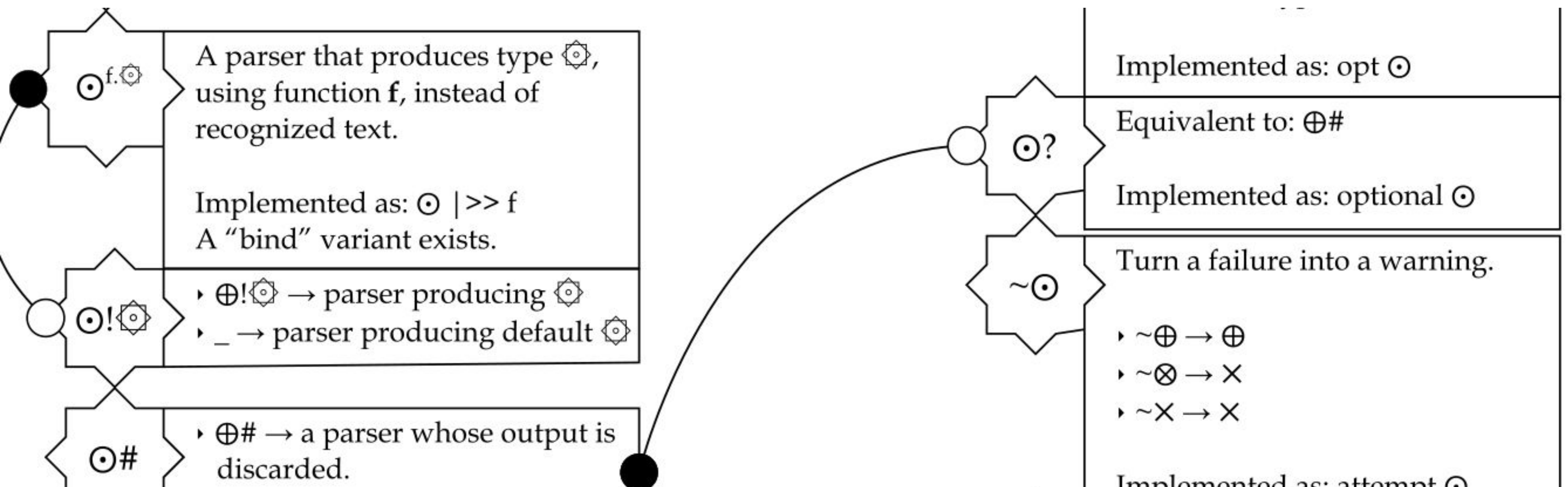
A “UML” equivalent for functional programming

Yusuf M Motara



Department of Computer Science

Problem

- No way to <u>model</u> <u>high-level</u>, <u>structural</u> 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

Ideal Solution

Express the structure of a

functional design,

in a way that is

useful

to programmers AND other stakeholders

Semantics

Programs, as human artifacts, have some *meaning* (semantics) on a *human* / real-world level.

```
List.map ((*) 1.15)
```

Types encode part of this meaning.

$a \rightarrow [a] \rightarrow [a]$

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!

Design

The basis of *all* design is an underlying philosophy.

Philosophy ➡️ Meaning ↔ Program

Key points:

- Underlying philosophy must be **invariant** — gives a *<u>shared perspective</u>* to practitioners
- Meaning is *built on* underlying philosophy
- Meaning can **evolve** along with program/system

UML’s “Theory of Forms” philosophy **won’t work**!

Language-games (1 of 2)

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.

Language-games (2 of 2)

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!

Programming isn’t natural language 😒

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 Language of Mathematics

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.

A Language of Functional Programming

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?

Usefulness revisited

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.

“Dictionary”

⊙
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.

$⊙^{f.⚙}$
A parser that produces type ⚙, using function **f**, instead of recognized text.

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

⊙!⚙
- ⊕!⚙ → parser producing ⚙
- _ → parser producing default ⚙

⊙#
- ⊕# → a parser whose output is discarded.
- _ → ⊗.

⊙+⊙
- ⊕+⊕ → a parser producing the results from both parsers.
- _ → ⊗.

Implemented as: ⊙ .>>. ⊙

Single-result variants:
⊙#+⊙ implemented as ⊙ >>. ⊙
⊙+⊙# implemented as ⊙ .>> ⊙

⊙[⊙]⊙
Equivalent to: ⊙#+⊙+⊙#

Useful for parsing text between delimiters.

Implemented as: between ⊙ ⊙ ⊙

⊙ | ⊙
- ⊕ | _ → ⊕
- ⊗ | ⊕ → ⊕
- _ → ⊗

Implemented as: ⊙ <|> ⊙
Left-associative.

