# Thesis planning

Jonathan Lam

9/13/21

## Contents

# 1 Introduction

## 1.1 Problem/Motivation

For practical and historical reasons, programming languages have remained in a textual form that is difficult for beginners to learn. While modern programming languages are arguably more user-friendly than older languages (with more English-like syntax, more consistent syntax) and IDEs have much better linting and static analysis support (e.g., Microsoft's Language Server Protocol (LSP)), all practical modern programming languages are still focused on the same syntax-based textual format as back when FORTRAN was invented in the 1950's.

This was useful out of necessity in the past, and because programming was in the land of geeks who could program directly in hexidecimal machine instructions and write their own device drivers [citation: Linus Torvalds]. However, programming and software development as become a commonplace tool not only for the dedicated programmer, but also for engineers, scientists, artists, and professionals in many unrelated fields. Thus, improving the quality of programming education is becoming more important as the need for quality programs and programmers is improved.

As a programming tutor for several years, and mostly-self-taught programmer for even more years, I have had experience with dozens of programming students and their struggles. Of course, syntax is a very common concern; while it offers efficiency for veteran programs and IDE aids are very effective, the apparent difficulty of syntax can easily overwhelm the beginner programmer [citation: Hedy programming language], which can take attention off of the greater ideas. This effect is compounded when being immediately exposed to a number of different programming tools at the same time (e.g., the introductory course at my school pushes Git + Bash + C on the programmer in the first two weeks). The general cause is that source code is (and always has been) textual and non-structural, which is prone to errors. The textual source code representation has other problems, such as overwhelming the user by a wall of plain text (syntax highlighting, "zen mode," and code folding only mitigate this problem slightly), and being a fundamentally linear format (which may not be the most intuitive since the programming language features fundamentally form a graph).

Besides syntax and other text-related difficulties, there are a number of other common difficulties shared amongst students related to programming languages and software development (without taking a class about software development principles), such as:

- *Translating ideas to code*: Taking an abstract set of specifications and implementing them in a controlled fashion. Untrained programmers can be frozen with "where should I start?" or miss implementation of many of the specifications without a clear definition of them. This can be allieved with a structured top-down approach, proper documentation, and specification of goals.

- *Clear and consistent documentation*: There is the quip floating around the Internet [citation?]: "Before the weekend, only God and I know what this code does. Now only God knows." We often get caught up with the implementation of some thought (the translation of ideas to code) that we forget what the original idea is. Having working with students, and even looking at the code of my coworkers in a professional environment, this is too often true, and the solution is very simple: a structured approach to documentation. Literate programming [citation: Knuth], "self-documentation," or "implicit documentation" are similar terms used to describe a simple solution, as well as generally clean code.

- *Comprehending error messages*: Error messages (at compile- or runtime) are often linked with the call stack or line numbers, which is a structural feature that may not reveal the intent of the code at that particular point. Of course, meaningful function names are useful to make error messages more meaningful for you or someone up the stack.

- *Inability to translate ideas between programming languages*: Certain languages embody certain programming paradigms better than others. Often scientific programming students get stuck in the imperative programming style (e.g., if first learning C) and produce spaghetti code in languages such as Python, which is much better suited for FP concepts (e.g., building meaningful composite operations out of `map` or `filter` primitives).

I propose that a programming model based on intentional programming may be useful to ease some of the common complaints about current programming education, as well as encourage general software engineering best practices and build practical code.

## 1.2 Intentional programming

Intentional programming [citation: The Death of Computer Languages, the Rise of Intentional Programming] was a concept engendered in the 1995 at Microsoft Research by Charles Simonyi, the creator of the WSIWYG editor and Microsoft Word. Simonyi suggested that the rate of programming language design was greatly slowed, and that there were still major unsolved problems when developing code. These problems include language non-intercompatibility, language non-intracompatibility, the mixing of intent (functional description) with implementation, the lack of "natural" notation, and comments being unused by the computer. Moreover, enough programming languages have proliferated that we are able to identify "genes" (features or "memes," using the original meaning of the term) carried throughout the languages and making them (un)desirable – such as garbage collection, object-oriented design, type systems, and special keywords or control-flow constructs – that we may shift our focus from programming in languages to programming in these generic language features.

