λ-Calculus and Functional Programming

A Bridge Between Mathematics and Programming

Xiang Li

University of California, Irvine

Undergraduate of Mathematics

2022

Table of Contents

0. Motivation

My effort is trying to raise **significant** framing questions before I answering them with **supportive** reading texts and other evidence. Each question is raised based upon my personal coding experience and reading materials. If possible, I want to truly master mathematics while I conducting historical research in this field.

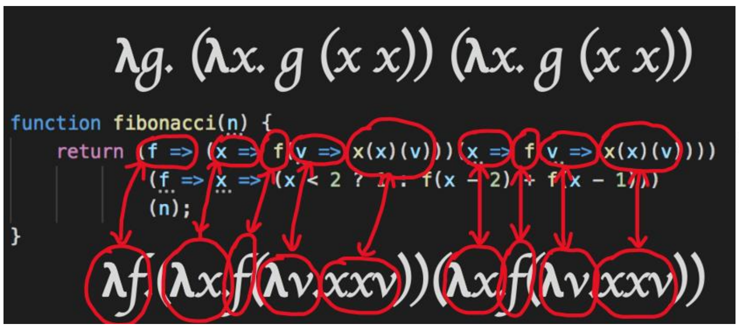
The main method for conducting research is **to feed key terms (#λ-Calculus #Functional Programming #LISP) to search engine (Google)**. This is what I am good at rather than **developing an original thesis independently**. But copy-and-paste is prohibited, I would like to use **diagrams** when I feel that further discussion is hard to be paraphrased. For instance, I prefer to use notations of **directed graphs** from graph theory to denote control flow statements.

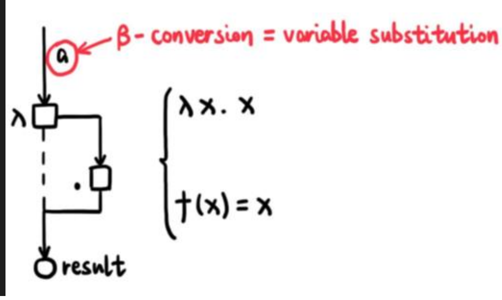
My poor discipline of proof-based math cannot support conducting any pure mathematical research, while I have to get my bachelor degree of math. **Notations do not support decision making**, therefore I give up asking any question about **"showing work"** (feedback from my TA of Math2A Fernando Quintino), and I focus more on reasoning forward.

This is my first time trying to practice math history research. My expectation is simple: it can be graded as A rather than C.

1. Alonzo Church and λ-Calculus

Without definition of data types, λ-Calculus only contains three components: **variables**, **functions**, and **applications**. It is easier than the syntax of structural programming since any statements except functions have been removed. As well as structural programming, λ functions support recursion as its only control flow structure.





https://medium.com/swlh/y-and-z-combinators-in-javascript-lambda-calculus-with-real-code-31f25be934ec

(From Upenn CIS511-c-S15)

Sets and **functions** are fundamental concepts which are used to define various mathematical concepts in proof-based math **(MATH 13)**; however, Church removed sets in λ-Calculus in order to establish a type-free language altering to set theory. In the end, the plan had been proved to be failed because of the **Russell's paradox**.

But it does not mean λ-Calculus is thoroughly failed. Church successfully turned λ-Calculus into a system for defining **computability**. Kleene proved in 1936 that <u>all the computable functions (recursive functions) in the sense of Herbrand and Gödel are definable in the λ-Calculus</u>, showing that it has universal computing power.

In 1937, Turing proved that Turing machines compute the same class of computable functions. All the total computable functions (total **recursive** functions) are definable in the λ-Calculus.

(From Dana Scott's *λ-Calculus Then & Now*)

Entscheidungsproblem = To determine whether a formula of the **first-order** predicate calculus is **provable** or not.

① Church's solution:

Theorem. Only a finite number of axioms are needed to define a **non-recursive** set of integers.

After the solution of Hilbert's 10^{th} problem, the applicability of this theory became even easier.

② Turing's solution:

Theorem. Only a finite number of axioms are needed to define the **Universal Turing Machine**.