$⚙^{opt}$
- $⚙^1$ → a ⚙ value exists
- $⚙^0$ → a ⚙ value doesn't exist

$⊙^{opt}$
Turn a failure into a qualified success.

- $⊕^{opt}$ → A ⊕ that produces type $⚙^1$ instead of type ⚙.
- _ → A ⊕ that produces type $⚙^0$ instead of type ⚙.

Implemented as: opt ⊙

⊙?
Equivalent to: ⊕#

Implemented as: optional ⊙

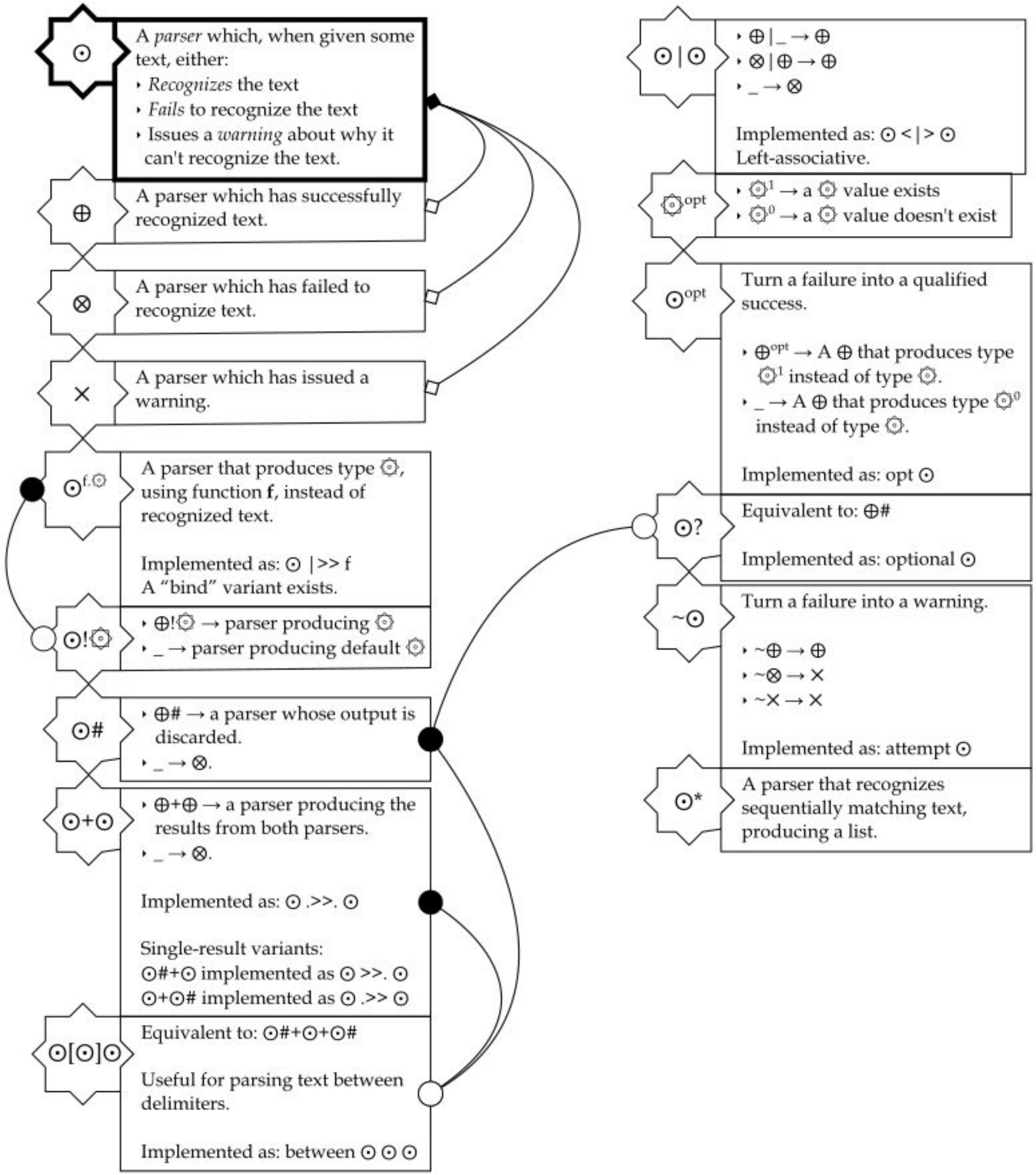
~⊙
Turn a failure into a warning.

