that \( /p0/ \) is false, \( /p/ \) is true, and \( /p2/ \) is false. Then \( /p \lor \neg p2/ \) is true, and so \( /p0 \land p \lor \neg p2/ \) must be false. On the other hand, \( /p0 \land p/ \) is false and \( /\neg p2/ \) must be true, making the entire string true. A formal analysis of this example runs as follows. Let us define the following primitive signs.

\[
(1.17) \quad \sigma := \langle p, 1 \rangle, \sigma_0 := \langle p0, 0 \rangle, \sigma_2 := \langle p2, 0 \rangle
\]

Let \( V := \{ \sigma, \sigma_0, \sigma_2 \} \). The given string has the following analysis terms: \( t = f_\land \sigma_0 f_\lor \sigma f_\rightarrow \sigma_2 \), and \( t' = f_\lor f_\land \sigma_0 \sigma f_\rightarrow \sigma_2 \). The map \( h \) is defined in the obvious way.
We now have
\[
\begin{align*}
h(f_\alpha f_\nu f_\sigma f_{-\sigma_2}) \\
= (E \times I)(f_\alpha)(\langle p\emptyset, 0 \rangle, h(f_\nu f_\sigma f_{-\sigma_2})) \\
= (E \times I)(f_\alpha)(\langle p\emptyset, 0 \rangle, (E \times I)(f_\nu)(\langle p, 1 \rangle, h(f_\sigma f_{-\sigma_2}))) \\
= (E \times I)(f_\alpha)(\langle p\emptyset, 0 \rangle, (E \times I)(f_\nu)(\langle p, 1 \rangle, (E \times I)(f_\sigma)(\langle p_2, 0 \rangle))).
\end{align*}
\]

(1.18)
\[
\begin{align*}
= (E \times I)(f_\alpha)(\langle p\emptyset, 0 \rangle, (E \times I)(f_\nu)(\langle p, 1 \rangle, (E \times I)(f_\sigma)(\langle p_2, 0 \rangle))) \\
= (E \times I)(f_\alpha)(\langle p\emptyset, 0 \rangle, (p \lor p_2, 1 \lor 1)) \\
= \langle p\emptyset \land p \lor p_2, 0 \rangle
\end{align*}
\]

For the other term we get
\[
\begin{align*}
h(f_\nu f_\alpha f_{\sigma 0} f_{\sigma 0} f_{-\sigma 2}) \\
= (E \times I)(f_\nu)(h(f_\alpha f_{\sigma 0}), h(f_{-\sigma 2})) \\
= (E \times I)(f_\nu)((E \times C)(f_\alpha)(\langle p\emptyset, 0 \rangle, \langle p, 1 \rangle), (E \times C)(f_{\sigma 0})(\langle p_2, 0 \rangle))) \\
= (E \times I)(f_\nu)(\langle p\emptyset \land p, 0 \rangle, (E \times C)(f_{-\sigma})(\langle p_2, 0 \rangle))) \\
= (E \times I)(f_\nu)(\langle p\emptyset \land p, 0 \rangle, (\neg p_2, 1))) \\
= \langle p\emptyset \land p \lor \neg p_2, 1 \rangle
\end{align*}
\]

(1.19)

Example 7. **English Style Syntax.**

- ② $N(f_\gamma)(\bar{x}) := /it \ is \ not \ the \ case \ that \ /\bar{x}$
- ③ $N(f_\lambda)(\bar{x}, \bar{y}) := \bar{x}^\ast /and /\bar{y}$
- ④ $N(f_\nu)(\bar{x}, \bar{y}) := \bar{x}^\ast /or /\bar{y}$
- ⑤ $N(f_\ast)(\bar{x}, \bar{y}) := /if /\neg \bar{x}^\ast /then /\neg \bar{y}$

Suppose we start with the basic sentences $a = /john \ is \ tall/$, $b = /claver \ is \ smart/$, and $c = /elaine \ is \ happy/$, then the term $f_{-\lambda}f_{\lambda}af_\nu b$ will unfold to a sign with expression $/it \ is \ not \ the \ case \ that \ john \ is \ tall \ and \ it \ is \ not \ the \ case \ that \ claver \ is \ smart/$. It is somewhat stilted but nevertheless well-formed.

We are now in a position to clarify some of the terminology that is used in linguistics. We say that a **reading** of a given sentence is an analysis term for that
sentence. A sentence is **ambiguous** if it has two readings. There are two types of ambiguities. One is **lexical**. An ambiguity is lexical if the terms are different only with respect to the variables that are present in the term; and it is **structural** if it is not lexical. Furthermore, we have spoken above about ambiguities that are spurious and those that are not. A spurious ambiguity is a pair of different terms that unfold to the same sign. These distinctions are independent.

**Example 8.** **Lexical Ambiguity.** Consider a word that has two meanings, like `/crane/`. It denotes a type of bird, and a machine to lift heavy weights. For various reasons we think of these as two signs: \( \sigma_b := \langle /crane/, b \rangle \) and \( \sigma_m := \langle /crane/, m \rangle \) where \( b \) and \( m \) are the two meanings that the word has. The sentence `/I like cranes./` is ambiguous since what enters the analysis term may be either \( \sigma_b \) or \( \sigma_m \).

Lexical ambiguity that is spurious falls into two kinds. One is the uninteresting sort: it derives from the fact that one and the same sign is called by two different names in the abstract syntax. This is of course of badly designed grammar. However, the sentence `/Cranes are cranes./` has four different readings all of which are different only in the lexical signs used in the terms. The four readings produce only two distinct signs. We paraphrase them as follows. "Bird cranes are bird cranes." (true), "Bird cranes are machines cranes." (false), "Machine cranes are bird cranes." (false) and "Machine cranes are machine cranes." So, if meanings of sentences are truth values, we are presented with lexical ambiguities some of which are spurious some of which are not.

Ambiguity presents us with a real computational problem. The grammar above is context free; so its parsing complexity is polynomial. Yet a single sentence can have exponentially many readings. Unless the number of nonspurious ambiguities is low, we might have to compute an exponential number of analyses to get at the meanings.

Recall from chart parsing the concept of a chart. If the string consists of \( n \) words (or whatever units we choose) then there are \( n + 1 \) positions (points between the words), and for each pair \((i, j)\) such that \( 0 \leq i \leq j \leq n \) we compute the possible nonterminals that span the string between position \( i \) and \( j \). In the present context, the string is partitioned such that variables consist of a unit, and every operator symbol is a unit. Thus, `/p0∧pvp2/` is partitioned into the list \([p0; ∧; p; ∨; p2]\) and
parsed according to the context free following grammar.

(1.20) \[ P \rightarrow p \vec{x} | \neg P | P \lor P | P \land P | P \rightarrow P \]

This yields in addition to the constituents of length 1 a constituent between positions 0 and 3 (\( /p0\land p/ \)), a constituent between 1 and 4 (\( /p\lor p2/ \)) and a constituent between 0 and 4 (the entire string). Since there are only boundedly many category symbols, the chart is of size \( cn^2 \) for some constant number \( c \). Each cell takes \( n \) steps at most to fill, so the total runtime complexity is \( O(n^3) \).

However, adding the semantics changes everything. First of all, a chart parse is done in the same way as before. The only difference is that the chart no longer is a set of elements of the form \( \langle \vec{u}, N \rangle \) where \( \vec{u} \) is a substring of \( \vec{x} \) (between certain positions) and \( N \) a nonterminal (such that \( \vec{u} \) is of category \( N \) in the CFG); rather, it now contains signs, namely pairs of the form \( \sigma = \langle \vec{u}, m \rangle \) such that \( \vec{u} \) is a substring of \( \vec{x} \) and \( m \) a meaning such that \( G \) derives \( \sigma \). Now, however, there no longer is a bound on the number of elements needed to store in a cell. That is to say, given \( e \) there is no upper bound on the number of \( m \) such that \( \langle e, m \rangle \) is in \( L(G) \). In the languages with a fixed valuation of course the meaning can only be 0 or 1. So lets change things a little bit.

**Example 9.** **Partial Valuation Logic.** Let \( X \) be the set of all functions from finite sets of variables to \{0, 1\}. A set \( R \subseteq X \) is **homogeneous** if \( \text{dom} f = \text{dom} g \) for every \( f, g \in R \). (Any two functions are defined on the same subset of \( X \).) For a homogeneous set \( R \) we may also write \( \text{dom} R \) for the set dom \( f \) of any of its members. Then for a subset \( Y \subseteq X \) containing \( \text{dom} R \) put \( R \uparrow Y := \{ f : Y \rightarrow \{0, 1\} : (f \uparrow \text{dom} R) \in R \} \). Thus, we can artificially expand a homogeneous set to a bigger domain on need. Let \( -f \) be defined by \( (-f)(x) := -f(x) \). Now put

\[
\begin{align*}
-R := & \{(-f) : f \in R \} \\
R \cap S := & (R \uparrow (\text{dom} R \cup \text{dom} S)) \cap (S \uparrow \text{dom} R \cup \text{dom} S) \\
R \cup S := & (R \uparrow (\text{dom} R \cup \text{dom} S)) \cup (S \uparrow \text{dom} R \cup \text{dom} S) \\
R \setminus S := & ((-R) \uparrow (\text{dom} R \cup \text{dom} S)) \cup (S \uparrow \text{dom} R \cup \text{dom} S)
\end{align*}
\]
This gives rise to a new kind of semantics for propositional logic.

\[ S(p, \vec{x}) := \{ (p, \vec{x}, 1) \} \]
\[ S(f_\land)(R) := -R \]
\[ S(f_\lor)(R, S) := R \cap S \]
\[ S(f_\land)(R, S) := R \cup S \]
\[ S(f_\lor)(R, S) := R \supset S \]

(1.22)

For example, the interpretation of /p/ is the function defined on just \( p \), with value 1. Similarly, /p_0/ is interpreted as the function defined only on \( p_0 \) with value 1. /f_\lor p_0 p_0/ is interpreted as all functions defined on both \( p \) and \( p_0 \) such that either they assign 1 to \( p \) or 1 to \( p_0 \).

The previous semantics has infinitely many meanings, and all of them are meanings of some formula. Now, if we do a chart parse in the standard bracketed notation this may turn out to increase the cost of computation, since the objects may become large. Yet, in addition to that we also must worry about the number of items to be stored in a single cell of the chart. As an example consider again the bracket free notation. The expression

(1.23) \[ p \lor p_0 \land p_1 \land p_2 \cdots \]

has linearly many analysis terms with different semantics (the scope of \( \lor \) can be put almost anywhere). Worse, there are formulae which have exponentially many different meanings. Here is an example. The formula /\neg p_0 p_0/ has two different analysis terms, and they give us two different meanings. Consider an \( n \) fold disjunction of such formulae on pairwise disjoint sets of variables. The number of (semantically different) readings is at least \( 2^n \).

(1.24) \[ \neg p_0 p_0 \land \neg p_1 p_2 \land \neg p_3 p_4 \cdots \]

The grammar is context free. Yet, to determine all possible meanings of an expression can in the worst case only be done in time exponential in the length of the string.

Let me return briefly to the case of ordinary bracket notation and the computation of meanings. The objects are functions from a set of variables to \{0, 1\}. The number of variables is linear in the length of the formula (slightly less than that, as the index of the variable takes up some space, too). If there are \( n \) variables,
there are up to $2^n$ many functions. Each function can be represented as a bit vector of length $n$. The total coding cost is length of $\vec{x}$ (we need to know the identity of the variables) plus $n2^n$, to store all the functions that belong to that set. So the cost of computing semantics as given above is exponential even when the language is uniquely readable. Unfortunately, this argument suffers from a shortcoming. The function set already has a code, namely $\vec{x}$ itself, which is linear in its own length! So, we may take $\vec{x}$ as a code for its semantics.

If this argument can be used for the bracketed notation why is not possible to use it in the unbracketed version? One reason is that when $\vec{x}$ is ambiguous it can no longer serve as a code of its own meaning. There is more than one. It is true that in the unbracketed notation we can use a suitable bracketed version as a code to its meaning under an analysis term. And so the computation does not have to use the sets given above, it might use some refined version of the string as a substitute. Now however the computation still has to go through exponentially many cases, and so the shortcut has not yielded much of a benefit.

1.5 Changing the Abstract Syntax

Finally, let us look at situations where the abstract syntax is different. As before, I present some examples, some trivial, some less so.

**Example 10.** Adding new functions. We add a new function in the semantics: let $+$ be defined as follows.

\[
\begin{array}{c|cc}
+ & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

(1.25) (This function is otherwise known as XOR or exclusive or.) Introduce to $\Omega$ a new function, $f$, with $\Omega(f) = 2$. The interpretation functions are spelled out in the predictable way: $f$ is interpreted by the function $+$, and in syntax by $\langle \vec{x}, \vec{y} \rangle \mapsto \langle \vec{x} + \vec{y} \rangle$. The new language presents a different abstract syntax: and this is because the functions denoted by the symbols in the set $F$ has changed.

Notice that it is not enough to just add a function to the semantics. We are not concerned with the question of what functions exist. We are concerned with
what functions are interpretations of some functions symbol that we have (cf.
Definition \[1.14\]).