③ Minskyizing the UTS:

Starting with Claude Shannon in 1956, many people – often in competition with Marvin Minsky – proposed very small UTMs. But, axiomatically, they do not require as many axioms as Turing did.

④ Post-Markov's Solution:

The basic idea of Post (1943) was that a logistic system is simply a set of rules specifying how to change one string of symbols (antecedent) into another string of symbols (consequent). This leads to:

The Word Problem for semigroups

⑤ Schönfinkel-Curry's Solution:

Schönfinkel in 1924 and then Curry in 1929, both at Göttingen, began the study of combinators, which were quickly connected with Church's λ-calculus of 1932.

The only problem with this theory is that you either need models or something like the Church-Rosser Theorem to know it is **consistent**. A **weaker** theory of deterministic reduction can be given **a fairly short axiomatization** and then be proved consistent by much **simpler means**.

2. John McCarthy and LISP

(From Dana Scott's *λ-Calculus Then & Now*)

LISP History according to McCarthy's memory in 1978. Presented at the ACM SIGPLAN History of Programming

Languages Conference, June 1-3, 1978. It was published in History of Programming Languages, edited by Richard Wexelblat, Academic Press 1981. Two quotations:

I spent the summer of 1958 at the IBM Information Research Department at the invitation of Nathaniel Rochester and chose **differentiating algebraic expressions** as a sample problem. It led to the following innovations beyond the FORTRAN List Processing Language:

... ...

(c) To **use functions as arguments** one needs a **notation** for functions, and it seemed natural to use the **λ-notation** of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions. Church used **higher-order functionals** instead of **using conditional expressions**. Conditional expressions are much more readily implemented on computers.

... ...

Logical completeness required that the notation used to express functions used as functional arguments be extended to provide for recursive functions, and the LABEL notation was invented by Nathaniel Rochester for that purpose. D.M.R. Park pointed out that LABEL was logically unnecessary since the result could be achieved using only $\boldsymbol{\lambda}$ – by a construction analogous to **Church's Y-operator**, albeit in a more complicated way.

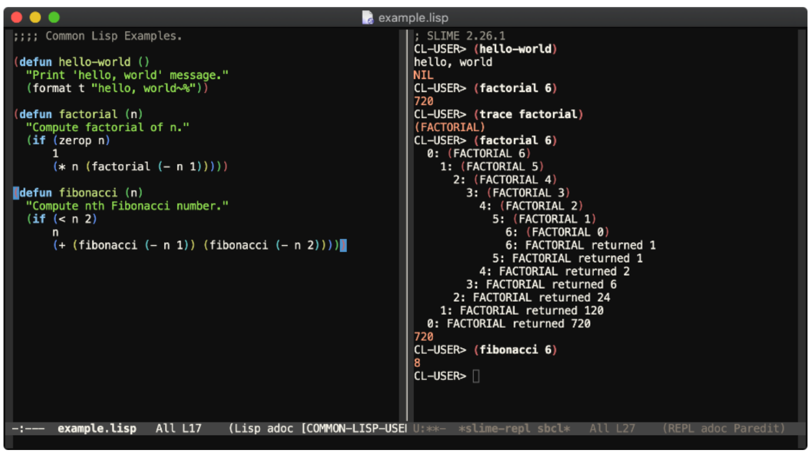
Other Key McCarthy publications:

1. *Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I)*. The original paper on LISP from **CACM**, April 1960. Part II, which never appeared, was to have had some Lisp programs for algebraic computation.

2. *A Basis for a Mathematical Theory of Computation*, first given in 1961, was published by North-Holland in 1963 in **Computer Programming and Formal Systems**, edited by P. Braffort and D. Hirschberg.

3. *Toward a Mathematical Science of Computation*, IFIPS 1962 extends the results of the previous paper. Perhaps the first mention and use of **abstract syntax**.

