

6.2.2. Functors

Truth-functional connectives express truth-valued functions of truth values, and predicates express truth-valued functions of reference values. A third sort of function not only takes reference values as input but also issues them as output. We will refer to this sort of function as a **reference function** or, in contexts where we do not need a more general concept, simply as a **function**. We will refer to expressions that are signs for these functions as **functors** and refer to the operation of applying a functor as **function application**. We can speak of the result of a function application as a **compound term**.

Functors are incomplete expressions that stand to individual terms as connectives stand to sentences, so we can extend the table of operations in [6.1.1](#) as follows:

| <i>operation</i> | <i>input</i> | <i>output</i> |
|------------------|--------------------|-----------------|
| connective | sentence(s) | sentence |
| predicate | individual term(s) | sentence |
| functor | individual term(s) | individual term |

Signs for mathematical functions provide examples of functors. The expression $7 + 5$ can be analyzed as

Individual terms: 7 5
 Functor: _ + _

But functors are not limited to mathematical vocabulary. Any individual term that contains one or more individual terms can be seen as the result of applying a functor to those component terms. Thus *the oldest child of Ann and Bill* can be analyzed as

Individual terms: Ann Bill
 Functor: *the oldest child of* ___ and ___

And the more complex individual term *the location of the home of Ann's father's best friend* has the following analysis:

Individual term Ann
 Functors ___'s father
 ___'s best friend
 the home of _____
the location of _____

The notation of lambda abstraction was introduced in [6.1.4](#) with an example of a mathematical reference function, and that notation can be

applied to any reference functions. Using it, the first two examples above could be given the analyses:

$$[\lambda xy (x + y)] \underline{Z} \underline{5}$$

$$[\lambda xy (\textit{the oldest child of } x \textit{ and } y)] \underline{\textit{Ann}} \underline{\textit{Bill}}$$

In the case of the third example, we need to use parentheses to show grouping

$$[\lambda x (\textit{the location of } x)] ([\lambda x (\textit{the home of } x)] ([\lambda x (x\textit{'s best friend})] ([\lambda x (x\textit{'s father})] \underline{\textit{Ann}})))$$

And, in general, compound terms should be enclosed in parentheses when they fill a place of a functor or predicate.

In full symbolic notation, unanalyzed functors will be represented by lower case letters and will be written before the individual terms filling their places. Our English notation for a compound term

$$\zeta \tau_1 \dots \tau_n$$

will be

$$\zeta \textit{ of } \tau_1, \dots, \textit{ 'n } \tau_n$$

which is in keeping with the usual way of reading a function application. When we need a general variable for functors we will use ζ , as has been done here, or sometimes ξ .

Using this symbolic and English notation, we can express the final analyses of the examples above as follows:

| <i>symbolic notation</i> | <i>English notation</i> | <i>key</i> |
|--------------------------|-------------------------|---|
| psf | p of s ' n f | [p: $\lambda xy (x + y)$; f: 5; s: 7] |
| oab | o of a ' n b | [o: λxy (<i>oldest child of x and y</i>); a: <i>Ann</i> ; b: <i>Bill</i>] |
| l(h(d(fa))) | l of h of d of f of a | [d: λx (<i>x's best friend</i>); f: λx (<i>x's father</i>); h: λx (<i>the home of x</i>); l: λx (<i>the location of x</i>); a: <i>Ann</i>] |

The symbolic notation for functors that is used here is different from the most common notation for function applications. Here are some examples for comparison

| <i>common mathematical notation</i> | <i>symbolic notation used here</i> | <i>English notation</i> |
|-------------------------------------|------------------------------------|-------------------------|
| f(a) | fa | f of a |
| f(a, b) | fab | f of a ' n b |
| f(g(a)) | f(ga) | f of g of a |
| f(a, g(b)) | fa(gb) | f of a ' n g of b |
| f(g(a), b) | f(ga)b | f of (g of a) ' n b |
| f(g(a, b)) | f(gab) | f of (g of a ' n b) |

The notation used here is a common one in logic and is designed to minimize parentheses and commas. The general rule for interpreting it is this: (i) after a predicate—i.e., after a capital letter—each unparenthesized letter and each parenthetical unit occupies one place of the predicate and (ii) within a parenthetical unit the first letter is a functor and each following unparenthesized letter and each parenthetical unit occupies one place of this functor.

While the English notation for compound terms provides a way of reading logical forms, the last two examples above show that it does not enable us to completely avoid parentheses, for the English notation for these two different forms would be the same without the parentheses. Because the letters used to represent functors and non-logical predicates do not have a fixed number of places associated with them, parentheses can be needed to show where a compound term ends. Although there are verbal ways of dealing with this problem, we will simply use parentheses when they are necessary to avoid ambiguity. Of course, parentheses, like other punctuation, can be reflected in speech and it is natural to mark the difference between f of (g of a) ' n b and f of (g of a ' n b), respectively, by varying the speed with which they are

spoken in ways that might be indicated by “f of g-of-a ' n b” and “f of g of a-' n-b”.

When analyzing sentences, functors are uncovered by analyzing terms as compound. Here is an example:

The cat on the mat was asleep and the dog that had chased it was, too

the cat on the mat was asleep \wedge *the dog that had chased the cat on the mat was asleep*

$[\lambda x (x \text{ was asleep})]$ *the cat on the mat* \wedge $[\lambda x (x \text{ was asleep})]$ *the dog that had chased the cat on the mat*

S (*the cat on the mat*) \wedge S (*the dog that had chased* *the cat on the mat*)

S ($[\lambda x (\text{the cat on } x)]$ *the mat*) \wedge S ($[\lambda x (\text{the dog that had chased } x)]$ *the cat on the mat*)

S(cm) \wedge S (d (*the cat on the mat*))

Scm \wedge S (d ($[\lambda x (\text{the cat on } x)]$ *the mat*))

S(cm) \wedge S(d(cm))

both S fits c of m and S fits d of c of m

[S: $\lambda x (x \text{ was asleep})$; c: $\lambda x (\text{the cat on } x)$; d: $\lambda x (\text{the dog that had chased } x)$; m: *the mat*]

It will be easier to make a full analysis of a sentence if you choose the largest individual terms possible when analyzing an atomic sentence or compound term. Otherwise, part of the logical form will end up being obscured by an abbreviation unless you go on to analyze the body of an abstract. That sort of problem would have arisen in this example if we had analyzed the first conjunct as

the cat on the mat was asleep

$[\lambda x (\text{the cat on } x \text{ was asleep})]$ *the mat*

While it is possible to recover from such choices by analyzing the bodies of abstracts, some care is needed in the choice of variables so each variable ends up having the correct antecedent, and we will not go on to consider how this may be done. (It would not merely create difficulties but actually be wrong to choose *the cat* as a component individual term of *the cat on the mat was asleep*; we will discuss this issue in [6.2.3](#).)

In the presence of functors, the potential for undefined terms increases considerably. Even if *the cat on the mat* has a non-nil reference value, *the cat on the refrigerator* may not—to say nothing of *the cat on the house of Ann’s father’s best friend* or *the cat on 6*. That is, functors accept a large variety of inputs and can be expected to issue output with

undefined reference for some of them. This problem can be reduced somewhat by limiting functors to input of certain sorts. That is usually done by assigning individual terms to various **types** and allowing only individual terms of certain types to serve as inputs to a given functor. For example, the functor $\lambda xy (x + y)$ might be restricted to numerical input. However, it is not easy to eliminate all undefined terms by use of types, and we will not introduce into our analyses the complications needed to use types.

Glen Helman 21 Oct 2004