# The Application Operator
# A Thought-Dialogue with E. W. Dijkstra

**Haskeller**: Considering that mathematics, formalization, elegance, provability and so on have been the hall-marks of your work, why do you still stick to the antiquated imperative paradigm?
**EWD**: What I am sticking to ... I am not very sure ...
It is very important to me that the language we use to talk in and communicate with be perspicuous. All expressions should satisfy the Leibniz property that equals may be substituted for equals, or, as you would say (*pompously*) they should be referentially transparent, functions should be first-class, (*end pompously*) above all application plays the primary role ...
**Haskeller**: WHOA! Hold it! You are misappropriating our religion! In Haskell all expressions are referentially transparent and functions are first-class.
**EWD**: Maybe semantically. Not syntactically.
**Haskeller**: What do you mean by ''syntactically referentially opaque?''
**EWD**: No I was not referring to referential transparency. I was referring to first-classness.
**Haskeller**: You mean that in Haskell although functions are semantically first-class they are not first-class syntactically?
**EWD**: Yes.
**Haskeller**: Can you explain this bizarre notion of yours?
**EWD**: Well I'll try... You are so devoted to your favourite functional language that if I show you the problem directly you will not be able to see it... So let me start with trivial examples. Let us take school algebra. Look at the expression $xy + 2xz$ What is between $x$ and $y$?
**Haskeller**: Nothing.
**EWD**: But something is intended?
**Haskeller**: Of course! Multiplication.
**EWD**: Does your beloved Haskell allow such expressions?
**Haskeller**: Of course not!
**EWD**: Why?
**Haskeller**: It is highly irregular – for addition the symbol + but for multiplication nothing.
**EWD**: But the traditional mathematicians use this elision of symbols?
**Haskeller**: Maybe so. But it compromises clean semantics and finally first-classness.
**EWD**: Why?
**Haskeller**: The traditional mathematician would have trouble referring to multiplication. Either he would have to use a symbol that he does not normally use or he would have to say, ''The nothing between $x$ and $y$.''
**EWD**: So semantics are not clean. But how does that spoil first-classness?
**Haskeller**: If something has no syntactic form we cant talk of its semantic properties. At least in FORTRAN, Pascal or C we can say: ''+ and * may not be passed into or returned from functions.'' How is the traditional mathematician to say it? ''The nothing between $x$ and $y$ — which, mind, is different from the nothing between $y$ and + — cannot be passed ...''?
**EWD**: So if something is not syntax, it cannot denote. If it cannot denote, it cannot be first-class or otherwise. Right?
**Haskeller**: Yes, but I don't see what that has to do with Haskell.
**EWD**: All in good time. But will you allow me to cite one more example – concatenation in SNOBOL or awk?
**Haskeller**: (*Jumping*) It is the same thing. Elision denotes string concatenation. Since such concatenation is very frequent in these languages, it may be convenient in the specialized context of these languages but the same problems as with school-algebra arises. Nothing is used to denote concatenation; hence we cannot talk of its first-classness.
**EWD**: Yet you claim that functions are first-class in your so-called functional language?
**Haskeller**: But of course! Any identifier (including an operator symbol) can denote a function. And such a function may be passed in and returned and stored and ...
**EWD**: Slowly. Slowly. You are mouthing all the usual FP indoctrination which is all very well. But let us go back to our two examples.
An important operator – multiplication – is elided from arithmetic expressions. That compromises first-classness. Right?
**Haskeller**: Right.
**EWD**: The most important operator in awk and SNOBOL is elided. That compromises first-classness. Right?
**Haskeller**: Right.
**EWD**: What is the most important operator in Haskell?
**Haskeller**: (*Beaming*) There is no 'most-important-operator' in Haskell. Everything is equal. Everything is uniform. That is first-classness.