4. *Correctness of a Compiler for Arithmetic Expressions* with James Painter. May have been the first proof of **correctness of a compiler**. Abstract syntax and Lisp-style recursive definitions kept the paper short.



https://github.com/susam/emacs4cl

3. Edsger W. Dijkstra and Structured Programming

(#Dijkstra #Structured Programming #Goto #Böhm-Jacopini Theorem #software engineering)

Recursion is the central concept in computability; however, in **Böhm-Jacopini Theorem,** any program can be coded by only three types of control-flow statements: **sequence, selection, and repetition**. After that, in the letter *GoTo Statement Considered Harmful*, Dijkstra advocated that Goto statements should be removed since they are the source of program complexity, and structured programming should be introduced to software engineering.

Personally speaking, when I used to be trained to write Pascal code, the textbook said two taboos: Goto statements and recursion. Goto would bring unnecessary complexity, and recursion is costly. Even if tons of algorithms are implemented based upon recursion, for now, I am still trying to avoid using it.

Indeed, imagine system states and control-flow statements form **a connected graph**, then a program written by Goto statements and another version written by sequence-selection-repetition are topologically equivalent since they achieve the same function but in different text order.

Just a guess, **computability also relates to connectivity of state diagrams, but diagrams are more likely to be used for system design (UML, Universal Modeling Language) rather than mathematical objects waiting for being analyzed. (CS 163) (MATH 141)**

4. Early Era of AI

4.1 Dartmouth Workshop (https://en.wikipedia.org/wiki/Dartmouth_workshop)

In the early 1950s, there were various names for the field of "thinking machines": cybernetics, automata theory, and complex information processing.[4] The variety of names suggests the variety of conceptual orientations.

In 1955, John McCarthy, then a young Assistant Professor of Mathematics at Dartmouth College, decided to organize a group to clarify and develop ideas about thinking machines. He picked the name 'Artificial Intelligence' for the new field. He chose the name partly for its neutrality; avoiding a focus on narrow automata theory, and avoiding cybernetics which was heavily focused on analog feedback, as well as him potentially having to accept the assertive Norbert Wiener as guru or having to argue with him.[5]

In early 1955, McCarthy approached the Rockefeller Foundation to request funding for a summer seminar at Dartmouth for about 10 participants. In June, he and Claude Shannon, a founder of information theory then at Bell Labs, met with Robert Morison, Director of Biological and Medical Research to discuss the idea and possible funding, though Morison was unsure whether money would be made available for such a visionary project.[6]

On September 2, 1955, the project was formally proposed by McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon. The proposal is credited with introducing the term 'artificial intelligence'.

4.2 AI Winter (https://en.wikipedia.org/wiki/AI_winter)

In the history of artificial intelligence, an AI winter is a period of reduced funding and interest in artificial intelligence research. The term was coined by analogy to the idea of a nuclear winter. The field has experienced several hype cycles, followed by disappointment and criticism, followed by funding cuts, followed by renewed interest years or decades later.

The term first appeared in 1984 as the topic of a public debate at the annual meeting of AAAI (the called the "American Association of Artificial Intelligence"). It is a chain reaction that begins with pessimism in the AI community, followed by pessimism in the press, followed by a severe cutback in funding, followed by the end of serious research. At the meeting, Roger Schank and **Marvin Minsky**—two leading AI researchers who had survived the "winter" of the 1970s—warned the business community that enthusiasm for AI had spiraled out of control in the 1980s and that disappointment would certainly follow. Three years later, the billion-dollar AI industry began to collapse.

Hype is common in many emerging technologies, such as the railway mania or the dot-com bubble. The AI winter was a result of such hype, due to over-inflated promises by developers, unnaturally high expectations from end-users, and extensive promotion in the media. Despite the rise and fall of AI's reputation, it has continued to develop new and successful technologies. AI researcher Rodney Brooks would complain in 2002 that "there's this stupid myth out there that AI has failed, but AI is around you every second of the day." In 2005, Ray Kurzweil agreed: "Many observers still think that the AI winter was the end of the story and that nothing since has come of the AI field. Yet today many thousands of AI applications are deeply embedded in the infrastructure of every industry."

Enthusiasm and optimism about AI have generally increased since its low point in the early 1990s. Beginning about 2012, interest in artificial intelligence (and especially the sub-field of machine learning) from the research and corporate communities led to a dramatic increase in funding and investment.