- ~⊕ → ⊕
- ~⊗ → ×
- ~× → ×

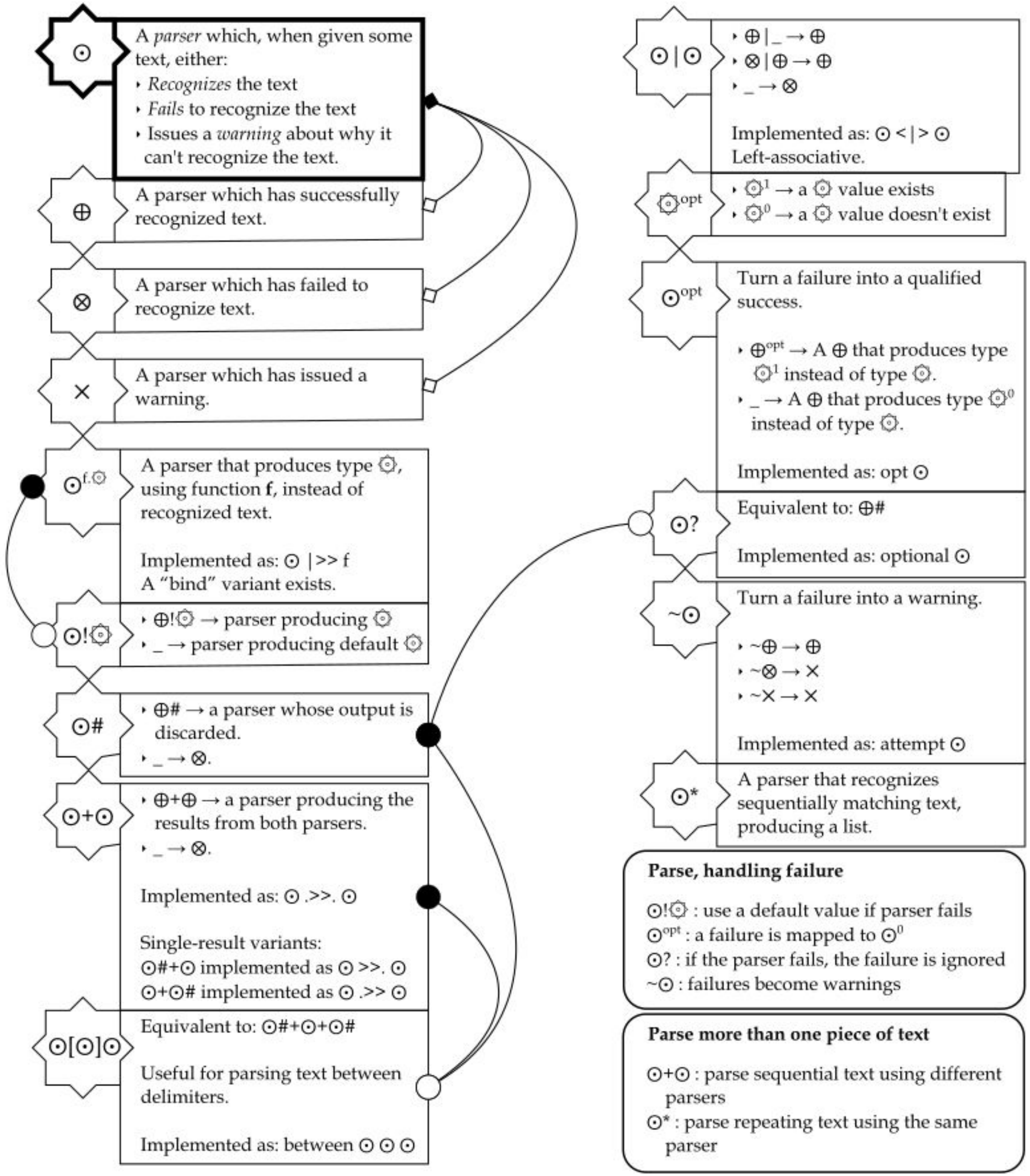
Implemented as: attempt ⊙

⊙*
A parser that recognizes sequentially matching text, producing a list.

"Dictionary"
+ "Etymology"



“Dictionary”
+ “Etymology”
+ “Thesaurus”



Reservations

(Suggested) reserved for notation:

- (and) for grouping
- ۞ for "any value/type"
- subscripts for sum type cases
- $\rightarrow$ for mapping cases
- ‣ and • for discrete cases / grouping respectively
- _ for "any other case"

Closing thoughts

- Symbols and namespacing
  - 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" 🙂