A “UML” equivalent for functional programming

Yusuf M Motara

A parser that produces type $\Diamond$, using function $f$, instead of recognized text.

 Implemented as: $\Diamond | >> f$

 A "bind" variant exists.

 - $\Diamond! $ → parser producing $\Diamond$
 - $\_ $ → parser producing default $\Diamond$

$\Diamond#$ → a parser whose output is discarded.

Implemented as: opt $\Diamond$

Equivalent to: $\Diamond#$

Implemented as: optional $\Diamond$

Turn a failure into a warning.

- $\sim \Diamond $ → $\Diamond$
- $\sim \otimes $ → $\times$
- $\sim \times $ → $\times$

Implemented as: attempt $\Diamond$
No way to **model high-level, structural** design of a functional system
- **structural**: expresses the way in which elements relate to each other
- **high-level**: expressing only relevant parts of a system
- **model**: something that is usable in place of the original with respect to some purpose

No shared language between functional programmers, OO programmers, project managers, clients, business analysts, …
- Reduced opportunity for collaborative design and shared understanding of problem domain
Express the structure of a functional design, in a way that is useful to programmers AND other stakeholders.
Programs, as human artifacts, have some *meaning* (semantics) on a *human* / real-world level.

*Types* encode part of this meaning.

*Useful* models reveal meaning that is otherwise non-obvious.

Even better models can be used to *design*, which is a form of forecasting the future!
The basis of all design is an underlying philosophy.

Key points:
- Underlying philosophy must be invariant — gives a shared perspective to practitioners
- Meaning is built on underlying philosophy
- Meaning can evolve along with program/system

UML’s “Theory of Forms” philosophy won’t work!
The semantics of a functional program resemble a language game (Wittgenstein 1953).

1. Every language-game has some purpose, and may have implicit presuppositions.
2. Words have no independent meaning outside of a language game, and may have specific meanings for only that language game.
3. Inexact meanings are fine! Meanings are only separated to the extent needed to avoid misunderstanding within the language game.
4. Words may have multiple, independent meanings. An unused word is meaningless.
5. The point of language is to faithfully describe; “solutions” are just good descriptions.
6. There is nothing that is naturally composite or naturally separate.
7. Nothing is gained by asserting that two things are more similar than they are, based on a shared heritage.
8. The meaning of a sentence is more important than the way in which it is constructed.
9. Two sentences with the same meaning are considered to be the same.

Key insight: fundamental FP techniques (composition, HOFs, closures, etc) have natural language analogues!
Within a computer natural language is unnatural.

~ Alan J. Perlis

There will always be things we wish to say in our programs that in all known languages can only be said poorly.

~ Also Alan J. Perlis

However — not all language-games are games of natural language!
The natural language structure of *mathematics* can be described as having the following features (Ganesalingam 2013):

1. Abbreviative **definitions**
2. Implicit presuppositions
3. Adaptive, layered meanings
4. Rhetorical blocks involving variable definition, naming, presuppositions, consequence, cross-referencing, conclusions, product types, and sum types.
Key points:

- FP retains **strong ties** to mathematics.
- FP techniques have **natural language** analogues.

Can we describe the design of a functional system using the philosophical basis of a *language-game*, and the guideline constrained language of *mathematics*?

Language of Mathematics + Language Game = Language for FP Design?
In languages, we have:

- **dictionaries** (“what words exist? what do they mean?”)
- **thesauri** (“which words have similar meanings?”)
- books of **etymology** (“where does this word come from?”)

Let’s start from there.
A parser which, when given some text, either:
- Recognizes the text
- Fails to recognize the text
- Issues a warning about why it can't recognize the text.

A parser which has successfully recognized text.

A parser which has failed to recognize text.

A parser which has issued a warning.

A parser that produces type Ω, using function f, instead of recognized text.

Implemented as: Ω | >> f
A “bind” variant exists.

Ω[f] → parser producing Ω
_ → parser producing default Ω

Ω# → a parser whose output is discarded.
_ → Ω.

Ω+Ω → a parser producing the results from both parsers.
_ → Ω.

Implemented as: Ω .>. Ω
Single-result variants:
Ω#+Ω implemented as Ω .>. Ω
Ω+Ω# implemented as Ω .>. Ω

Equivalent to: Ω#+Ω#

Useful for parsing text between delimiters.