We look in detail at a different case. Suppose we defined a function

\begin{equation}
\text{let et } x \ y = x \ && y
\end{equation}

This looks as if /et/ is just the same as /&&/. It isn’t. The reason is a behaviour
of the infix operators: they are not defined if just one operator is present. In other
words, the expressions /true \ &&/ and /&& false/ are illegal. On the other hand,
/et true/ is a legal expression. Its value is a function, the identity function. This
is because /et true x/ is synonymous with /x/. There are basically two ways
to explain this behaviour. One is to blame this on the syntax, the other on the
semantics. If it is a semantic issue we expect the meaning of the two expressions
to be different. But according to OCaml they are not different. It asserts that /&&/
is of type bool→bool→bool. (You have to type /&&( )/ to get that answer. /&&/ alone
is not a legal expression.) Thus the problem is that of syntax; the syntactic
functions are such that they allow us to form the expression /et true/ but not the
epression /true &&/. We could alternatively think that the answer is semantic. In that case we
could that say that /&&/ is interpreted as a binary function from truth values to truth values, while /et/ is interpreted as a unary function. So, /&&/ is of type
bool*bool→bool, while /et/ is of type bool→bool→bool/. The difference, however, between these two solutions is not attestable. Due to the syntactic limitation we can never see wether /&&/ is of type bool*bool→bool of type
bool→bool→bool/. This is because syntactically we are required to fill both argument slots; and when we do the result is a boolean in both cases.

At this point it is perhaps necessary to get deeper into the issue of typed uni-
verses. Notice that OCaml assigns a type to every kind of expression. The types
are an explicit way to check whether the proposed operation is meaningful. We
start with basic types, in our case just bool, but we shall soon have more types,
and a map that assigns to every type a range of interpretations. For example, bool has the range \{0, 1\}, int the set of integers, and so on. From types we build more
complex types as follows.

1. If \(\alpha\) and \(\beta\) are types with interpretation \(M_\alpha\) and \(M_\beta\) then \(\alpha \rightarrow \beta\) is a type
with interpretation

\begin{equation}
(M_\beta)^{M_\alpha} := \{ f : M_\alpha \rightarrow M_\beta \}
\end{equation}
If \( \alpha \) and \( \beta \) are types with interpretation \( M_\alpha \) and \( M_\beta \), respectively, then \( \alpha \times \beta \) is a type with interpretation \( M_\alpha \times M_\beta \).

There are more type constructors (list for example) but this should suffice. The ordinary function is now an object in its own right. However, it can also be applied to an argument. There is a general rule: if \( \vec{x} \) is an expression denoting \( f \) of type \( \alpha \to \beta \) and \( \vec{y} \) an expression denoting \( g \) of type \( \alpha \) then \( \vec{x} \cdot \vec{y} \) is an expression denoting \( f(g) \) of type \( \beta \).

It is customary in connection with combinatory logic to make the notion of function application more visible in notation. I shall write \( f \cdot g \) in place of \( f(g) \). This is to remind us of the fact that function application is the result of an operation that takes two objects and returns another. This operation is partial. It is only defined if \( f \) is a function of a type that can take \( g \) as an argument. However, it is not of a particular type itself. It is type abstract. In OCaml notation, its “type” would be written \( ('a \to 'b) \to 'a \to 'b \). If you want OCaml to give you that, do the following. Issue \( \text{let app x y = x(y)} \). It will then tell you that the type of \( \text{app} \) is \( ('a \to 'b) \to 'a \to 'b \). In OCaml, there is a module called Pervasives containing an array of other such functions. For example, \( \text{let} \) \( /= \) \( \text{can be used in between any objects of identical type. Its “type” is} \ 'a \to 'a \to \text{bool} \).

The semantics for such a language is more complicated. I shall give two versions of semantics, both in some sense equivalent. The first says that rather than having one set of objects, we have a family \( \langle M_\alpha : \alpha \in \text{Typ} \rangle \) of sets, with \( \text{Typ} \) the set of all types. And we interpret a function symbol by a family of maps between these sets. For example, \( \text{app} \) is interpreted by \( \bullet \), which is a family \( \langle \bullet_{\alpha \to \beta} : M_{\alpha \to \beta} \times M_\alpha \to M_\beta : \alpha, \beta \in \text{Typ} \rangle \). Very quickly this way of doing things becomes very clumsy in terms of notation. And it does not solve all problems (for example, division is not always defined). Therefore, another solution is to allow the semantic (and syntactic) operations to be partial. It is clear how to turn the typed universe into a universe with partial functions. Just let \( U := \bigcup_{\alpha \in \text{Typ}} M_\alpha \), and \( \bullet = \bigcup_{\alpha, \beta \in \text{Typ}} \bullet_{\alpha \to \beta} \). Given a pair \( \langle x, y \rangle \) we first establish their types \( \gamma \) and \( \alpha \). We have assumed that these types are unique. If there is a type \( \beta \) such that \( \gamma = \alpha \to \beta \) then the operation is defined and gives \( x \bullet \gamma \). Otherwise it is undefined.
1.6 Tensed Propositional Logic

There are several ways in which the language of propositional logic can be enriched to capture more of the semantics of natural language. I shall present one way before dealing with predicate logic. This is tense logic. Tense logic allows to talk about truth in time. Sentences normally have no eternally fixed truth value. For example, /The door is open./ is sometimes true sometimes false. Only some rare exceptions exist (laws of physics and mathematics). The way to handle this situation semantically is as follows. A frame is a structure of the form \( \mathfrak{W} = (W, \prec) \), where \( W \) is a set (called the set of worlds) and \( \prec \subseteq W \times W \) a binary relation. A valuation on the frame \( \mathfrak{W} \) is a function \( \nu \) that assigns to every world \( w \) a valuation in the standard sense. So, given \( w \in W \), \( \nu(w) \) is a function from variables into \( \{0, 1\} \). We write \( \langle \mathfrak{W}, \nu, w \rangle \models \varphi \) to say that the proposition \( \varphi \) holds at \( w \) under the valuation \( \nu \). The following is clear from what we have defined so far:

\[
\langle \mathfrak{W}, \nu, w \rangle \models \vec{x} : \iff \nu(w)(\vec{x}) = 1 (\vec{x} \text{ a variable})
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (\neg \vec{x}) : \iff \langle \mathfrak{W}, \nu, w \rangle \not\models \vec{x}
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (\vec{x} \land \vec{y}) : \iff \langle \mathfrak{W}, \nu, w \rangle \models \vec{x} \text{ and } \langle \mathfrak{W}, \nu, w \rangle \models \vec{y}
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (\vec{x} \lor \vec{y}) : \iff \langle \mathfrak{W}, \nu, w \rangle \models \vec{x} \text{ or } \langle \mathfrak{W}, \nu, w \rangle \models \vec{y}
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (\vec{x} \rightarrow \vec{y}) : \iff \langle \mathfrak{W}, \nu, w \rangle \models \vec{x} \implies \langle \mathfrak{W}, \nu, w \rangle \models \vec{y}
\]

Next we add four unary operators: \( F, P, G \) and \( Q \). Their semantics is as follows.

\[
\langle \mathfrak{W}, \nu, w \rangle \models (F \vec{x}) : \iff \text{for all } v \text{ s.t. } w \prec v : \langle \mathfrak{W}, \nu, v \rangle \models \vec{x}
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (P \vec{x}) : \iff \text{for all } v \text{ s.t. } v \prec w : \langle \mathfrak{W}, \nu, v \rangle \models \vec{x}
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (G \vec{x}) : \iff \text{for some } v \text{ s.t. } w \prec v : \langle \mathfrak{W}, \nu, v \rangle \models \vec{x}
\]
\[
\langle \mathfrak{W}, \nu, w \rangle \models (Q \vec{x}) : \iff \text{for some } v \text{ s.t. } v \prec w : \langle \mathfrak{W}, \nu, v \rangle \models \vec{x}
\]

**Example 11.** Let \( W = \{0, 1, 2, 3\} \), \( i \prec j \) iff \( i < j \), and the valuation be as in the picture (variable are written at a world where they are true. Notice that arrows
indicate only next neighbours, the actual relation is the transitive closure.)

\[
\begin{array}{cccc}
p & p & p & p \\
0 & 1 & 2 & 3
\end{array}
\]

Then we have \( \langle W, \nu, i \rangle \models (Fp_0) \), as for all \( i > 1 \) (the only choices being 2 and 3) we have \( \langle W, \nu, i \rangle \models p_\emptyset \). Also, for every \( j \) we have \( \langle W, \nu, j \rangle \models (\neg p_1) \rightarrow (Fp_0) \). A somewhat trickier example is this. \( \langle W, \nu, 0 \rangle \models (P(p \land \neg p)) \). For since there is no \( i < 0 \), all worlds \( i \) such that \( i < 0 \) make both \( p \) and \( \neg p \) true.

There are natural language equivalents for all these operators.

1. “\((F\bar{x})\)” is “it will always be the case that \( \bar{x} \)”
2. “\((P\bar{x})\)” is “it has always been the case that \( \bar{x} \)”
3. “\((G\bar{x})\)” is “it will be the case that \( \bar{x} \)”
4. “\((F\bar{x})\)” is “it was the case that \( \bar{x} \)”

The tenses in English (future, present and past) are of the existential sort. /Hans will bring me tea./ means that at some point in the future /Hans is bringing me tea./ is true, not that it is true at all such points.

In principle, I should go through the same process as before, defining a signature to incorporate the new possibilities to build formulae, and then defining rules to obtain signs. Moreover, it seems that we should define \( \models \) to be a relation between a triple \( \langle W, \nu, w \rangle \) and an analysis term rather than a particular syntactic object, here a string. Yet, since we know the language above is uniquely readable, this is a superfluous addition to make and is only needed when we deal with natural language. And then I will have to say something about the way the approach can be “algebraified”. For now, however, let us refine the apparatus somewhat.

First, we should note that there are certain formulae that can never be true, as there are formulae that can never be false. The same is actually true for propositional logic without tense operators. For example, \( \langle p \land \neg p \rangle \) is never true under
any valuation while \((p \lor \neg p))\) always is. The first sort of formula is called a \textbf{contradiction}, the second kind \textbf{tautology}. It is easy to see that if \(x\) is a contradiction, \((\neg x)\) is a tautology, and if \(x\) is a tautology then \((\neg x)\) is a contradiction. Anything that is not a contradiction can be made true under a valuation. The tableau calculus was a method to see first of all whether a formula is a contradiction, and if not to find a valuation that makes it true. There exist such calculi for modal logics, but we shall not go into that matter here. The complication is that if a formula is not contradictory we need to guess a frame and then a valuation on top of that. Frames however can be of arbitrary size and it is not always possible to know in advance how big a frame we need to make a formula true.

I used above the word “logics”. This indicates that there can be several. Indeed, when we look at all the frames and ask: what are the tautologies of that set, that is, what is the set of formulae that come out true no matter what frame and what valuation we pick? This is the set known as \(Kt\) (“Kripke tense”). However, for our purposes the set of all frames is too general. For it allows all sorts of things that are impossible with time, such as: looping back into the past, branching in the past, and so on. In what is to follow I will therefore refine the class of frames we shall look at as follows. (Notice that this by far not the only way to define tense frames, I am just making a choice.)

\textbf{Definition 1.15} A \textbf{tense frame} is a frame \(\langle W, \triangleleft \rangle\) where \(\triangleleft\) is transitive, and linear in both directions.

This allows for all sorts of choices. \(\langle \mathbb{N}, < \rangle, \langle \mathbb{Z}, < \rangle, \langle \mathbb{Q}, < \rangle, \langle \mathbb{R}, < \rangle\) are among the objects, as are finite frames such as \(\langle\{0, 1, \cdots, n-1\}, <\rangle\).

\textbf{Example 12.} This continues Example 11. We now prepare for a grammar that generates signs in the formal sense. To be able to do this we have to speak about meanings. While in classical propositional logic a proposition was assigned a truth value, now a proposition is assigned a set of worlds, in this case a subset of \(\{0, 1, 2, 3\}\). The set \(A\) that a proposition \(x\) is assigned turns to be the set \(\{w : \langle W, \nu, w \rangle \models x\}\). It can now be checked that if \(x\) is a proposition with value \(A\) and \(y\) a proposition with value \(B\), \((x \land y)\) is a proposition with value \(A \cap B\). Similarly for the connectives \(\lor\), \(\rightarrow\) and \(\neg\).