The solution is to raise software to a level of abstraction that encodes functional intents (*what are you trying to do?*) separately of implementation (*what language construct should I use to do it, and what syntax would that entail?*). No programming language is perfectly intuitive to every user, and no syntax is perfectly natural for every user. Thus, by having a user encode their functional intent in terms of their own vocabulary, using their own predefined grammar. This gives rise to the idea of programming with "intents" as the building block of a program – blocks of presudo-code with a well-defined purpose and API that are expressed in the user's own notation.

Benefits of this system include the abolishment of predefined syntax (we are now working at the abstract-syntax tree (AST) level), more natural editing of code using a GUI (akin to how a WSYWYG document editor allows easier formatting of rich text than a plaintext editor), self-documenting code, and simpler mechanisms for structural editing of code (e.g., refactoring).

At Microsoft, Simonyi was able to demonstrate this idea by creating an IP IDE [citation]. Simonyi left Microsoft to found Intentional Software [citation], which developed tools implementing the intentional programming paradigm and improve programmer productivity. Since the creation of Intentional Software, there has not been much literature on intentional programming. At the time of writing, it is difficult to find much information about the company or its products, which seem to be never released except to some of the company's partners [citation]. Intentional Software was acquired by Microsoft in 2017.

TODO: do some digging about Intentional Software – did it ever produce any software?

## 1.3 Intentional programming for education

Intentional programming and variants have been attempted for use in the industry, but not aimed at students in programming. Intentional Software is highly geared towards domain experts who wish to incorporate more of their nontechnical specifications in the software system.

There are a number of benefits that a level of metaprogramming may bring to handling the problems with learning programming discussed earlier. This may include:

- *Code annotations*: The simplest implication of code generation is that it will force users to include documentation about each statement or small group of statements. Very rarely is a full description of the code included within the source itself – annotations are often included separately because text-based source code is not a good medium for explanation. This can be implemented in several ways: we can decide that every element of a program should fall under some annotated intent.

- *Eliminating syntax errors*: As mentioned earlier, we are working on the level of abstract syntax trees, and syntax errors will become difficult. In most cases, syntax is essentially abolished in favor of user-defined phrases.

- *Top-down programming*: Specifying a functional API before filling in the details (implementation details). This ensures proper documentation and well-specified APIs, and the code is always in a "complete" (compileable) state. This goes hand-in-hand with test-driven development (TDD).

- *Attention*: "Code zooming" (folding) on the level of (functional) intents, rather than (structural) blocks, reduces cognitive overload by only showing information related to the current intent, or a small number of nested intents.

- *Polyglot programming*: Different programming languages have different strengths in different domains, but it may be overwhelming to learn the proper syntax for each language.
- *Natural representations*: This is perhaps the most powerful aspect of intentional programming (more specifically, language-oriented programming), and which allows for quicker comprehension and improved recall than plaintext documents. This manifests itself in several ways:
  - *Personalized vocabulary*: The user expresses and idea in a phrase that is easy to understand. These phrases have placeholders for their arguments, which allow for an arbitrary combination of prefix, infix, or postfix notation without worrying about syntax ambiguities (since we eliminate parsing). Library writers are not bound by the terseness or syntax of the language, and are instead encouraged to express intent APIs in a natural prose-like manner. Users are also encouraged to freely alias intents using more intuitive phrasing. This also allows for seamless internationalization.
  - *Graphical program layout*: The program projection (view) and navigation is less cluttered than a plaintext document. Visual representations may also allow for non-linear (i.e., 2-D or even 3-D) representations of a program (like Scratch).
  - *Data projections*: Data that are well-suited to visual representations (e.g., equations, tables) are not limited to textual representations in a graphical editor.
- *Learn by example*: As opposed to learn by error, students may be able to express a higher-level idea using the IP representation and generate code in the target language(s). They can gain an undestanding of the language's syntax by examining the generated code rather than learning it by trial and error or by reading documentation.