Implemented as: between Ω Ω Ω

Ω< | >Ω
Left-associative.

Ω>pt → a Ω value exists
Ω>0 → a Ω value doesn't exist

Ω>pt
Turn a failure into a qualified success.

Ω>pt → Ω that produces type Ω⊥ instead of type Ω.
_ → Ω that produces type Ω0 instead of type Ω.

Implemented as: opt Ω
Equivalent to: Ω#

Ω?
Implemented as: optional Ω

~Ω
Turn a failure into a warning.

~Ω → Ω
~Ω → x
~x → x

Implemented as: attempt Ω

A parser that recognizes sequentially matching text, producing a list.
A parser which, when given some text, either:
- Recognizes the text
- Fails to recognize the text
- Issues a warning about why it can't recognize the text.

A parser which has successfully recognized text.

A parser which has failed to recognize text.

A parser which has issued a warning.

A parser that produces type \(\emptyset\), using function \(f\), instead of recognized text.

Implemented as: \(\emptyset \mid >> f\)

A "bind" variant exists.

+ \(\emptyset!\emptyset\) \(\emptyset\) \(\emptyset\)
- \(\emptyset!\emptyset\) \(\emptyset\) \(\emptyset\)

A parser producing \(\emptyset\).
- \(\emptyset\) \(\emptyset\)

A parser producing default \(\emptyset\).

\(\emptyset\#\) \(\emptyset\) \(\emptyset\)
- \(\emptyset\#\) \(\emptyset\) \(\emptyset\)

A parser whose output is discarded.
- \(\emptyset\) \(\emptyset\)

\(\emptyset\#\) \(\emptyset\) \(\emptyset\)
- \(\emptyset\#\) \(\emptyset\) \(\emptyset\)

A parser producing the results from both parsers.
- \(\emptyset\) \(\emptyset\)

Implemented as: \(\emptyset \ggg \emptyset\)

Single-result variants:
\(\emptyset\#\#\) \(\emptyset\) \(\emptyset\)
\(\emptyset\#\#\#\) \(\emptyset\) \(\emptyset\)

Equivalent to: \(\emptyset\#\#\emptyset\#\#\)

Useful for parsing text between delimiters.

Implemented as: \(\emptyset \emptyset\)
Dictinary
+ "Etymology"
+ "Thesaurus"

A parser which, when given some text, either:
- Recognizes the text
- Fails to recognize the text
- Issues a warning about why it can't recognize the text.

A parser which has successfully recognized text.

A parser which has failed to recognize text.

A parser which has issued a warning.

A parser that produces type $\Theta$, using function $f$, instead of recognized text.
Implement as: $\Theta | >> f$
A "bind" variant exists.

$\Theta f\Theta \rightarrow$ parser producing $\Theta$
$\_ \rightarrow$ parser producing default $\Theta$

$\Theta# \rightarrow$ a parser whose output is discarded.
$\_ \rightarrow \Theta$.

$\Theta+\Theta \rightarrow$ a parser producing the results from both parsers.
$\_ \rightarrow \Theta$.

Implement as: $\Theta >> . \Theta$
Single-result variants:
$\Theta# + \Theta$ implemented as $\Theta >> . \Theta$
$\Theta+\Theta#$ implemented as $\Theta >> . \Theta$

Equivalent to: $\Theta# + \Theta + \Theta#$

$\Theta[\Theta] \Theta$
Useful for parsing text between delimiters.
Implement as: between $\Theta \Theta \Theta$
(Suggested) reserved for notation:

- ( and ) for grouping
- ♻ for “any value/type”
- subscripts for sum type cases
- → for mapping cases
- ‣ and • for discrete cases / grouping respectively
- _ for “any other case”
Closing thoughts

• Symbols and namespaces
  ○ What to do when you want to use the same symbols in another context?

• Symbolic overload
  ○ What is “overwhelming”, what is “rich”, and what is “sparse” or “poor”? What design guidelines should there be?

• What is modelled depends on what needs to be modelled
  ○ Is there a guideline set of things that needs modelling?
  ○ What do we mean by need, and why would it be a need?
Future Work & Questions

- Tooling
- Refining notation
- Case studies
- ...and much more!

“I’m Hiring” 😊