There are 16 subsets. On this set, \(f_\Delta\) is interpreted as complement, \(f_A\) as conjunction, \(f_\mathcal{O}\) as union and \(f_*\) as the operation \((A, B) \mapsto (W - A) \cup B\). The
tense operators are defined as follows. (The name of the semantic operation is given first, followed by its syntactic counterpart in brackets.)

\[
\begin{array}{|c|ccccc|}
\hline
& \Box (F) & \Diamond (G) & \Box (P) & \Diamond (H) \\
\hline
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\{0\} & \emptyset & \emptyset & \{0, 1\} & \{1, 2, 3\} \\
\{1\} & \emptyset & \{0\} & \emptyset & \{2, 3\} \\
\{2\} & \emptyset & \{0, 1\} & \emptyset & \{3\} \\
\{3\} & \{2, 3\} & \{0, 1, 2\} & \emptyset & \emptyset \\
\{0, 1\} & \emptyset & \emptyset & \{0, 1, 2\} & \{1, 2, 3\} \\
\{0, 2\} & \emptyset & \{0, 1\} & \emptyset & \{1, 2, 3\} \\
\{1, 2\} & \emptyset & \{0, 1\} & \emptyset & \{2, 3\} \\
\{1, 3\} & \emptyset & \{0, 1, 2\} & \emptyset & \{2, 3\} \\
\{2, 3\} & \emptyset & \{0, 1, 2\} & \emptyset & \{3\} \\
\{0, 1, 2\} & \emptyset & \emptyset & \{0, 1, 2, 3\} & \{1, 2, 3\} \\
\{0, 1, 3\} & \emptyset & \emptyset & \{0, 1, 2\} & \{1, 2, 3\} \\
\{0, 2, 3\} & \emptyset & \emptyset & \{0, 1, 2\} & \{1, 2, 3\} \\
\{1, 2, 3\} & \emptyset & \emptyset & \{0, 1, 2\} & \{2, 3\} \\
\{0, 1, 2, 3\} & \emptyset & \emptyset & \{0, 1, 2, 3\} & \{1, 2, 3\} \\
\hline
\end{array}
\]

Now we spell out the full grammar.

\[(1.32) \quad F = \{p, \emptyset, \ldots, 9, f_-, f_+, f_y, f_\ast, f_\phi, f_\psi, f_\theta\}\]

with \(\Omega'(f_\phi) = \Omega'(f_\psi) = \Omega'(f_\theta) = 1\). We choose a map \(\nu : V \to \wp(W)\). Based on this map we define the value of complex propositions as follows.

\[
(C \times M)(p\bar{x}) := \langle p\bar{x}, \nu(p\bar{x}) \rangle
\]

\[
(C \times M)(f_\ast)((\bar{x}, A)) := (\neg \bar{x}, W - A)
\]

\[
(C \times M)(f_\wedge)((\bar{x}, A), \langle \bar{y}, B \rangle) := (\bar{x} \wedge \bar{y}, A \cap B)
\]

\[
(C \times M)(f_\vee)((\bar{x}, A), \langle \bar{y}, B \rangle) := (\bar{x} \vee \bar{y}, A \cup B)
\]

\[(1.33) \quad (C \times M)(f_\ast)((\bar{x}, A), \langle \bar{y}, B \rangle) := (\bar{x} \ast \bar{y}, (W - A) \cup B)
\]

\[
(C \times M)(f_\phi)((\bar{x}, A)) := (F\bar{y}, \Box A)
\]

\[
(C \times M)(f_\psi)((\bar{x}, A)) := (F\bar{y}, \Diamond A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]

\[
(C \times M)(f_\theta)((\bar{x}, A)) := (F\bar{y}, \lozenge A)
\]
For example,

\[
\begin{align*}
  h(f_{\neg f} p^0 f_{\neg p^2}) \\
  = (C \times M)(f_{\neg f} (h(f_{\neg p^2}), h(f_{\neg p^2}))) \\
  = (C \times M)(f_{\neg f})((C \times M)(f_{\neg f})(h(p^0)), (C \times M)(f_{\neg f})(h(p^2))) \\
  = ((\forall \neg p^2), (\exists[2, 3]) \cap \{0, 1, 2, 3\} - \{3\}) \\
  = ((\forall \neg p^2), \{1, 2, 3\} \cap \{0, 1, 2\}) \\
  = ((\forall \neg p^2), \{1, 2\})
\end{align*}
\]

(1.34)

It is clear now, we an analysis of the sentence like the following will proceed.

\[
(1.35) \quad \text{It will be the case that cheese is white or it was} \\
\quad \text{always the case that squirrels are fast.}
\]

It will be the case that cheese is white or it was always the case that squirrels are fast.

The analysis reaches down to the basic propositions (cheese is white/, squirrels are fast/), and the operators are translated as /it will be the case that/ and /it was always the case that/. This analysis faces two problems in connection with natural language. The first is of purely morphological nature. It is not always possible to render the operator as a string. For example, the sentence /John will sing/ is obviously semantically the same as /it will be the case that John sings/, yet it is not clear how we shall derive its form. For that we need to derive /will sing/ from, say, /sings/ or some underlying form (in this case /sing/). In the present case we simply need to have a separate semantic analysis of the VP (which I shall provide in the next chapter). Additionally, however, we shall have to tackle some morphological issues. In these lectures, however, we shall discard such problems, assuming that if there is a semantic decomposition, we can also find a suitable morphological decomposition to accompany it (in the worst case using lots of empty elements).

The second question we have to deal with is that of the syncategorematicity of the operators. If you look closely you realise that the expression /it will be the case that/ has no meaning of its own. Instead, it is inserted as the effect of a unary function that takes a string and attaches something to it. We are trained to see things differently: we think that /it will be the case that/ is the exponent of a sign, the corresponding meaning meaning of which is “it will be the case that”. However, the latter function is nowhere in sight. In λ-calculus it is written \( \lambda p. \Diamond p \). Thus, if we remained true to our intuition we should set things
1.6. Tensed Propositional Logic

up as follows. The symbols $f_F$, $f_G$, $f_P$, and $f_Q$ are no longer unary; instead they are constants (zeroary). They are mapped to the following signs.

\[
\begin{align*}
    h(f_F) &:= \langle \text{it will be the case that} /, \lambda p.\Diamond p \rangle \\
    h(f_G) &:= \langle \text{it will always be the case that} /, \lambda p.\Box p \rangle \\
    h(f_P) &:= \langle \text{it was the case that} /, \lambda p.\Diamond p \rangle \\
    h(f_Q) &:= \langle \text{it has always been the case that} /, \lambda p.\Box p \rangle
\end{align*}
\]

Furthermore, we shall agree that there are two binary function symbols, $a <$ and $a >$, that the following is their interpretation:

\[
\begin{align*}
    (C \times O)(a_<)(\langle \vec{x}, m \rangle, \langle \vec{y}, n \rangle) &:= \langle \vec{x} \omega \vec{y}, m(n) \rangle \\
    (C \times O)(a_>)(\langle \vec{x}, m \rangle, \langle \vec{y}, n \rangle) &:= \langle \vec{x} \omega \vec{y}, n(m) \rangle
\end{align*}
\]

In the semantics we can now use the fact that meanings are typed. For the term $m(n)$ (alias $m \bullet n$) is defined only if $m$ is of type $\alpha \to \beta$ for some $\beta$ and $\alpha$ such that $n$ has the type $\alpha$. Thus a lot of ungrammatical expressions are banned simply because they cannot be evaluated in the semantics. This is, if you remember, the way Montague proceeds. There is however one important difference: there are no categories! Thus, as matters stand you are free to combine /it is the case that/ to the right and to the left, giving you

\[
\begin{align*}
    \text{it will be the case that John sings} \\
    \text{John sings it will be the case that}
\end{align*}
\]

This is because if $a>_f f_G t$ unfolds to some sign $\langle \text{it will be the case that} \vec{x}, \Diamond \varphi \rangle$ then $a_<f_f f_G t$ unfolds to $\langle \vec{x} \text{ it will be the case that}, \Diamond \varphi \rangle$. Of course, one may immediately think of getting rid of the operation $a_$, but that is a bad idea. Recall that eventually we want to do the same to conjunction too (can you see how that will be done?).

Another, quite effective approach, is to make also the concatenative function partial. In the case of the operators above it will be defined only if you put them to the left. This is what we shall do here. So, $C(a_<)(\vec{x}, \vec{y})$ is undefined if $\vec{y} = /\text{it is the case that}/$, for example. The partiality is more general than the categorial system employed by Montague. It allows for finer distinctions that can be made, but by and large the systems are equivalent. Let me therefore fix things for further reference.
Example 13. Montague style semantics. In Montague style semantics (not exactly his own, but very close) there are only two binary function symbols, \( a_\succ \) “forward application” and \( a_\prec \) “backward application”. Their interpretation is fixed as given in (1.37). To make notation easier we use the following shorthand. For signs \( \sigma \) and \( \tau \) we write \( \sigma \bullet_\succ \tau \) for the forward application of \( \sigma \) to \( \tau \), and \( \sigma \bullet_\prec \tau \) for the backward application of \( \tau \) to \( \sigma \) (both are accompanied by concatenation with a blank in between). The reader will surely have noticed that if forward application is licit backward application is not, and conversely, if backward application is licit, forward application is not. Hence, we often write \( \sigma \bullet \tau \), to denote whichever of the two is defined, and undefined else.

1.7 Satisfiability

In the previous section we were concerned with evaluating a formula in a particular model. In this section we shall look at the problem of finding out for a given proposition whether or not it can at all be satisfied. We have met this question before. Here it shall be discussed at somewhat greater length.

We start with a seemingly different topic, namely arguments.

Definition 1.16 An argument is a pair \( \Delta/\varphi \), where \( \Delta \) is a set of propositions, the premisses, and \( \varphi \) a proposition, called the conclusion. The argument is valid if for every model \( \mathfrak{M} \), if every member of the premiss is true, so is the conclusion.

This definition is somewhat less rigorous, but at an informal level clear enough. An argument is written like this:

(1.40) \[
\begin{align*}
\text{John sings or dances.} & \quad \text{John does not sing.} \\
\therefore \text{John dances.}
\end{align*}
\]

The triple dots ‘\( \because \)’ signal the conclusion (“therefore”). The premisses are written above the line.

From the definition it is obvious that either an argument is valid, or there is a model where all premisses are true but the conclusion is not. So, in order to decide whether an argument is valid we need to show that no such model exists; to show that it is invalid we need to produce such a model. We have previously
1.7. Satisfiability

illustrated the tableau method, which was a method to decide whether or not a single formula has a model. These two methods are interconnected. To establish this connection let me first point out a special case, $\Delta = \emptyset$. Suppose $\Delta$ is empty; then the argument $\Delta/\varphi$ is valid if there is no model in which $\varphi$ is false; in other words, if $\varphi$ is a tautology. The argument is invalid if $\varphi$ is not a tautology, that is, if $\neg \varphi$ is satisfiable. So, we have seemingly generalised the notion of satisfiability.

Now notice the following. Suppose we have an argument of the form $\Delta \cup \{\theta\}/\varphi$. Then this is actually the same as the argument $\Delta/\theta \rightarrow \varphi$. Sameness here means that validity is preserved both ways. Assume that $\Delta \cup \{\theta\}/\varphi$ is valid. Take any model $\mathcal{M}$ that makes $\Delta$ true. Two cases arise. If it makes $\{\theta\}$ true, it also makes $\varphi$ true, by assumption. Then $\theta \rightarrow \varphi$ is true as well. If it does not make $\theta$ true, however, it makes $\theta \rightarrow \varphi$ true. So, $\Delta/\theta \rightarrow \varphi$ is valid. Assume now that the latter is valid. Take a model $\mathcal{M}$ that makes $\Delta \cup \{\theta\}$ true. Then it makes $\Delta$ true, so it makes $\theta \rightarrow \varphi$ true (by assumption). And since $\theta$ is true, so is now $\varphi$.

This fact is known as the Deduction Theorem. It assures us that an argument can be reduced to a single formula. And the argument is valid if and only if that formula is a tautology. The models of the last section also had that property.

Logic is mostly considered to be an axiomatization of the tautologies. This is to say that it defines a procedure to generate all tautologies. (This is based on terms rather than string objects, but I shall present it here in the latter form.) The way it is done is by giving a number of schematic formulae:

1. $((F(\varphi \rightarrow \chi) \rightarrow (F\varphi) \rightarrow (F\chi)))$,
2. $(\varphi \rightarrow (F\varphi))$,
3. $(\varphi \rightarrow (P\varphi))$.

Everything that is obtained by replacing the letters by true formulae is a tautology. In addition, there are three rules.

1. If $(\varphi \rightarrow \chi)$ and $\varphi$ are tautologies so it $\chi$.
2. If $\chi$ is a tautology then so is $(F\chi)$.
3. If $\chi$ is a tautology then so is $(P\chi)$. 
In fact, Kt can be so obtained from all propositional tautologies using these above axioms and rules (and the propositional tautologies in turn can be axiomatically characterised). This system allows to generate each and every tautology in a finite number of steps.

However, there are formulae that are tautologies by the fact that the structures are special. One such formula is (in informal notation)

\[ \Diamond \Diamond \varphi \rightarrow \Diamond \varphi \lor \varphi \lor \Diamond \varphi \]

In ordinary language it comes down to the validity of the following argument.

\[ \text{it will be the case that it was the case that } A \]
\[ \text{either it was the case that } A \text{ or } A \text{ or } \]
\[ \text{it will be the case that } A \]

To see why this is true, take a model \( \langle W, \prec, \nu, w \rangle \) such that

\[ \langle W, \nu, w \rangle \models \Diamond \Diamond \varphi \]

Then there is \( x \models \Diamond \varphi \), and so a \( y \) such that \( y \models \varphi \). But \( \prec \) is linear, and so either \( w \prec y \) in case \( w \models \Diamond \varphi \); or \( w = y \) in which case \( w \models \varphi \); or \( y \prec w \), and so \( w \models \Diamond \varphi \).

So, in order to be able to say whether a given argument is valid or whether a given formula is a tautology we must do the following. (1) We must axiomatise the tautologies; this allows to effectively enumerate them. Thus if the given formula is a tautology we simply have to wait until it shows up in the list. However, we do not necessarily know how long we need to wait. Therefore, (2) we have a procedure to enumerate all non tautologies as well. One such method (a very crude one) is as follows.

We suppose that the logic is finitely axiomatised, and that it is determined by all its finite structures. Then we simply list all triples \( \langle W, \nu, \varphi \rangle \), where \( W \) is a finite graphs, \( \nu \) is a partial valuation over the variables of \( \varphi \), and \( \varphi \) is a formula. Next we look at whether the graph satisfies all the axioms of the logic (finitely many, it is a finite task). When it passes the test we check whether the value of \( \varphi \) under \( \nu \) is empty. If not, \( \varphi \) is satisfiable (and its negation not a tautology). If it is, we know nothing, unfortunately.