## 1.4 Concerns

One of the major issues is that this system should not make programming more confusing or less intuitive. What we aim to do is encode generic traits inside an intermediate representation (pseudocode) that does not have those traits, and is smart enough to generate correct code. The problem with arbitrarily abstracting programming languages is that we need a precise formal specification in order to actually generate working code [citation:

`https://qr.ae/pGc3tj`, `https://qr.ae/pGc3tf`] – this is what a programming language is. Yet we can define high-level software by using many layers of abstraction. Thus, this is *not* a project about arbitrary code synthesis using AI or other generative methods; this is a framework to guide the programmer towards building cohesive, well-documented code as a means towards correctly implementing intent.

Another concern is that this software should generate correct code. By loosening the restraints of formal language and working on a level that is more general than any one programming language, we may produce code that is not syntactically or semantically correct. A considerable amount of knowledge about the underlying source code is necessary to generate correct code, and this limits the ability for the system to be language-independent.

One last major concern is that we cannot enforce that the user's intent lines up with the implementation. This whole project is an essay towards improving the chances of this happening, but there is no enforcement. There needs to be a better metric towards accomplishing this.

A major goal of this project is determining how to correct these issues. A possibility is to add API-driven constraints ("duck typing") to the language as a very basic type checking – however, this is circular as there is no way to check the correctness or completeness of the duck typing.

One might argues that everything in intentional programming is achievable with well-documented code (e.g., following good documentation). This is true, but this is the same of any compiler or generative programming – we are always compiling code down from a more intuitive format to a more obtuse format. In most older languages, we compiled languages down to assembly. Newer languages often transpile down to a reasonably-low level language such as C. In our case, we aim to transpile down to arbitrary languages, as well as drivers are written for that language.

A related concern is that assessment of this method is highly based on usability. While there are certain specific and measurable desired features (see features section, below), the effects of these features are unknown.

The concerns can be summarized with the following questions:

- How can we ensure that we produce correct code when loosening language formality? (And without introducing much complexity?) In other words, how can we decrease the formality of language while maintaining a precise specification?

- How can we enforce that the intent implementations match the users' true intentions?

- How can we assess this system (to test whether it actually aids students' learning process?)

## 2 Related work

Several visual programming languages exist to help aid programming education, especially for younger students. Notable examples include MIT's Scratch [citation] and App Inventor languages [citation], which are visual block-based programming languages. With block-based programming languages, syntax errors are scarce if not impossible. These languages are somewhat limited in their capabilities, and thus are not well-suited to "real-world" programming [citation].

The Hedy programming language [citation] has the intent of reducing cognitive load on students, citing the fact that the short term memory can be overwhelmed quickly by unimportant things like syntax. It brands itself as "gradual programming": there are progressively-difficult "levels" of the Hedy language, each adding new concepts, syntax, and functions on top of the previous level. This reduces the cognitive load at each step, and allows the user to proceed through the increasing difficulty like a leveled game. Hedy also notes that most programming education recommends trial-and-error learning, which may be discouraging to students who may prefer a more intuitive or guided approach.

There are many programming education websites currently that teach elementary programming in a rather standard manner – by teaching new skills gradually and having students replicate these in exercises. Khan Academy, Udemy, and Codecademy are examples of this. Another example Exercism, which focused mostly on providing mentored exercises without the standard learning track (the learning track was only added recently), which focuses more on a trial-and-error approach. Websites like these are a common entry point for self-taught programmers, but they face the problems discussed with standard programming education.

Intentional programming and metaprogramming are related to a number of other software-engineering practices and principles, such as (but not limited to): generative programming [citation], literate programming [citation], test-driven development (TDD) [citation], top-down design [citation], and language-oriented programming [citation]. It heavily involves the use of a domain-specific language (DSL) in its representation; Lisp is historically known for its metaprogramming capabilities [citation] and used as the basis for this project's DSL.

Domain-oriented programming is probably the closest idea to this that has been put into practice. See JetBrains MPS (Meta Programming System) – it uses the same jargon as Simonyi, such as "intentions" and "projections."

TODO: cite works talking about the faults in programming education

TODO: talk about error messages in programming languages

## 3 Features and functional requirements

This section is split into two subsections. General features indicate features that embody some general software engineering principle (and are thus difficult to measure directly). These will be argued intuitively. The other section includes measurable features.