**EWD**: Spare me the advertisement. Look before you brag. Look at the Haskell expression

```
  map (1 + ) xs
```

What is between `1` and `+`?
**Haskeller**: Space.
**EWD**: And what does it denote?
**Haskeller**: Nothing.
**EWD**: And what is between `+` and `)`?
**Haskeller**: Space.
**EWD**: And what does it denote?
**Haskeller**: Nothing.
**EWD**: And what is between `map` and `(`?
**Haskeller**: Space.
**EWD**: And what does it denote?
**Haskeller**: Application.
**EWD**: Aha! There you are!
**Haskeller**: What do you mean "There you are!" Application is the most important concept in functional programming languages. How can you write a program without it?
**EWD**: (*Laughing*) So there is a most important operator after all!
**Haskeller**: (*Irritated*) Its not an operator. Its the fundamental glue for binding together any functional program.
**EWD**: Just as concatenation is the glue in SNOBOL and awk?
**Haskeller**: (*Gulp*) Er ... I ... I am not sure what you are driving at ...
**EWD**: I am sorry to be harsh with you but the fact is that in Haskell, functions are given poorer treatment than say in C. In C, as in traditional mathematics, at least the parenthesis surrounding the parameter list denotes function application. In Haskell you have nothing.
**Haskeller**: This is going too far. You cannot handle a function in C with anything like Haskell's power. And you can never create a function at all.
**EWD**: (*Laughing*) I was only pulling your leg. In Haskell nothing denotes application, in C (as in traditional mathematics) parenthesisation is overloaded, sometimes it parenthesises, sometimes it denotes function application. Each is worse than the other. Anyway we were talking of syntax, remember?
**Haskeller**: If its only a question of syntax then you can define an apply operator. Hudak does it in the Haskell tutorial using a `$`. Thus:

```
  f $ x  =  f x
```

Then you can use `$` wherever you want to make application explicit.
**EWD**: Hmm... can you write the definition?

```
  length$[] = 0
```

**Haskeller**: Of course.
**EWD**: Are you sure?
**Haskeller**: Yes.
**EWD**: Then how does Haskell distinguish between the first definition which defines `$` and the second which uses `$` to define `length`?
**Haskeller**: (*Gulp*) Er... No you cannot use a `$` on the lhs.
**EWD**: (*Laughing*) So you are a traditional mathematician after all. Sometimes he writes $xy$, sometimes $x{\times}y$ and sometimes $x \cdot y$. In the same way you sometimes write `f x` and sometimes `f$x`
**Haskeller**: Well OK. So what do you think the syntax should be?
**EWD**: Simple. If application is the central concept in functional programming languages and regularity is a cardinal design principle (I hope it is!) then application must always be explicit.
**Haskeller**: Do you want us to write?

```
  length$(x:xs)  = 1 + length$xs
```

**EWD**: Um...
**Haskeller**: I find the `$`s ugly!
**EWD**: The `$`s are your choice.
**Haskeller**: (*Exasperated*) But then what do you want?
**EWD**: This is a question on design... Application must show but it must not dominate. This calls for an operator that even when ubiquitous, will not overwhelm. For myself, after I realised that functions and application are being given second-class treatment in computing science, (and that is because of the bad habits we inherit from mathematicians) I have started using a dot for this purpose. Thus: $map.\,(1+).\,xs$
**Haskeller**: Does not look very regular to me. Why is there no dot between 1 and +?
**EWD**: Oh that's easy. $map.\,((+).\,1).\,l$ In fact for any operator * we may define sections by postulating the following as equal.

$$x * y$$
$$(x*).\,y$$
$$(*y).\,x$$
$$(*).\,x.\,y$$

We treat the others as short-forms for the last.
**Haskeller**: I am somehow very uneasy ...
**EWD**: Why?
**Haskeller**: In Haskell we already use the dot for composition. What do we do then for composition?
**EWD**: Hmm ... As I said this is a matter of design ... I do not know your Haskell very well but is it not true that you use indentation to demarcate blocks, individual elements of blocks and the like?
**Haskeller**: (*Warily*) Yes, and what is wrong with that?
**EWD**: O nothing! It is good, very good. A language that makes clean programming

mandatory is good. I asked because if the semicolon is free you could use it to denote composition, you could even define it in diagrammatical order thus:

$(f; g).\, x = g.\, (f.\, x)$

**Haskeller**: But the semicolon is not free, sorry. It is allowed as an explicit alternative to separate declarations.

**EWD**: (*Puzzled*) But you just said that indentation is used for that purpose?

**Haskeller**: You may use indentation or you may use braces and semicolons. Its the programmer's choice.

**EWD**: (*Laughing*) Like choosing a hot-dog or a sandwich at a fast-food restaurant?

**Haskeller**: What is wrong with allowing the programmer some choice?

**EWD**: Well then why not allow him some more choice? Like assignments, pointers, gotos...?

**Haskeller**: Ridicule me if you like ... Yes I've got it! There is a serious problem with application as a first-class operator. If application is itself an operator it can itself be sectioned. Then we are in trouble if we listen to you. Or are you suggesting that application be an exception to the rule and not be allowed in sections? That would be an ugly wart on the face of Haskell.

**EWD**: I never suggested making application special. Sectioning it gives some nice results. Here is one for the left-section:

$f$
by $\eta$-abstraction
$= \lambda x \rightarrow f.\, x$
left sectioning the dot
$= \lambda x \rightarrow (f.).\, x$
by $\eta$-reduction
$= (f.)$

**Haskeller**: What does this mean?

**EWD**: It has a profound significance. I says that a function is nothing more than what it means to apply it.

**Haskeller**: Ok. What about the right section. I cannot see what use that may be put to.

**EWD**: On the contrary it has many practical uses Here is one. The function *pam* (converse of *map*) has the following informal specification

$pam.\, x.\, [f_1,\ f_2, \dots\ f_n]\ =\ [f_1.\, x,\ f_2.\, x, \dots\ f_n.\, x]$

Using the right-sectioned dot it is easily defined as:

$pam.\, x.\, fl\ =\ map.\, (.\, x).\, fl$

It would be much harder to discover this if the application were not evident in the specification.

Also the first result can be generalised as

$f = (f.) = ((f.).) = \cdots$

**Haskeller**: *pam* is nice but the last is horrible. How do we implement it?

**EWD**: Do you have trouble implementing + just because $x = x + 0 = x + 0 + 0 = \cdots$ ?
Or do you have trouble implementing Hindu-Arabic numerals just because we can write 3 as 03 or 003 or $\cdots$?

**Haskeller**: I'll accept if you can give me a use for the full-section.

**EWD**: Do you accept that one of the chief benefits of $\lambda$-calculus is that higher-order relations of profound significance can be expressed as $\lambda$-expressions? eg The fact that functions can be written in curried or tupled notation can be expressed in the object notation in terms of the combinators: *curry* and *uncurry*?

**Haskeller**: Sure

**EWD**: How would you express the idea of application?

**Haskeller**: $apply = \lambda x\ y \rightarrow x\ y$

**EWD**: I would write that as $apply = \lambda x\ y \rightarrow x.\, y$

**Haskeller**: What's the advantage?

**EWD**: For one thing it is very clear that the *xy* to the left of the $\rightarrow$ is just syntax whereas to the right we are denoting something. Better still observe the following proof.

$apply$
by definition
$= \lambda x\ y \rightarrow x.\, y$
by definition of full section
$= \lambda x\ y \rightarrow (.).\, x.\, y$
by left-associative convention
$= \lambda x\ y \rightarrow ((.).\, x).\, y$
by $\eta$-reduction
$= \lambda x \rightarrow ((.).\, x)$
by $\eta$-reduction
$= (.)$

Which brings us back to where we started. Application is denoted by a symbol. And the calculus is enough to arrive at it.