However, our most natural model is that of the real line. And it is not immediately clear that its logic has this property. It satisfies a rather curious looking
1.7. Satisfiability

formula:

\[(1.44) \quad p \land \Box(p \rightarrow \Box p) \land \Box(p \rightarrow \Diamond p) \rightarrow (\Box p \lor \Diamond (\neg p \land \Box p))\]

It says that if the value of \( p \) has no last point, and is open to the left then either \( p \) is always true or there is a first non-\( p \) point. If this holds, the linear order is said to possess no Dedekind cuts. So, matters are delicate; but the logic of the real line does have the finite model property, though this takes some effort to prove.
1. Propositional Calculus
Chapter 2

Predicate Calculus

2.1 Basic Clauses

In this chapter we shall start with predicate logic; this means that we start analysing propositions into smaller units. Less and less I shall detail the algebraic setup and start to concentrate on issues of evaluation instead. I shall occasionally make reference to types. The notation is as follows. There is a new type $e$, which is Montague’s type of objects. This is not an abstract type; it is a constant, though it cannot be found of course in OCaml. As for Montague’s $t$, the type of truth value, I shall use $\text{bool}$ here. The way to implement $e$ is to think of a suitable type to use, for example $\text{string}$ and then issue \texttt{type e = O of string} to OCaml. This will allow to code objects as things of the form $0 \vec{x}$, where $\vec{x}$ is a string. Type declarations used below are understood in this way.

The first example will consist in basic intransitive sentences of the following kind.

(2.1) Paul is asleep.
(2.2) Some cat is asleep.
(2.3) Every cat is asleep.

A model will consist in a pair $(D, I)$ such that $D$ is a set and $I$ is a function that assigns to the common nouns and adjectives a subset of $D$, and assigns to proper names a member of $D$. 

41
Example 14. Let \( D = \{a, b, c, d\} \), \( I(\text{Paul}) := a \), \( I(\text{asleep}) := \{a, c, d\} \), \( I(\text{cat}) := \{a, b, c\} \). In the model \( \langle D, I \rangle \), (2.1) is true (\( a \) is Paul and asleep), (2.2) is true (since Paul is a cat), and (2.3) false (\( b \) is a cat and not asleep).

To formalise our judgements, we need a precise formulation of the value that complex expressions have. To do that, we take first a middle road and explicate the truth conditions for the sentences and then abstract this to the level of analysis terms.

Example 15. Syllcogist. In scholastic logic, arguments were done on stylised surface form. Propositions had four forms.

(2.4) Some \( A \) is \( P \). (existential affirmative)
(2.5) Some \( A \) is not \( P \). (existential negative)
(2.6) All \( A \) are \( P \). (universal affirmative)
(2.7) No \( A \) is \( P \). (universal negative)

Let \( I \) be the function assigning subsets of the domain to the terms. Then it was required that

1. (2.4) is true iff \( I(A) \cap I(P) \neq \emptyset \).
2. (2.5) is true iff \( I(A) - I(P) \neq \emptyset \).
3. (2.6) is true iff \( I(A) \neq \emptyset \) and \( I(A) \subseteq I(P) \).
4. (2.7) is true iff \( I(A) \neq \emptyset \) and \( I(A) \cap I(P) = \emptyset \).

In this tradition, a universal sentence always has existential import. If you say that all cats are asleep you are implying that there is a cat. To modern ears this sounds false, but as a convention it is just as good as the modern one (and might even be closer to our preformal understanding).

One particular way to formulate the language is to introduce four binary functions, \( f_a, f_b, f_c \) and \( f_d \), which introduce the quantifier (/some/, /all/, /no/) and the copula (/is/, /are/) and has the semantics as shown above. This means that the only constituents of (2.4) are \( A \) and \( P \). Thus there is no VP, as /is \( P\) is not a
constituent. There are reasons why we need to depart from this analysis. One is that there is no copula in the presence of intransitive verbs.

(2.8) Paul runs.
(2.9) Some cat sleeps.

Instead, one has chosen to have the quantifier rule insert only the quantifier, while the copula is inserted to make a noun or an adjective into an intransitive verb. In this analysis, /some cat/ is a constituent and has a meaning. If we look at its use we see that the meaning must be a function that takes intransitive verb meanings (= subsets of the domain) and returns a truth value. In Montague semantics, sets were considered functions into \{0, 1\} (= bool), so given this the type now is \(e \rightarrow \text{bool} \rightarrow \text{bool}\). Now, this is the meaning of /some cat/, and it is the combination of /some/ and /cat/, the latter being of type \(e \rightarrow \text{bool}\). So the quantifier has the type \((e \rightarrow \text{bool}) \rightarrow (e \rightarrow \text{bool}) \rightarrow \text{bool}\). The meaning is defined by

(2.10) \(\lambda f. \lambda g. (\exists z)(f(z) \land g(z))\)

This says: there is some \(z\) of which both the first argument and the second argument is true.

The advantages of this view become apparent when we look at transitive sentences.

(2.11) Some man loves every woman.

Here it is clear that we do not want the quantifier to introduce the copula. Rather, we want to think of it as forming a constituent together with the noun phrase. I shall give various ways to continue.

**Example 16. Distinct Subject and Object Quantifier.** The verb /loves/ is interpreted as an object loves' of type \(e \rightarrow e \rightarrow \text{bool}\). The first argument is the object, the second the subject. So, loves'(a)(b) means “b loves a”. The constituent structure is unequivocally S(VO). So, we must combine /loves/ with /every woman/. The entire phrase is further combined with /some man/. To achieve this, the meaning of /some/ must be as given above. So the meaning of /loves every woman/ is

(2.12) \(\lambda y. (\forall x)(\text{woman}'(x) \rightarrow \text{loves}'(x)(y)) : e \rightarrow \text{bool}\)
(So, this is a property of individuals, in other words, the same type as that of common nouns.) Abstracting away the meaning of /loves/ we are left with the following meaning for /every woman/:

\begin{equation}
\lambda Q. \lambda y. (\forall x) (\text{woman}'(x) \rightarrow Q(x)(y)) : \text{(e→e→bool)} \rightarrow \text{e→bool}
\end{equation}

This is a function from binary predicates to unary predicates. Finally, if we abstract away the meaning of the common noun /woman/ we get

\begin{equation}
\lambda P. \lambda Q. \lambda y. (\forall x) (P(x) \rightarrow Q(x)(y)) : \text{(e→bool)} \rightarrow \text{(e→e→bool)} \rightarrow \text{e→bool}
\end{equation}

Thus we end up with two types of quantifiers: those that quantify over subjects and those that quantify over objects.

One may think that distinguishing subject and object quantifiers is not a good idea. This analysis seems to miss the common aspect: quantification over some domain. Though one needs to realise that despite the two meaning being very similar, they are not the same. For the subject quantifier does quantify over different objects than the object quantifier. It is just that this cannot be expressed as directly as one wishes. This is indeed a problematic aspect of this interpretation, so we try another one.

Montague’s approach is highly unusual for natural language, and is inspired by predicate logic. He first introduces formal string objects. These are called pronouns, and written \(e_0, e_1,\) and so on. Their meaning is glossed here as \(x_i;\) under a valuation \(\beta\) they are interpreted as \(\beta(x_i)\). This creates strings of the form “\(e_1\) sees \(e_3\)”. The interpretation of this string is that \(\beta(x_1)\) sees \(\beta(x_3)\), which is to say \(\langle \beta(x_1), \beta(e_3) \rangle \in I(\text{see})\). (Actually, Montague himself used /he/ in place of \(e_i\), but that make no factual difference.) However, these strings are not what we see on the surface, so they need to be changed eventually. This is done using a quantifier rule. A quantifier is introduced in a somewhat unusual fashion. It abstracts that variable away. In Montague grammar this abstraction has the effect of removing all occurrences of \(e_i\) and inserting an overt expression for them. Let us see how this is done by Montague himself (the following is from [Dowty et al., 1981]).

\begin{enumerate}
\item[14.] If \(\alpha \in P_T\) and \(\varphi \in P_I\), then \(F_{10,\alpha}(\alpha, \varphi) \in P_I\) where either (i) \(\alpha\) does not have the form he\(_k\) or him\(_k\) by and \(F_{10,\alpha}(\alpha, \varphi)\) comes from \(\varphi\) by replacing the first occurrence of he\(_n\) by \(\alpha\) and all other occurrences by he/she/it or him/her/it, respectively, according as the gender of
the first $B_{CN}$ or $B_T$ in $\alpha$ is masc./fem./neut. or (ii) $\alpha$ has the form he$_k$ or him$_k$, and $F_{10,n}(\alpha, \varphi)$ comes from $\varphi$ by replacing all occurrences of he$_n$ or him$_n$ by he$_k$ or him$_k$, respectively.

So, basically, if $\alpha = \text{he}_k$, then we replace it (and all other occurrences) by he$_n$. So the operation divides into two cases; in the first case, $\alpha$ is not a pronoun. Then it replaces the first occurrences of he$_n$:

(2.15) $F_{10,2}(\text{/every man/}, \text{he}_2 \text{ loves him}_7)/\text{every man loves him}_7$

(2.16) $F_{10,3}(\text{/every man/}, \text{he}_2 \text{ loves him}_7)/\text{he}_2 \text{ man loves him}_7$

(2.17) $F_{10,7}(\text{/every man/}, \text{he}_2 \text{ loves him}_7)/\text{he}_2 \text{ man loves every man}$

The other case is shown here:

(2.18) $F_{10,2}(\text{/he}_2/), \text{he}_2 \text{ loves him}_7)/\text{he}_2 \text{ loves him}_7$

(2.19) $F_{10,2}(\text{/he}_3/), \text{he}_2 \text{ loves him}_7)/\text{he}_2 \text{ loves him}_7$

(2.20) $F_{10,7}(\text{/him}_7/), \text{he}_2 \text{ loves him}_7)/\text{he}_2 \text{ loves him}_7$

We do not need extra rules to eliminate the indices. Insertion of case is not fully worked in Montague Grammar (it is basically treated on a case-by-case basis), and will be left aside here too. Below we shall eliminate the phonetic content of the pronouns anyway.

Here $T$ is a category, the category of subjects of intransitive verbs. $t$ is the category of sentences. This rule is accompanied by the following interpretive rule. Condition (ii) is only for the case that we put in a pronoun. We shall concentrate on (i).

$T14_n$. If $\alpha \in P_T$ and $\varphi \in P_t$, and $\alpha, \varphi$ translates into $\alpha', \varphi'$, respectively, then $F_{10,k}(\alpha, \varphi)$ translates into $\alpha'(\lambda x_n, \varphi')$.

Combining these two gives us a single mode of combination (now using my notation):

(2.21) $M_{10,k}(\langle \vec{x}, m \rangle, \langle \vec{y}, n \rangle) := \langle F_{10,k}(\vec{x}, \vec{y}), m(x_n, n) \rangle$

Notice that there is a parameter $k$ here. It coordinates syntax and semantics, because both contain indices. In semantics they will remain, while in syntax they have to be eliminated.
For example, we can remove $e_1$ with the help of the quantifier rule that inserts a quantified phrase, say /every man/, and replace the first (and in this case only) occurrence of $e_1$ by it. This gives us /every man sees $e_3$/. And its interpretation is: for all $\beta' \sim_1 \beta$: $\langle \beta(x_1), \beta(x_3) \rangle \in I(\text{see})$. We can visualise this as follows. From the pair

$$
\langle /e_1 \text{ loves } e_3/, \text{love'}(e_3)(e_1) \rangle
$$

and

$$
\langle /\text{every man loves } e_3/, (\forall x_1)(\text{man'}(x_1) \rightarrow \text{love'}(x_3)(x_1)) \rangle
$$

we get

$$
\langle /\text{every man loves some woman/},
(\exists x_3)(\text{woman'}(x_3) \land (\forall x_1)(\text{man'}(x_1) \rightarrow \text{love'}(x_3)(x_1)) \rangle
$$

The motivation behind this move is that it allows to account for different quantifier scopes with ease. We can easily change the order of abstraction.

$$
\langle /e_1 \text{ loves some woman/}, (\exists x_3)(\text{woman'}(x_3) \rightarrow (\forall x_1)(\text{man'}(x_1) \rightarrow \text{love'}(x_3)(x_1)) \rangle
$$

Now the pronouns $e_i$ are real strings in Montague’s analysis, but here I take them to be silent, because I assume that there is no deletion in syntax. So, here now is my own version of Montague’s PTQ (Proper Treatment of Quantifiers).