4.3 Decision Support System (https://en.wikipedia.org/wiki/Decision_support_system)

The concept of decision support has evolved mainly from the theoretical studies of organizational decision making done at the Carnegie Institute of Technology during the late 1950s and early 1960s, and the implementation work done in the 1960s.[3] DSS became an area of research of its own in the middle of the 1970s, before gaining in intensity during the 1980s.

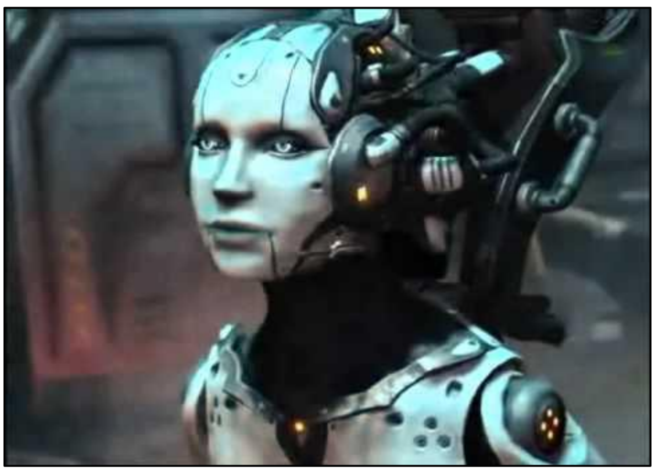
In the middle and late 1980s, executive information systems (EIS), group decision support systems (GDSS), and organizational decision support systems (ODSS) evolved from the single user and model-oriented DSS. According to Sol (1987),[4] the definition and scope of DSS have been migrating over the years: in the 1970s DSS was described as "a computer-based system to aid decision making"; in the late 1970s the DSS movement started focusing on "interactive computer-based systems which help decision-makers utilize data bases and models to solve ill-structured problems"; in the 1980s DSS should provide systems "using suitable and available technology to improve effectiveness of managerial and professional activities", and towards the end of 1980s DSS faced a new challenge towards the design of intelligent workstations.[4]

In 1987, Texas Instruments completed development of the Gate Assignment Display System (GADS) for United Airlines. This decision support system is credited with significantly reducing travel delays by aiding the management of ground operations at various airports, beginning with O'Hare International Airport in Chicago and Stapleton Airport in Denver Colorado.[5] Beginning in about 1990, data warehousing and on-line analytical processing (OLAP) began broadening the realm of DSS. As the turn of the millennium approached, new Web-based analytical applications were introduced.

DSS also have a weak connection to the user interface paradigm of hypertext. Both the University of Vermont PROMIS system (for medical decision making) and the Carnegie Mellon ZOG/KMS system (for military and business decision making) were decision support systems which also were major breakthroughs in user interface research. Furthermore, although hypertext researchers have generally been concerned with information overload, certain researchers, notably Douglas Engelbart, have been focused on decision makers in particular.

The advent of more and better reporting technologies has seen DSS start to emerge as a critical component of management design. Examples of this can be seen in the intense amount of discussion of DSS in the education environment.

Science Fiction Style



Adjutant in StarCraft II:

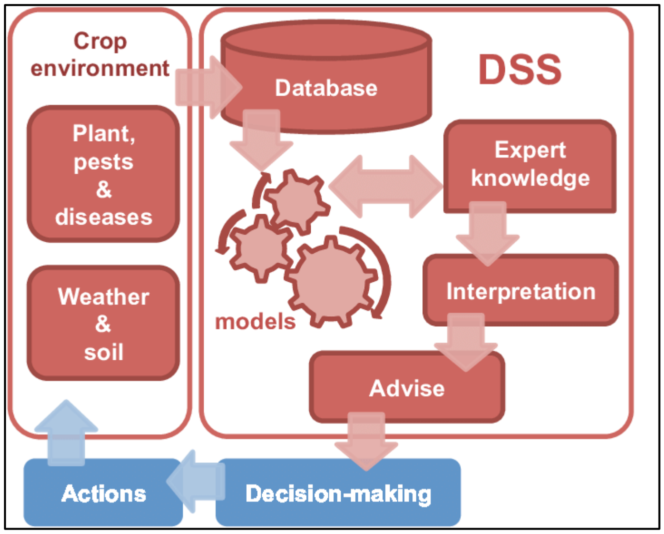
(1) Notify real-time info,

(2) analyze battlefield,

(3) and suggest potentially useful strategies

A possible implementation for DSS: AR, Augmented Reality

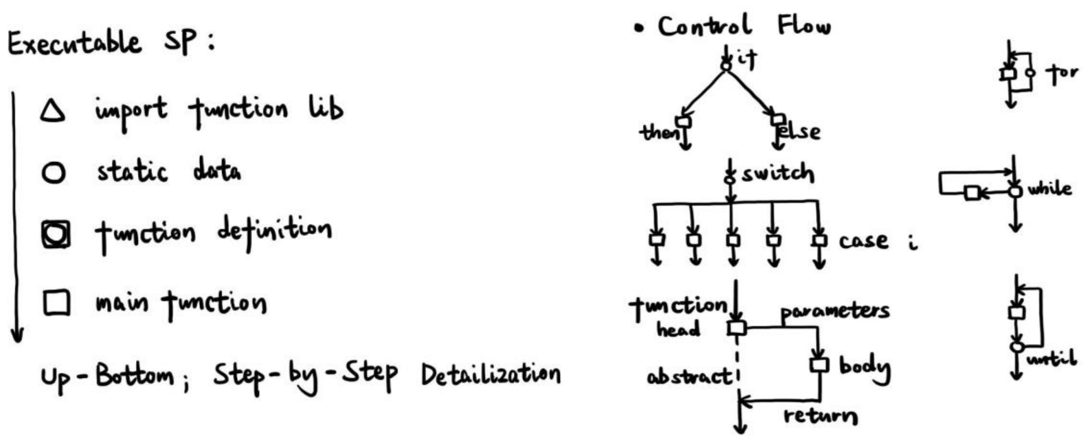
Real-world System



https://www.researchgate.net/figure/Scheme-of-an-innovative-Decision-Support-System-DSS-for-plant-disease-management-The_fig5_267262102

5. Multiple Paradigms

5.1 Executable Structured Programming



5.2 Executable Object-Oriented Programming

- Executable OOP:

  library = classes contributed by other coders

  □ class

  ○ = □ initialization

  ○.▽ visit attribute

  ○.△ call a method

Why OOP?

1. Code reuse
2. Easy to conduct data flow analysis
3. Simulation
4. Natural Semantics

5.3 Executable Functional Programming

① Functions = Value; ② Functions as methods of lists

Imperative vs. Functional (JavaScript)

https://en.wikipedia.org/wiki/Functional_programming

```
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let result = 0;
for (let i = 0; i < numList.length; i++)
        {
                if (numList[i] % 2 === 0)
                        {result += numList[i] * 10}
        }
```

```
const result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
.filter(n => n % 2 === 0)
.map(a => a * 10)
.reduce((a, b) => a + b);
```

5.4 Abelian Groups and Concurrent Programming **(MATH 120A)**

In concurrent programming, Instructions are **independent** rather than sequential. Indeed, independency means instructions can be parallel or sequential, and the order of instructions does not matter.

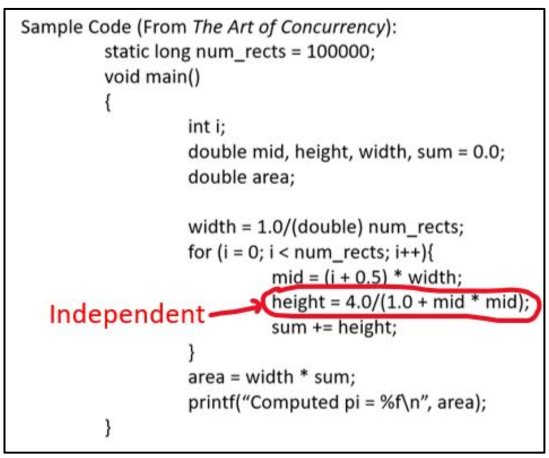
Let some independent instructions be elements of a finite instruction set. Then we say the elements are parallelable if the order of instructions does not matter. For instance, addition $(+)$ and multiplication $(\times)$ are commutative operators in any given abelian groups. $2+3=3+2=5 \text{ } and \text{ } 2\times 3=3\times 2=6$. Contrary to abelian operators, matrix multiplication does not obey commutativity, which means two matrices applied matrix multiplication are dependent and sequential.