### 3.1 General features

TODO: desired features: write something about intents?

#### 3.1.1 Intents can express any language feature

TODO: intents can thus model language "genes", such as scoping, types, etc., even when it doesn't have it itself; we're dealing with syntactic modifications at this point, which may require some knowledge outside of our system (which the target language compiler should be able to detect)

#### 3.1.2 Intents separate "intent" from implementation

TODO: intents may be implemented as one of several language features, e.g., control flow operators, special keywords, procedures, macros (code replacement), etc.

#### 3.1.3 Automatic test framework

TODO: automatically generate unit tests for each intent based on its api

TODO: automatically generate assertions for its inputs based on its api

#### 3.1.4 Gradual programming with default options

TODO: default values on intents (similar to default parameter values) used to ease use of intents

### 3.1.5 Language interoperability

TODO: specialized intents for I/O between programs written in different languages (a binary exchange fixpoint)

TODO: allow multiple target languages in the same project

### 3.1.6 Learning ecosystem

TODO: learn by example (a.o.t. learn by error), where examples all follow the intentional programming paradigm

TODO: package-like system with centralized system of intents

TODO: need some concept of fully-qualified intent names, and versioning; one way is to make each intent version have a different name so that a particular version is fixed

## 3.2 Measurable features

### 3.2.1 Single hierarchical namespace

TODO: only namespace for most use cases (except for those writing the low-level drivers) is the intent namespace – can group intents using a hierarchy and use fully-qualified names; simplifies model for the programmer

### 3.2.2 Single-file project, multiple views

TODO: replaces the traditional file/directory structure of a code project, which may confuse beginners

TODO: users can open several intent-level "views" into the project; see also: zooming capability

### 3.2.3 Target compilation information is associated with intents

TODO: if errors/messages are encountered during target code compilation, the generated code is linked to the intent

### 3.2.4 Literate programming

TODO: everything is annotated (enforced)

TODO: the API is annotated

### 3.2.5 Plain-english syntax and language-oriented design

TODO: talk about Lisp; however Lisp is hard to use (only prefix operators), so we want a more flexible syntax. Each intent is a small arbitrary (English) phrase with placeholders

TODO: not limited to english – easily translatable to other languages (most languages' syntaxes are English-only)

TODO: talk about recall [citation about recall?]

TODO: for teaching purposes, impedes cheating (since everyone's self-defined language is slightly different)

### 3.2.6 Structural correctness

TODO: scratch

TODO: paredit

TODO: we are basically working on the AST level

### 3.2.7 Structural completeness

TODO: top-down design

### 3.2.8 Syntactically-correct generated target code

TODO: cannot always be semantically correct due to additional constraints

TODO: structural correctness and completeness in the IR (correct AST) should translate to syntactically-correct code (but not semantic correctness)

### 3.2.9 Semantic "zoom" (code folding)

TODO: reduce cognitive overload

TODO: semantic (intent-based) a.o.t. structural (block-based) code folding

TODO: folded code can be replaced with its high-level description

### 3.2.10 Semantic error reporting

TODO: intents have access to the intent-stack, can be used to generate semantic (intent-based) exception messages rather than purely structural messages (call stack dumps)

### 3.2.11 Text-based intermediate representation

TODO: useful for text-based versioning/collaboration systems (e.g., Git)

TODO: easy to read, based on Lisp

### 3.2.12 Target language static analyses

TODO: implement the language server protocol (LSP) client

TODO: editing is much more feasible if we see the errors that are generated as we are editing, rather than when compiling the target code

## 4 (Proposed) Architecture

TODO: give a rundown of how this may be built

## 5 Analysis

TODO: talk about potential methods of evaluating this method: e.g., usability studies for UX

TODO: performance analysis: see if this generates correct code, whether it is "good quality code," and if the code generation is fast

TODO: talk about whether this matches the guarantees (in the features)

## 6 Concerns

TODO: what if a user edits the text file and leaves it in an inconsistent state?

TODO: going from a source file to an IR representation?