**Example 17. Montague Style [Dowty et al., 1981].** Consider having empty signs of the form $e_i := \langle e, x_i \rangle$ in addition to $\sigma_i := \langle /\text{loves/}, \text{love'} \rangle$ and $\sigma_m := \langle /\text{man/}, \lambda x.\text{man'}(x) \rangle$ and $\sigma_w := \langle /\text{woman/}, \lambda x.\text{woman'}(y) \rangle$. $x_i$ is of type e, and nothing else is. Then the string /loves/ is manifold ambiguous. It can be all of the following: $\sigma_i$, $\sigma_i \bullet e_j$, the meanings of which are however different.

$$
\sigma_i = \langle /\text{loves/}, \text{love}' \rangle
$$

(2.24) $\sigma_i \bullet e_j = \langle /\text{loves/}, \text{love}'(x_j) \rangle$

$$
e_i \bullet (\sigma_i \bullet e_j) = \langle /\text{loves/}, \text{love}'(x_j)(x_i) \rangle
$$

The existential quantifier is interpreted as

$$
\lambda x.\lambda Q.\lambda P.(\exists x)(Q(x) \land P) : e \rightarrow (e \rightarrow \text{bool}) \rightarrow \text{bool} \rightarrow \text{bool}
$$

(2.25) This follows from two facts: the standard interpretation of a binary existential as $E = \lambda Q.\lambda P.(\exists x)(Q(x) \land P(x))$ and the fact that in Montague’s rule T14 we compute
\[ E(Q)(\lambda x. U), \] so that \( \lambda x. U = P \), whence \( P(x) = ((\lambda x)U)(x) = U. \) So the actual interpretation that is directly applied here, if you will the ‘real’ interpretation, is \( \lambda U. \lambda P. (\exists x)(Q(x) \land U). \) So we have additional signs
\[(2.26) \ \sigma^i_3 := \langle /\text{some}, \lambda Q. \lambda P. (\exists x_i)(Q(x_i) \land P(x_i)) \rangle \]

The string /\text{some man/} is then parsed as a forward application of \( \sigma^i_3 \) for some \( i \) (say, 3), and the sign \( \sigma_m. \) This gives
\[(2.27) \ \sigma^3_3 \bullet \sigma_m = \langle /\text{some man/}, (\exists x_3)(\text{man}'(x_3) \land P) \rangle \]

This is of type \text{bool} \rightarrow \text{bool}. 

The advantage of this analysis is that it keeps the unity of the existential meaning. However, it comes at a big price. One is that we need to introduce an infinite number of quantifiers. Another is that it allows for other meanings as well. Consider the following term
\[(2.28) \ \sigma^3_3 \bullet \sigma_m = ((\epsilon_2 \bullet (\sigma_l \bullet \epsilon_7)) \bullet (\sigma^9_9 \bullet \sigma_w)) \]

This unfolds to
\[(2.29) \ \langle /\text{some man loves every woman/}, \rangle \]
\[ (\exists x_3)(\forall x_0(\text{man}'(x_3) \land (\text{woman}'(x_0) \rightarrow \text{love}'(x_7, x_2)))) \]

This is equivalent to “\( x_7 \text{ loves } x_2 \)”. Clearly, this is not what is intended. The fix for this according to syntactic theory is to impose constraints on the occurrence of empty categories (for example, they have to be bound). The unfortunate problem is that the grammar cannot always detect unbound occurrences. In Montague’s original version the pronouns were not empty and had to be replaced; in this case a simple surface constraint on the occurrence of such strings will suffice. But the present formulation does away with deletion. In this case an empty pronoun is invisible at the syntax. However, there are predicate logical formulae with a free variable whose meaning does not depend on them (take, for example \( p(x_0) \lor \neg p(x_0) \)). Such formulae have a semantics that thinks no free variables occur in the formula. Syntactic theories, however, are built around the idea that the constraint on variables is not plainly semantic in this way. Consider the sentence.

\[(2.30) \ \text{Everything is either red or it is not red}. \]
If binding was purely semantic, we can choose whether to bind the variable in the VP or not. If it was syntactic, only the first choice exists. Now, the outcome is logically speaking still the same:

\[(\forall x_0)\text{red}'(x_0) \lor \neg \text{red}'(x_0)) \equiv (\forall x_0)\text{red}'(x_0) \lor \neg \text{red}'(x_0))\]

The real problem is therefore not which variables are free but rather a different one, namely, which of the free variables need to be bound. To see this, take a look at the following two sentences.

(2.32) Some man loves every woman.
(2.33) Every woman is loved by some man.

In composing the meaning of these sentences we start by feeding empty pronouns. If the index can be chosen arbitrarily, as we assume, then we can in principle assume that they contain the constituents \langle /loves/, \text{loves}'(x_2)(x_7) \rangle and \langle /is loved by/, \text{loves}'(x_2)(x_7) \rangle. (You may simply start with \(\sigma_I = \langle /\text{loves}/, \text{loves}' \rangle\) and \(\sigma_P = \langle /\text{is loved by}/, \lambda y.\lambda x.\text{loves}'(x)(y) \rangle\) and then form the constituents \((e_2 \bullet (\sigma_I \bullet e_7))\) and \((e_7 \bullet (\sigma_P \bullet e_2))\). In the next step we must choose which variable to abstract. The quantifier to the right quantifies over \(x_2\) in (2.32) and over \(x_7\) in (2.33). The problem is that we have principled way to tell which variable we should abstract over: both are free variables of the formula. Once we have discharged the \(\lambda\)-operators we have flattened the type to \text{bool}, and so we have lost the syntactic information about the argument status of the variable that was encoded in the type.

In generative grammar syntactic structure is not a string but a tree, and here again empty material can be detected.

Example 18. Heim/Kratzer Style [Heim and Kratzer, 1998]. This semantics is similar to the previous. However, here the existential quantifier /some/ has the semantics \((\lambda P)(\lambda Q)((\exists x)(P(x) \land Q(x)))\). Montague’s idea of “silent” arguments remains. But rather than binding the variable through the quantifier, a separate binding device (a \(\lambda\)-operator) is introduced. (Recall that T14 was formulated using \(\lambda\); yet, from a standpoint of the grammar, no separate abstractor has been postulated, so it cannot be used in full generality.) There are new signs

\[(2.34) \mathfrak{u}^i := \langle \epsilon, \lambda P, (\lambda x_i)P \rangle\]
2.1. Basic Clauses

When combined with a predicate \( R \) that has free occurrence of \( x_i \), these occurrences are bound after \( \lambda P.\lambda x_i. P \), is applied, and so it acts as a binder of the variable \( x_i \). Again we need to have an finite array of such binders.

Let’s go through a derivation of the same sentence above.

\[
\begin{align*}
T & := e_3 \bullet (\sigma_\ell \bullet e_7) = \langle \text{loves/}, \text{love}'(x_7)(x_3) \rangle \\
L_7 \bullet T_3 & = \langle \text{loves/}, \lambda x_3. \text{love}'(x_7)(x_3) \rangle \\
\sigma_3^2 \bullet w & = \langle \text{every man/}, \lambda P, (\exists x_2)(\text{woman}'(x_2) \land P(x_2)) \rangle \\
(2.35) \quad u & := (\sigma_3^2 \bullet w) \bullet (L_7 \bullet T) = \langle \text{loves some woman/}, \quad \\
& \quad (\exists x_2)(\text{woman}'(x_2) \land \text{love}'(x_2)(x_3)) \rangle \\
(\sigma_\gamma^1 \bullet m) \bullet (L_3 \bullet u) & = \langle \text{every man loves some woman/}, \\
& \quad (\forall x_1)(\text{man}'(x_1) \rightarrow (\exists x_2)(\text{woman}'(x_2) \\
& \quad \land \text{love}'(x_2)(x_3)) \rangle
\end{align*}
\]

**Example 19.** *Ploucquet Style* [Lenzen, 2008]. Ploucquet reads the quantifiers not as quantifiers over individuals but as quantifiers over sets of individuals. In his reading of the syllogistic he treats “all \( P \)” as the set \( P \) itself, while “some \( P \)” is “some nonempty subset of \( P \)”. The copula he reads as “is the same as”. Negation is read as “is disjoint from”. This gives eight sentence forms.

1. Some \( A \) is some \( P \).
2. Some \( A \) is all \( P \).
3. All \( A \) are some \( P \).
4. All \( A \) are all \( P \).
5. Some \( A \) is not some \( P \).
6. Some \( A \) is not all \( P \).
7. No \( A \) are some \( P \).
8. No \( A \) are all \( P \).
I give a Montague style interpretation, using sets of sets of objects in place of functions over properties of individuals. For the quantifiers we take the following unary signs:

\[(2.36)\]
\[
\langle/\text{some}, \lambda P.\{Q : \emptyset \subseteq Q \subseteq P\}\rangle
\]
\[
\langle/\text{all}, \lambda P.\{P\}\rangle
\]

For the copula there are four signs:

\[(2.37)\]
\[
\langle/\text{is}, \lambda P.\lambda Q.\exists P \in Q.\exists Q \in P. P = Q\rangle
\]
\[
\langle/\text{are}, \lambda P.\lambda Q.\exists P \in Q.\exists Q \in P. P = Q\rangle
\]
\[
\langle/\text{is not}, \lambda P.\lambda Q.\exists P \in Q.\exists Q \in P. P \cap Q = \emptyset\rangle
\]
\[
\langle/\text{are not}, \lambda P.\lambda Q.\exists P \in Q.\exists Q \in P. P \cap Q = \emptyset\rangle
\]

The script letters denote variables over sets of sets of objects, while the ordinary capital letters denote just sets of objects. The choice between /is/ and /are/ and between /is not/ and /are not/ is controlled by the choice of the subject term, and a morphological matter that we shall ignore here. The interesting part here is that the negated sentence is still interpreted existentially.

The Ploucquet calculus is interesting. It offers subtleties that the standard syllogistic logic does not have. It can express, for example, that two concepts are the same: /All \(P\) are all \(Q\)/. It is not clear, however, whether this is at all in line with standard intuitions.

### 2.2 Starting with Predicate Logic

Montague built the type universe over a standard model of predicate logic, so this is how we start. Predicate logic is actually not a single language but a language schema. It has a component that is universal (the connectives, the quantifiers, and the variables) and a component that can be added as one sees fit (function and relation symbols). The latter symbols are also called nonlogical symbols. So, we have two sorts of nonlogical symbols; relation symbols and function symbols. The is defined by naming (a) the sets \(F\) and \(R\) (must be disjoint), and a function \(\Omega : F \cup R \rightarrow \mathbb{N}\). So this is much like the signature, except that we distinguish two kinds of primitive symbols. Thus, the following are our basic symbols:
2.2. Starting with Predicate Logic

1. function symbols; relation symbols;
2. variables \( x_i, i \in \mathbb{N} \);
3. quantifiers \( \exists, \forall \);
4. boolean connectives \( \neg, \land, \lor, \rightarrow \).

Given a signature, we define the set of terms as follows. An expression is a term if it either a variable or of the form \( f(s_0, \ldots, s_{\Omega(f)-1}) \) where \( f \in F \) is a function symbol. An expression is a formula if it has one of the following forms.

1. \( r(s_0, \ldots, s_{\Omega(r)-1}) \), where \( r \in R \) and \( s_i \) is a term for \( i < \Omega(r) \);
2. \( \neg \chi, \varphi \land \chi, \varphi \lor \chi, \varphi \rightarrow \chi \) where \( \varphi \) and \( \chi \) are formulae;
3. \( (\forall x_i) \varphi, (\exists x_i) \varphi \) where \( x_i \) is a variable and \( \varphi \) a formula.

If we want to be as precise as in the previous chapter we must settle on a concrete syntax (alphabet of letters) and then introduce the signs and their functions on them. I shall fix a particular surface variant.

1. Variables have the form \( x^\tilde{y} \), where \( \tilde{y} \) is a sequence of digits;
2. Function and relation symbols are strings of lower case alphabetic letters, and the punctuation marks underscore and hyphen; the string \( x \) is however excluded (it is a variable);
3. Predicates are placed before their arguments, and no brackets are being used; instead arguments are separated by a space (/see \( x0 \ x1 \)/).
4. Function symbols are placed before their arguments, with no brackets being used, but a blank is placed in between items (/successor \( x1 \)/).
5. Constants, being special functions or relations, are used without any subsequent arguments (by the conventions issued above);
6. \( \neg \) is prefixed, and no brackets are added (/\( \neg \) see \( x0 \ x1 \)/).
7. \( \land, \lor, \rightarrow \) are infixed, but brackets are obligatory (/\( \neg \) see \( x0 \ x1 \land \text{neighbour} \ x1 \ x0 \)/).
Quantifiers are enclosed in brackets together with their variable, and simply placed before their argument; a space separates the quantifier from its variable \((\exists x_0)(\neg \text{see } x_0 x_1 \land \text{neighbour } x_1 x_0)\).

Most of the spaces are unnecessary; some of the brackets can also be dropped (can you see where I was too cautious?). But life is then easier in the implementation. Brackets need not be separated from alphabetic letters by a space, but again it is easier if they do.

I now proceed to the semantics. For the case where \(\Omega(r) = n\), \(r\) a relation symbol, we say that \(I(r) \subseteq D^n\), and in the case where \(\Omega(f) = n\), \(f\) a function symbol we put \(I(f) : D^n \to D\). The pair \(\mathcal{D} := \langle D, I \rangle\) is called a model structure. It interprets the basic symbols. A valuation is a function \(\beta : V \to D\), where \(V = \{x_i : i \in \mathbb{N}\}\) is our set of variables.

Given the pair \(\langle D, \beta \rangle\) we define the truth of a complex formula as follows. First, we define terms and their values under \(\beta\).