$$Suppose\ A \in \mathbb{R}^{m\times n}\ and\ B \in \mathbb{R}^{n\times m}, A\ and\ B\ are\ not\ identity\ matrices,\ n, m \in \mathbb{N}.$$

$$Then\ AB \neq BA.$$

Obviously, matrix multiplication cannot be parallel since order of computation matters.

Problem. Compute π by coding the formula $\pi = \int_0^1 \frac{4}{1+x^2}\,dx$



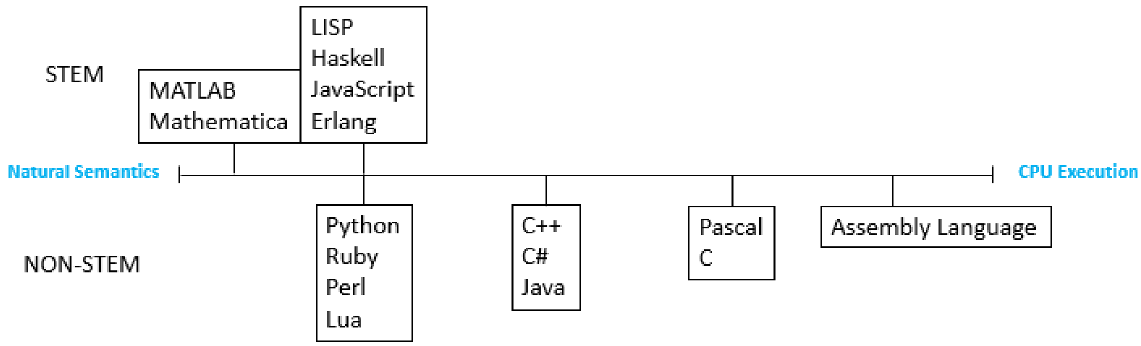
```
Sample Code (From The Art of Concurrency):
        static long num_rects = 100000;
        void main()
        {
                int i;
                double mid, height, width, sum = 0.0;
                double area;

                width = 1.0/(double) num_rects;
                for (i = 0; i < num_rects; i++){
                        mid = (i + 0.5) * width;
Independent             height = 4.0/(1.0 + mid * mid);
                        sum += height;
                }
                area = width * sum;
                printf("Computed pi = %f\n", area);
        }
```

Q: To what extent will programming paradigm affect the efficiency of coding and the readability of code?

When I was a Pascal user, I thought Delphi or C++ are redundant. Extra work of defining classes was annoying since in Pascal I was not required to do this kind of work. Later, when I got started to learn Swift in Swift Playgrounds, I firstly knew the meaning of classes and objects: **coding like using natural language, doing simulation, and reusing code templates**. By importing packages, I was allowed to code few lines rather than a long piece of class. After that, when I studied Machine Learning in Python, I realized that **dataflow modeling was naturally related to OOP**. A useful program must be able to manage data in a good manner.

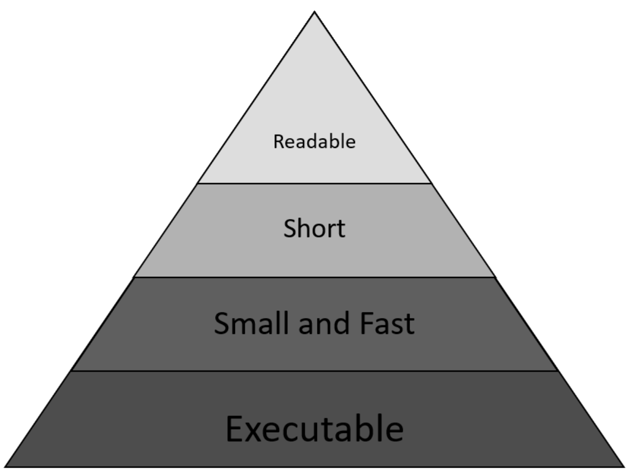
Q: Trade-off: natural semantics prior or efficiency of CPU prior?