1. A variable is a term, and \(x_i^\beta := \beta(x_i)\).
2. If \(f\) is a function symbol, then \(f(s_0, \cdots, s_{\Omega(f)-1})\) is a term and
   \[
   f(s_0, \cdots, s_{\Omega(f)-1})^\beta := I(f)(s_0^\beta, \cdots, s_{\Omega(f)-1}^\beta)
   \]

For the purpose of the next definition, write \(\beta \sim_i \gamma\) if for all \(j \neq i\): \(\beta(x_j) = \gamma(x_j)\).

\[
\begin{align*}
\langle D, \beta \rangle \models r(s_0, \cdots, s_{\Omega(r)-1}) & : \Leftrightarrow \langle s_0^\beta, \cdots, s_{\Omega(r)-1}^\beta \rangle \in I(r) \\
\langle D, \beta \rangle \models \neg \chi & : \Leftrightarrow \langle D, \beta \rangle \not\models \chi \\
\langle D, \beta \rangle \models \chi \land \varphi & : \Leftrightarrow \langle D, \beta \rangle \models \chi \land \langle D, \beta \rangle \models \varphi \\
\langle D, \beta \rangle \models \chi \lor \varphi & : \Leftrightarrow \langle D, \beta \rangle \models \chi \lor \langle D, \beta \rangle \models \varphi \\
\langle D, \beta \rangle \models \chi \rightarrow \varphi & : \Leftrightarrow \text{if } \langle D, \beta \rangle \models \chi \text{ then } \langle D, \beta \rangle \models \varphi \\
\langle D, \beta \rangle \models (\exists x_i)\varphi & : \Leftrightarrow \text{there is } \beta' \sim_i \beta : \langle D, \beta' \rangle \models \varphi
\end{align*}
\]

(2.38)

For the moment we fix this language just for reference. I shall return to the problem of implementing it.

First step is the definition of language and models. As I earlier said, objects are made of strings using a type constructor. There is a new type of sets of objects.
This is produced via the module `Dom`. The rest of the program is type definitions, accompanied by a single function which checks for a model and an input signature whether the model is appropriate for the signature. There are relational and function symbols.

```ocaml
let is_homogeneous lst dom num = List.for_all (fun x -> ((List.length x = num) && (List.for_all (fun y -> Dom.mem y dom) x))) lst

let appropriate m s = match m, s with Md (d, ip), Lg (fsymb, omega) -> List.for_all (fun x -> match x with Rel r -> is_homogeneous (ip r) d (omega r) fsymb | Fct f -> is_homogeneous (ip f) d ((omega f)+1)) fsymb
```

This is somewhat crude. A model has as its first component a set, and the second is the interpretation. A relation is interpreted by a list of lists. Similarly a function. However, there is no safeguard against improper definitions (like not having a value for an input or having too many). This needs to be implemented. It would be better to convert the interpretation of a function into an object of type `e list`
The data is still of the form `e list list`; a function is used to covert that into the first format while checking for consistency. In the interest of space, I shall now drop functions from consideration here and concentrate on relations.

```plaintext
(2.40)
type ind = I of string
type vals = ind -> e
let change v i y = (fun j -> if j = i then y else (v j))
let ext d i v = List.map (change v i) (Dom.elements d)
```

The task of interpreting the language is done by converting formulae into objects of a certain recursive type. The interpretation of the objects of this type is done by induction on the type. Here is how.

```plaintext
(2.41)
type var = V of ind
type term = Bs of var | Cm of term list

type fml = B of symb * term | Cj of fml * fml
            Dj of fml * fml | Im of fml * fml | N of fml
            Ex of var * fml | Un of var * fml
```

Notice that for every logical symbol we have introduced a type constructor.

Now we are ready to issue the core procedure for defining truth in a model the following is a function from models, valuations, and formulae to `bool`. Recall
that we have no function symbols, so terms are variables.

\[
\text{let rec true_in m v f =}
\begin{align*}
\text{match m with Md (d, ip) ->} \\
\text{match f with} \\
\text{B (r, t) -> List.mem (List.map v t) (ip r)} \\
\text{| N x -> not (true_in m v x)} \\
\text{| Cj (x,y) -> (true_in m v x) && (true_in m v y)} \\
\text{| Dj (x,y) -> (true_in m v x) || (true_in m v y)} \\
\text{| Im (x,y) -> if (true_in m v x)} \\
\text{  then (true_in m v y)} \\
\text{| Ex (i,x) -> List.exists (true_in m v)} \\
\text{  (ext d i v)} \\
\text{| Un (i,x) -> List.for_all (true_in m v)} \\
\text{  (ext d i v)}
\end{align*}
\]

(2.42)

This defines the mapping from a formula into a truth value. Finally we need to look into the translation from a string into a formula. This will use as input also the signature. We can use a stream object here, since the language is not only uniquely readable but can also be parsed deterministically from left to right. The details are left as an assignment. We proceed basically as in the propositional case. The basic elements are no longer propositions but consists of strings of the form \( r_{i_0} \cdots r_{i_{n-1}} \), where \( r \) is a string for a relation symbol, \( n = \Omega(r) \), and \( i_j, j < n \), have the form \( x^{r_{i_j}} \). Thus whenever we have identified a relation symbol we scan for the right amount of variables. If there are less variables or more the input is rejected.

## 2.3 Implementation: Basic Issues

Now that we have implemented the semantics of predicate logic we shall move to natural language. In this section I shall deal with the question of implementing the various semantics of Section 2.1. First I will review some basic techniques before I start with the actual implementations in the next section.

First, we need to discuss the issue parsing. A simple tool to parse context free
grammars is a **chart parser**. Let \( G \) be a context free grammar in 2-standard form, that is, assume that rules have the form \( A \rightarrow BC \), with \( A, B \) and \( C \) nonterminal, of \( A \rightarrow \vec{x} \) with \( A \) nonterminal and \( \vec{x} \) a terminal string.

**Example 20.** A particular grammar of this form is **categorial grammar**. Here, nonterminals have the form \( \alpha/\beta \) or \( \alpha \setminus \beta \), where \( \alpha \) is either a nonterminal or a basic category. The set \( C \) of basic categories is arbitrary. We write \( \text{Cat}(C) \) for the set of categories over \( C \). There are two rule schemata:

\[
\begin{align*}
(2.43) & \quad \alpha \rightarrow \alpha/\beta \quad \beta \\
(2.44) & \quad \alpha \rightarrow \beta \quad \beta \setminus \alpha
\end{align*}
\]

Furthermore, we have terminal rules of the form

\[
(2.45) \quad \alpha \rightarrow \vec{x}
\]

where \( \vec{x} \) is a string. There is a tight connection between the categories and the types, which runs as follows. Each category \( \gamma \) is assigned a unique type \( s(\gamma) \). Let \( C \) be the set of basic categories. Then the map \( s : \text{Cat}(C) \rightarrow \text{Typ}(B) \) satisfies

\[
(2.46) \quad s(\gamma/\delta) = s(\delta \setminus \gamma) = s(\delta) \rightarrow s(\gamma)
\]

For example, in Montague Grammar, nouns and intransitive verbs all have the type \( e \rightarrow \text{bool} \). However, they are syntactically distinct. So we say that there are two basic categories, call them \( n \) and \( t \), of names and sentences, respectively. We stipulate that \( s(t) := \text{bool} \) and \( s(n) := e \rightarrow \text{bool} \). The category of verbs is \( n \setminus t \). This takes care of the fact that /John runs/ is grammatical while /John man/ is not, despite the fact that the types allow the application of \( \text{man}' \) to \( \text{john}' \). Table 2.1 shows a category assignment for primitive words. Notice that for every category we have to postulate a different category for /and/. Here is an example of a string that the grammar generates:

\[
\begin{align*}
\text{every} & \quad \text{dog} & \quad \text{runs} & \quad \text{and} & \quad \text{sees} & \quad \text{a} & \quad \text{cat} \\
(t/(n/t))/n & \quad n & \quad n/t & \quad ((n/t)/(n/t)) & \quad (n/t) & \quad ((n/t)/(n/t)) & \quad n \\
(2.47) & \quad t/(n/t) & \quad : & \quad : & \quad : & \quad : & \quad ((n/t)/(n/t)/(n/t)) & \quad n/t \\
& \quad : & \quad : & \quad : & \quad n/t & \quad : & \quad (n/t)/(n/t) \\
& \quad : & \quad : & \quad : & \quad : & \quad n/t \\
& \quad : & \quad : & \quad : & \quad : & \quad t
\end{align*}
\]
Table 2.1: Categories for basic words

<table>
<thead>
<tr>
<th>Word</th>
<th>Category</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
<td>( t )</td>
<td>bool</td>
</tr>
<tr>
<td>blue</td>
<td>( n )</td>
<td>e ( \rightarrow ) bool</td>
</tr>
<tr>
<td>runs</td>
<td>( n/n )</td>
<td>(e ( \rightarrow ) bool) ( \rightarrow ) e ( \rightarrow ) bool</td>
</tr>
<tr>
<td>sees</td>
<td>( n/t )</td>
<td>e ( \rightarrow ) bool</td>
</tr>
<tr>
<td></td>
<td>( (n/t)/n )</td>
<td>e ( \rightarrow ) bool</td>
</tr>
<tr>
<td>a</td>
<td>( t/(n/t) )</td>
<td>(e ( \rightarrow ) bool) ( \rightarrow ) (e ( \rightarrow ) bool) ( \rightarrow ) bool</td>
</tr>
<tr>
<td>a</td>
<td>( ((n/t)/(n/t))/n )</td>
<td>(e ( \rightarrow ) bool) ( \rightarrow ) (e ( \rightarrow ) e ( \rightarrow ) bool) ( \rightarrow ) e ( \rightarrow ) bool</td>
</tr>
<tr>
<td>and</td>
<td>( (t/t)/t )</td>
<td>bool ( \rightarrow ) bool ( \rightarrow ) bool</td>
</tr>
<tr>
<td>and</td>
<td>( (n/n)/n )</td>
<td>(e ( \rightarrow ) bool) ( \rightarrow ) (e ( \rightarrow ) bool) ( \rightarrow ) bool</td>
</tr>
<tr>
<td>and</td>
<td>( ((n/t)/(n/t))/(n/t) )</td>
<td>(e ( \rightarrow ) bool) ( \rightarrow ) (e ( \rightarrow ) e ( \rightarrow ) bool) ( \rightarrow ) e ( \rightarrow ) bool</td>
</tr>
<tr>
<td>and</td>
<td>( ((n/t)/(n/t))/(n/t)/n )</td>
<td>(e ( \rightarrow ) e ( \rightarrow ) bool) ( \rightarrow ) e ( \rightarrow ) bool</td>
</tr>
</tbody>
</table>

It may be checked that the type assignments work out appropriately: every time a constituent is formed, it is either the right hand side or the left hand side that becomes a function taking the other side as argument.

For simplicity we assume that words are separated by a blank. Given a string \( \vec{x} \), we shall convert \( \vec{x} \) into a list of words. Thus, from the input string

\[(2.48) \quad "\text{every dog runs and sees a cat}" \]

we produce the list

\[(2.49) \quad ["\text{every}"; "\text{dog}"; "\text{runs}"; "\text{and}"; "\text{sees}"; "\text{a}"; "\text{cat}"] \]

In OCaml, the following one-line command define a function \( \text{chop} \) that does the job:

\[(2.50) \quad \text{let chop s = Str.split (Str.regexp "[ ]+") s} \]

Let \( \ell \) be this list, and \( n \) its length. A chart is a map \( c \) from pairs \([i, j]\) where \( 0 \leq i < j \leq n \) to sets of syntactic objects (here categories). The interpretation is that \( c([i, j]) \) is the set of all categories \( A \) such that the concatenation of \( \ell(i), \ell(i+1), \ldots, \ell(j-1) \) is of category \( A \) in the language of the grammar \( G \). So, \( c([i, i+1]) \) is the set of all \( A \) such that \( A \rightarrow \ell(i) \) is a rule of the grammar. This is how we start filling the chart. We start with pairs \([i, j]\) such that \( j - i = 1 \). Now, suppose that
we have computed all $c([i, j])$ where $j - i \leq k$. Now we shall compute $c([i, i + k])$ in the following way. $A \in c([i, i + k])$ iff there is $j$ such that $0 < j < k$, $B \in c([i, j])$ and $C \in c([j, k])$ such that $A \rightarrow BC$ is a rule of the grammar.

So we fill the chart by successively adding the values for pairs $[i, i + d]$, where $d$ goes from 1 to $n$. The chart is filled when we have computed $c([0, n])$. If this set contains the start symbol, we have a sentence.

I shall describe a chart parser implementation using categorial grammar. The first step is to implement the categories. Basic categories are made of type string, and there are two ways to construct complex categories, using forward slash and backslash. Thus we have three type constructors, Bas "a" creates the category "a", Fw $(x, y)$ makes $x/y$, and Bw $(x, y)$ makes $x\backslash y$.

```
exception Error of string

type cat = Bas of string | Fw of cat * cat
 | Bw of cat * cat

let combine a b = match a, b with
 | Fw (c,x), y when x = y -> c
 | x, Bw (y,c) when x = y -> c
 | _ -> raise (Error "Operation undefined")
```

A convenient structure for a chart is an array. We make the chart as

```
let chart = Array.make_matrix (n+1) (n+1)
```

Here $n$ is set to the length of the list. The next step is the initialisation. Here we must assume that we know something about the basic words of our language.

(...)

Before we can do that, however, we must speak a little about the various type shifts that occur in the formulation of these semantics. Montague for example uses only functional types. Thus, instead of interpreting a unary relation as a set, he interprets it as a function from objects to truth values. The rationale for this is that the two objects are just variants of each other.

**Example 21.** **Sets as Functions.** Let $X$ be given. A function $f : X \rightarrow \{0, 1\}$ can be coded as the set $f^* := f^{-1}(1)$. Conversely, let $Y \subseteq X$. Then put $Y_\bullet(x) = 1$ iff
\(x \in Y\) (and 0 otherwise). Then for every function \(f\), \((f^*)_{\wedge} = f\) and for every set \(Y \subseteq X\), \((Y^*)_\wedge = Y\). For take \(x \in X\). Then \((f^*)_{\wedge}(x) = 1\) iff \(x \in f^* = f^{-1}(1)\) iff \(f(x) = 1\). So the two are the same function. Also, \(x \in (Y^*)_{\wedge}\) iff \(Y^*(x) = 1\) iff \(x \in Y\). Thus, the two are the same set.

Notice that sets of objects are of some abstract type while functions have the type \(e \rightarrow \text{bool}\). In OCaml it is thus considerably easier to use functions to \text{bool} than to use sets. For our purposes I shall use the constructor \((-)^\circ\) to create the type of sets over something. Thus \(e^\circ\) will denote the type of sets of objects.

A binary relation is coded as a set of pairs of objects. We can use the OCaml pair function to create pairs of objects. Again we have a type constructor \((-)^\ast(-)\) that creates the type of pairs of objects. So, a relation is interpreted as an object of type \((e^\ast e)^\circ\). Using the previous method we can now code the relation not as a set of pairs but rather as \(e^\ast e + \text{bool}\). We apply another step to convert this.

**Example 22.** Currying. Let \(f : A \times B \rightarrow C\) be a function. Define \(C^B\) to be the set of functions from \(B\) to \(C\). Define \(f^\circ : A \rightarrow C^B\) as follows. \(f^\circ(x) : y \mapsto f(x,y)\). So, \(f^\circ(x)\) is a function, and \(f^\circ(x)(y) = f(x,y)\) by definition. Conversely, let us be given a function \(g : A \rightarrow C^B\). Then define \(g^\circ : A \times B \rightarrow C\) by \(g^\circ(x,y) = g(x)(y)\). Again one can show that \((f^\circ)_\circ = f\) and \((g^\circ)_\circ = g\). So this recodes a function on two arguments into one of a single argument.

Applying this to our binary relation we finally get an object of type \(e + e + \text{bool}\), the one that Montague uses. Currying is actually automatic in OCaml. If you define a function by \(\text{let } f x y\) then not only is the object \(f\ a\ b\) defined (provided the arguments are), but also \(f\ a\) and \(f\). So, the interpretation of \(f\) in OCaml is not a function from some pairs to results but is its Curried equivalent.

There is a third type of conversion that I briefly like to mention. The pair operator allows to combine only two objects, but sometimes we have to group more objects together. We can do this by iterating the pair construction. Also, notice that for sets, \(A \times (B \times C)\) can be mapped bijectively onto \((A \times B) \times C\) via \((x, (y, z)) \mapsto ((x, y), z)\). From a type theoretic view this corresponds to converting \('a^\ast('b^\ast 'c)\) to \('(a^\ast 'b)^\ast 'c)\.

I shall now start by exploring the first semantics, from Example [16]. I shall show how to build a parser for such a semantics. For simplicity I shall assume that the units are words. The first step therefore consists in splitting the input
string into words. (If this is a real sentence we need to eliminate the period and deal with the first letter, which we may or may not need to render lower case.)

The sentence

\[(2.53) \text{ Every cat sees a mouse.} \]

then becomes the list

\[(2.54) ["every"; "cat"; "sees"; "a"; "mouse"] \]

Using this list we shall create a chart. The cells of the chart (= array) are filled by sets of pairs consisting of a nonterminal category and a meaning each. At this point we realise that there is a problem with this approach as far as OCaml is concerned. The type of the meaning objects are quite different, so we cannot put them together into the chart. One solution to this is as follows. In a purely applicative categorial grammar we generate only finite many types, so we can effectively enumerate them. We the create a new type \text{global}, which will have as its subtypes all the one that we need for this grammar. Let us see how this works. The following is a list of representative lexical entries together with the type that their associated meaning has.

\[(2.55) \]

\[
\begin{align*}
/cat/ & \text{ e-bool} \\
/sees/ & \text{ e-bool} \\
/some/ & \text{ (e-bool)\rightarrow(e\rightarrow\text{bool})\rightarrow\text{bool}} \\
/some/ & \text{ (e-bool)\rightarrow(e\rightarrow\text{bool})\rightarrow\text{bool}} \\
\end{align*}
\]

These are just the preterminal objects, we need to add some more entries to this list in order to get a complete overview of the types that we shall derive.

\[(2.56) \]

\[
\begin{align*}
/some \ cat/ & \text{ (e-bool)\rightarrow\text{bool}} \\
/some \ cat/ & \text{ (e\rightarrow\text{bool})\rightarrow\text{e-bool}} \\
/sees \ some \ cat/ & \text{ e-bool} \\
/some \ cat \ sees \ some \ cat/ & \text{ bool} \\
\end{align*}
\]

So, we issue the following:

\[(2.57) \]

\[
\text{type global} = \text{ N of e-bool | TV of e\rightarrow\text{bool}} \\
| \text{ IV of e-bool | S of bool} \\
| \text{ SQ of (e-bool)\rightarrow(e\rightarrow\text{bool})\rightarrow\text{bool}} \\
| \text{ OQ of (e\rightarrow\text{bool})\rightarrow(e\rightarrow\text{bool})\rightarrow\text{e-bool}} \\
| \text{ SDP of (e-bool)\rightarrow\text{bool} | VP of e-bool} \\
| \text{ ODP of (e\rightarrow\text{bool})\rightarrow\text{e-bool}} \\
\]
The names of the type constructors are arbitrary. In the present setting, they can be even be used to store the category (you can use different type constructors with the same type).

Another solution is to create a single universal type for the entire functional hierarchy. Suppose our basic types are `bool` and `e`. Then we issue

\[
\text{type } \text{univ} = \text{B of bool} \mid \text{E of e} \\
\mid \text{F of (univ→univ)}
\]

This allows to cram all types into a single type. Beware however that in order to do this we first need to convert our functions into the type `univ`. So, if we have a function `f` of type `e→bool` we convert it as follows.

\[
\text{let e_to_bool } f = \text{fun } x \to \text{match } x \text{ with (E u)} \\
\quad \text{-> B } (f u)
\]

This conversion must still be done individually for the functions, as the input types are still distinct. However, here we only need to do the work for the lexical entries. The intermediate types generated during parsing need not be treated. They are already of type `univ`.

**Example 23.** We may use our models of predicate logic here. Suppose that `/see/` is a relation symbol of arity 2. Then its interpretation is a list of lists of length 2 over the domain. We can convert its interpretation into a function as follows:

\[
\text{let convert } \text{lst } x \text{ y } = \text{List.mem } [y; x] \text{ lst}
\]

Now, `convert lst` is a function of type `e→e→bool`, as required. Notice that the order in which the arguments are aligned is crucial. We write `/see john mary/`, when John sees Mary, but `/Mary/` must be the first argument because it is the object.

For intransitive verbs a separate conversion clause is needed, since they need one argument.

The chart is now a doubly indexed array over objects of type `global`. The chart is filled by applying either of two rules: forward application or backward application. We can stipulate the direction. So allow to combine the object “SQ
x” from positions [i, j] and “$N \ y$” from positions [j, k] to obtain “SDP x (y)” at [i, k] but not if they are ordered conversely.

Now we shall turn to Montague style semantics. This is distinct from the variable free semantics in that it makes heavy use of expressions with free variables. And so, while we so far only needed the structure to interpret expressions, now we need the model, that is, the pair $\langle D, \beta \rangle$. This was the same in propositional logic, where propositions are evaluated using a valuation. However, one difference is that the clauses for quantifiers quantify externally over valuations. So, $\langle D, \beta \rangle \models love'(x_2)(x_7)$ if $\langle \beta(x_7), \beta(x_2) \rangle \in I(love)$. Montague however convert everything to a functional type. So, while the verb /see/ has a counterpart $\text{see}'$ in predicate logic (of type $e*e\rightarrow\text{bool}$), Montague chose the type $e\rightarrow e\rightarrow\text{bool}$ (with semantics $\lambda x.\lambda y.\text{see}'(y, x)$ for /see/).

In order to make this work we need to take on board the valuations. The implicit dependency of the truth values must be made explicit, since the dependency is exploited in the definition of truth of a quantified expression. I shall propose below an approach that interprets every formula as a function from valuations to objects of some type. This allows to account for the dependency on the valuation.

Recall the type $\text{ind}$ for indices. Valuations are of type, say, $\text{ind}\rightarrow e$. The variable $x_i$ shall denote a function $x_i^*$ from valuations to objects as follows.

$$x_i^*(\beta) := \beta(x_i)$$

The formula “$\text{love}'(x_2)(x_7)$” shall be interpreted as of type $(\text{ind}\rightarrow e)\rightarrow\text{bool}$:

$$\lambda \beta. \text{love}'(x_2^*(\beta))(x_7^*(\beta))$$

In general, when we apply an object $f$ of type $(\text{ind}\rightarrow e)\rightarrow \eta\rightarrow \theta$ to an object $g$ of type $(\text{ind}\rightarrow e)\rightarrow \eta$, the result is going to be

$$\lambda \beta. f(\beta)(g(\beta))$$

which is of type $(\text{ind}\rightarrow e\rightarrow\theta)$. This pretty much takes care of function application.

The only thing we need to spell out after that is the interpretation of the quantifiers. For quantifiers are not interpreted keeping the valuation constant. They quantify over valuations. They become constants of type $(\text{ind}\rightarrow e\rightarrow\text{bool})\rightarrow(\text{ind}\rightarrow e)\rightarrow\text{bool}$, in other words, from a valuation dependent truth value to a valuation dependent truth value.

$$E_i(\varphi)(\beta) := \begin{cases} 1 & \text{iff for some } \gamma \sim_i \beta: \varphi(\gamma) = 1 \\ 0 & \text{else} \end{cases}$$
The operators are implemented as follows. We use only a finite set $S$ of indices (in the present circumstances, two indices are enough). It is possible to do away with the restriction to finitely many indices, but I leave that as an exercise. We create the set $V$ of all valuations; again, this is a finite set. Then we define the relations $\sim_i$. We define a three place relation $\text{sim}$ of type $\text{ind}\rightarrow(\text{ind}\rightarrow\text{e})\rightarrow(\text{ind}\rightarrow\text{e})\rightarrow\text{bool}$, which is true of $\beta$ and $\gamma$ and $i$ if $\beta(j) = \gamma(j)$ for all $j \neq i$. Once these primitives are implemented we can create a Montague style interpretation.

The Heim/Kratzer style semantics is yet again different. For this semantics we also need to implement an abstraction operator. At first blush this seems to be straightforward since OCaml is built around the $\lambda$-calculus. But that is an illusion. For OCaml is actually based on combinatory logic, which has no variables. There is no abstraction over a specific variable. To see this, assume that a constant “fct” of type $\beta$ has been defined using some object of type $\alpha$. How can one abstract over its argument of type $\alpha$? The answer is that this is impossible unless “fct” is equivalent to an already defined expression of the form “fct1 a” with “a” of type $\beta$.

Indeed, the only way is to proceed as above and define an abstractor of type $(\text{ind}\rightarrow\text{e})\rightarrow\text{e}\rightarrow\text{a}$. Let $\beta[x_i \mapsto a]$ be the valuation defined by (1) $\beta[x_i \mapsto a] \sim_i \beta$, and (2) $\beta[x_i \mapsto a](x_i) = a$.

\begin{equation}
L_i(\varphi)(\beta)(a) := \varphi(\beta[x_i \mapsto a])
\end{equation}

So the implementation will start by defining an operation, say, “reset”, which takes a valuation $\beta$, an index $i$ and an object $x$ and returns the valuation $\beta[x_i \mapsto a]$. Then it defines the abstractor /\lambda/, which takes an index $i$, some object $\varphi$, a valuation $\beta$, and an object $a$ and returns $\varphi(\beta[x_i \mapsto a])$. The latter is of type “a”. So the whole is of type

\begin{equation}
\text{ind}\rightarrow((\text{ind}\rightarrow\text{e})\rightarrow\text{e}\rightarrow\text{a})
\end{equation}

So the expression /\lambda/ is more general than $L_i$ in \ref{2.65} since we have a type of indices of which we can take advantage.
2.4 Implementation: Some Fragments

With the type and category system installed we can actually define complex types in an easy way.

\[
\begin{align*}
\text{let } & \text{iv }= \text{Bw } (\text{Bas } "n", \text{Bas } "t") \\
\text{let } & \text{tv }= \text{Fw } (\text{iv}, \text{Bas } "n") \\
\text{(2.67) let } & \text{sdet }= \text{Fw } (\text{Fw } (\text{Bas } "t", \text{iv}), \text{Bas } "n") \\
\text{let } & \text{odet }= \text{Fw } (\text{Bw } (\text{tv}, \text{iv}), \text{Bas } "n") \\
\text{let } & \text{conj } x = \text{Fw } (\text{Bw } (x, x), x)
\end{align*}
\]

Notice in particular the last line: it defines a type constructor which for any input category \( x \) produces the conjunctive category for \( x \) ordinarily written \( (x \setminus x)/x \).

If we only did syntax then a grammar could look like this.

\[
\begin{align*}
\text{let } & \text{gram }= \\
& \begin{cases}
\text{["dog", Bas "n"), ("a", sdet); ("a", odet);}
\text{("cat", Bas "n"), ("sees", tv);}
\text{("runs", iv); ("and", conj (Bas "t")));}
\text{("and", conj iv); ("and", conj tv);}
\text{("and", conj (Bas "n")])}
\end{cases}
\end{align*}
\]

Instead, let us now look at grammars that additionally also include semantics. Here is a tiny example.

\[
\begin{align*}
\text{(2.68) let } & \text{gram }= \\
& \begin{cases}
\text{(["fido", ((Bas "e"), Ob (O "f")))};
\text{("bella", ((Bas "e"), Ob (O "b")));}
\text{("barks", ((Bw (Bas "e", Bas "t")), (Fc (fun x ->}
\text{match x with Ob (O "f" -> Bl true | _ -> Bl false)))])}
\end{cases}
\end{align*}
\]

The grammar is a list of pairs \((x,y)\) where \( y \) is a pair consisting of a category and a meaning. No attempt is made to synchronise the category assignments and meaning assignment. This is the task of the grammar writer. The grammar accepts /fido barks/ as true and /bella barks/ as false. The initialisation is done by transforming the string into a list of words and then taking the grammar to look
up all the possible category meaning pairs that match the given words. After that the grammar can be discarded.

```
let initialize_chart g x =
  let l = Str.split (Str.regexp "\[\ ]+\") x in
  let n = List.length l in
  let c = Array.make_matrix (n+1) (n+1) [] in
  for i = 0 to n-1 do
    c.(i).(i+1) <-
      (let p = (List.filter (fun x ->
                    ((fst x) = (List.nth l i))) g)
          in List.map snd p)
    done;
  c
```

Now the chart is filled inductively. For that purpose, we need to define the product of two pairs \((\alpha, m), (\beta, n)\). There are two possibilities. (i) \(\alpha = \gamma/\beta\) in which case \(m\) is the function applied to \(n\), and so the result is \((\gamma, m(n))\), or (ii) \(\beta = \alpha\backslash\gamma\), in which
case $n$ is the function applied to $m$ and the result is ($\gamma, n(m)$).

\[
\begin{aligned}
\text{let cat_product } p q &= \\
&\text{match } p, q \text{ with } (\text{Fw } (c, x), f), (y, g) \\
&\text{when } x = y \to \\
&\quad (c, \text{apply } f \ g ) \\
&\mid (x, f), (\text{Bw } (y, c), g) \text{ when } x = y \to \\
&\quad (c, \text{apply } g \ f) \\
&\mid \_ \to \text{raise (Error "operation undefined" )}
\end{aligned}
\]

(2.71) \[
\begin{aligned}
\text{let add_to } l1 l2 l3 &= \\
&\text{let res = ref } l1 \text{ in } \\
&\text{List.iter } (\text{fun } x \to (\text{List.iter } (\text{fun } y \to \\
&\quad \text{try } (\text{let } c = \text{cat_product } x \ y \text{ in } \\
&\quad \text{if List.mem } c \ !\text{res} \text{ then () else } \\
&\quad \text{res := } c :: !\text{res}) \\
&\quad \_ \to () ) \\
&\quad 13)) \ 12; \\
&!\text{res}
\end{aligned}
\]

These definition of cat_product is as discussed above. The function add_to basically computes the product of the second and the third argument and appends that to the first. However, it also checks for multiple occurrences. The final part is
shown in Table 2.2

```
let fill c =
  let n = Array.length c in
  for d = 2 to (n-1) do
    for i = 0 to (n-1-d) do
      for s = 1 to (d-1) do
        c.(i).(i+d) <- add_to c.(i).(i+d)
        c.(i).(i+s) c.(i+s).(i+d)
      done;
    done;
  done;
  c
```

The working of this parser is as follows.

(2.73) parse gram "fido barks"

This receives the answer that this sentences has only true readings (there is only one and it is true). If you want to see the parse type show () and it returns the matrix.

2.5 Temporal Predicate Logic

We shall now look at models for temporal predicate logic. We shall use the simplest kind of models, so-called models with constant domain. In these models, there is a single domain of individuals. Predicate symbols are interpreted as time dependent relations. In the most conservative setting, the predicate logical language is not changed at all, nor is our notion of signature. We just expand the model structure. So let us be given a signature \( \Omega : F \rightarrow \mathbb{N} \). Then a model for the temporal language \( L_\Omega \) is a quadruple \( \langle D, T, \prec, I \rangle \), where \( D \) is a set (the domain of individuals), \( T \) is a set (the set of time points), \( \prec \subseteq T \times T \) the order on the time points, and \( I : F \times T \rightarrow D^{\Omega(f)} \).

Valuations are maps \( \beta : V \rightarrow D \). For the purpose of the next definition, write \( \beta \sim_i \gamma \) iff for all \( j \neq i \) and all \( t \in T \): \( \beta(x_j, t) = \gamma(x_j, t) \). The value of a term is now a function \( T \rightarrow D \), established in the following way.
2. Predicate Calculus

Table 2.2: The final part of the parser

let table = ref (Array.make_matrix 1 1 []);

let parse g x =
  let c = initialize_chart g x
  in let n = Array.length c in
  fill c;
  table := c;
  let final = c.(0).(n-1)
  in let ls = List.filter (fun p ->
      (fst p) = (Bas "t")
   ) final
  in let h = List.partition (fun p ->
      match (snd p) with
      | Bl true -> true
      | Bl false -> false
      | _ -> raise (Error "Type to Category mismatch.")
    ) ls
  in match h with
    ([], []) -> print_endline
    ("This is not a sentence.")
  | (_, []) -> print_endline
    ("This sentence has only true readings.")
  | ([], _) -> print_endline
    ("This sentence has only false readings.")
  | (_,_) -> print_endline
    ("This sentence has true and false readings.")
  ;;

  let show () = !table
2.5. Temporal Predicate Logic

1. \( x^\beta(t) = \beta(x) \).

2. \( f(s_0, \ldots, s_{\Omega(f)-1})^\beta(t) = I(f, t)(s_0^\beta(t), \ldots, s_{\Omega(f)-1}(t)) \).

\[
\langle D, \beta, t \rangle \models r(s_0, \ldots, s_{\Omega(r)-1}) \iff \langle s_0^\beta(t), \ldots, s_{\Omega(r)-1}(t) \rangle \in I(r, t)
\]

\[
\langle D, \beta, t \rangle \models \neg \chi \iff \langle D, \beta, t \rangle \not\models \chi
\]

\[
\langle D, \beta, t \rangle \models \chi \land \varphi \iff \langle D, \beta, t \rangle \models \chi \text{ and } \langle D, \beta, t \rangle \models \varphi
\]

\[
\langle D, \beta, t \rangle \models \chi \lor \varphi \iff \langle D, \beta, t \rangle \models \chi \text{ or } \langle D, \beta, t \rangle \models \varphi
\]

\[
\langle D, \beta, t \rangle \models \chi \rightarrow \varphi \iff \text{if } \langle D, \beta, t \rangle \models \chi \text{ then } \langle D, \beta, t \rangle \models \varphi
\]

\[
\langle D, \beta, t \rangle \models \exists x \phi \iff \text{there is } \beta' \sim_i \beta : \langle D, \beta', t \rangle \models \phi
\]

\[
\langle D, \beta, t \rangle \models \forall x \phi \iff \text{for all } s \triangleright t : \langle D, \beta, s \rangle \models \phi
\]

\[
\langle D, \beta, t \rangle \models \forall x \phi \iff \text{there is } s \triangleright t : \langle D, \beta, s \rangle \models \phi
\]

\[
\langle D, \beta, t \rangle \models \exists x \phi \iff \text{for all } s \triangleleft t : \langle D, \beta, s \rangle \models \phi
\]

\[
\langle D, \beta, t \rangle \models \forall x \phi \iff \text{there is } s \triangleleft t : \langle D, \beta, s \rangle \models \phi
\]

(implementation of language)

We shall now use this models to implement a temporal semantics. Before we can do this we need to agree on a few basics on how temporal information is used. For example, the meaning of /blue/ and /train/ are functions from time points to individuals.

\[
\lambda t. \lambda x. \text{blue}'(x, t), \quad \lambda t. \lambda x. \text{train}'(x, t)
\]

The result of combining them is of the same type:

\[
\lambda t. \lambda x. \text{blue}'(x, t) \land \text{train}'(x, t)
\]

This is often dealt with using a special adjunct rule; but we like to propose (following Montague) that the semantics of the adjective is rather this:

\[
\lambda P. \lambda t. \lambda x. \text{blue}'(x, t) \land P(t)(x)
\]

The predication is simultaneous. Something is a blue train if it is both blue and a train at the same time. This is different with words like /former/ that change the point of evaluation. The semantics of this word is

\[
\lambda P. \lambda x. \Diamond P(x)
\]
Using explicit time point we get

\[(2.79) \quad \lambda P.\lambda t.\lambda x.(\exists s)(s < t \land P(s)(x))\]

The first step is the definition of language and models. As I said earlier, objects are made of strings using a type constructor. There is now a new type of sets of time points. This is produced via the module Time. The previous program is augmented by this:

\[
type w = T of int
\]
\[
module Time = Set.Make(struct
  type t = w
  let compare = compare
end)
\]

\[(2.80)\]
\[
make_time_points n =
  let s = ref Time.empty in
  for i = 0 to (n-1) do
    s := Time.add (T i) !s
  done;
  !s
\]

The definition of signature remains the same. From now on we shall simply assume that if \(f\) is an \(n\)-ary relation symbol, it is interpreted as an \(n + 1\)-ary relation symbol with the first component being the time.

Relations now are lists of pairs \((t, \ell)\), where \(t\) is a time point and \(\ell\) a list of objects. An interpretation is a function that assigns to each relation symbol \(f\) such a relation.

The next step is to redefine the notion of a model.

\[(2.81)\]
\[
type model = \{\text{domain} : Dom.t; \text{worlds} : \text{Time.t};
  \text{inter} : \text{int}\}
\]

The previous definitions of the parser need only mildly be adapted. The point is that the meanings of sentences are functions from time points to truth values. So if we want to know whether a sentence is true now we need to feed the time point
and evaluate the meaning there. All we need to do next is to feed the model and convert it to meanings for the grammar.

\[
\text{let parse } g \text{ tp } x = \\
\text{let } c = \text{initialize_chart } g \text{ x in} \\
\text{let } n = \text{Array.length } c \text{ in} \\
\text{fill } c; \\
\text{table := } c; \\
\text{let final = } c.(0).(n-1) \\
\text{in let } ls = \text{List.filter (fun p ->} \\
\text{(fst p) = (Bas "t")}) \text{ final} \\
\text{in let } h = \text{List.partition (fun p ->} \\
\text{match (apply (snd p) tp) with} \\
\text{Bl true -> true} \\
\text{| Bl false -> false} \\
\text{| _ -> raise (Error "Type to Category mismatch."))} \\
\text{ls} \\
\text{in match } h \text{ with} \\
\text{([], []) -> print_endline} \\
\text{("This is not a sentence."))} \\
\text{| (_, []) -> print_endline} \\
\text{("This sentence has only true readings."))} \\
\text{| ([], _) -> print_endline} \\
\text{("This sentence has only false readings."))} \\
\text{| (_, _) -> print_endline} \\
\text{("This sentence has true and false readings."))} \\
; ;
\]

Let's return to the implementation of the semantics into OCaml. We can more or less directly translate the formulae. In place of \( f(g) \) we write \( \text{apply } f \text{ g} \); and in place of \( \lambda x. f \) we write \( \text{Fc } (\text{fun } x \rightarrow f) \). For example (2.77) becomes (with m
the model)

$$\text{Fc } (\text{fun } f \rightarrow \text{Fc } (\text{fun } w \rightarrow \text{Fc } (\text{fun } o \rightarrow \text{match } (\text{apply } (\text{apply } (f \ w) \ o) \ \text{with } \text{Bl } p \\ \rightarrow (\text{List.mem } (w, [o]) \ (m.\text{inter } (\text{Rel } "\text{blue}")))) \\ \&\& \ p))))$$

Essentially, after displaying all the lambda-operators we need to take care of the ground clause $\text{blue}'(x, t) \land P(t)(x)$. The second part is easy, since here we just use $\text{apply}$. For the first clause we need to look up the relation via the list $m.\text{inter } (\text{Rel } "\text{blue}")$. This is of the form: pairs consisting of a time point and a list of objects. We check for membership with $\text{List.mem}$. The result is a boolean, so we need to do some type adjustment.
Index

abstract syntax, 17
algebra, 12
alphabet, 11
ambiguity, 16
  lexical, 24
  spurious, 16
  structural, 24
analysis term, 16
argument, 36
  valid, 36
boolean algebra, 15
conclusion, 36
contradiction, 32
exclusive or, 27
formula
  signed, 10
frame, 30
function, 50
function symbol, 12
grammar, 15
homomorphism, 14
model structure, 52
occurrence, 9
predicate logic

language, 50
premiss, 36
product, 16
projection, 16
reading, 23
relation, 50
sign, 11
signature, 12
symbols
  nonlogical, 50
tableau, 10
tautology, 32
tense frame, 32
term, 13, 14, 51, 52
transfix, 21
truth value, 6
unfolding, 16
valuation, 5
  partial, 9
variable, 5
well-formed expression, 5
world, 30
Symbols

$E, M, \{11\}$
$\sim, \cup, \cap, \subseteq, \{12\}$
$2, \{13\}$
$Tm_{\Omega}(V), \{14\}$
$
\langle W, < \rangle, \{30\}$
$\models, \{30\}$
$F, G, P, Q, \{30\}$
$Kt, \{32\}$
Bibliography