A spectrum between natural languages and machine code is enough for evaluating programming paradigms:



6. Need for Verification: Test, Evaluation, or Proof?

6.1 The Pyramid of Coding Performance



·**Executable:** Code should be always runnable whatever the result is correct or not. If incorrect, it can be easily tested by using print statements or a debugger rather than stuck on compiling errors.

·**Small and Fast:** In order to succeed in informatics competition, a program which is able to generate correct result is not enough for players. Within a constraint of limited memory and time, players must optimize their code by using some tricky algorithms.

·**Short:** In the book *Short Coding*, Ozy showed his experience of writing short code, which is fully functional as well as a piece of long code doing the same work. Short code usually means less memory use for source code.

·**Readable:** Based upon Rainer Dömer, my **EECS 10** Professor teaching Introduction to C, readability counts credit in coding assignments. For some reason about teamwork, it is easy to maintain a piece of readable code rather than a non-readable one. Other coders do not need to read more documents in order to understand the purpose of programs.

6.2 Skeleton Key in Mastering Math (?)

·Core skills in training of mathematics:

1. Prove statements/theorems/algorithms;
2. Computation and Analysis;
3. Draw graphs

·Core skills in training of computer science:

1. Blind typing with ten fingers;
2. Solve problems by knowing the semantics of tools;
3. Evaluate performance by the product of multiple key indicators

I used to regard **connectivity** of graphs as the central idea of mathematics, because graphs are able to represent relations: formulas between parameters, theorems between properties, and transformation between functions. My work is to show the connectivity of intended graphs. However, college math I have learned does not meet this expectation. Professors only give me the setback of low grades based upon their "mature" rubrics. Merely every day I subconsciously curse professors and TAs in a place of nobody. A potentially useful approach to obtain a nice-looking grade, I think, is to ask professors some questions which I have already resolved, because professors are more likely to use my questions for examination.

7. A Timeline of λ-Calculus (From Dana Scott's *λ-Calculus: Then & Now*)

1870s

Begriffsschrift	**Frege** (1879)

1880s

What are numbers?	Dedekind (1888)
Number-theoretic axioms	**Peano** (1889) **(MATH 140A)**

1890s

Vorlesungen über die Algebra der Logik	Schröder (1890--1905)
Grundgesetze der Arithmetik	**Frege** (1893)
Formulario Mathematico	**Peano** (1895--1901)

Grundlagen der Geometrie	**Hilbert** (1899)

1890s

Diophantine problem	**Hilbert** (1900)
Russell's Paradox	Russell (1901)
Principles of Mathematics	Russell (1903)
Richard's Paradox	Richard (1905)
Theory of Types	Russell (1908)

1910s

Principia Mathematica	Whitehead-Russell (1910—12--13)
Calculus of relatives	Löwenheim (1915)

WW I---

1920s

Löwenheim-Skolem Theorem	Skolem (1920)
Propositional calculus completeness	Post (1921)
Monadic predicate calculus decidable	Behmann (1922)
Abstract proof rules	Hertz (1922)
Primitive recursive arithmetic	Skolem (1923)
Combinators	Schönfinkel (1924)
Function-based set theory	von Neumann (1925)
"Conceptual" undecidability	Finsler (1926)
Epsilon operator	Hilbert-Bernays (1927)
Combinators(again)	Curry (1927)
Ackermann function	Ackermann (1928)
Entscheidungsproblem	Hilbert-Ackermann (1928)
Abriss der Logistik & simple type theory	Carnap (1929)

1930s

Combinatory logic	Curry (1930-32)
Herbrand's Theorem	Herbrand (1930)
Completeness proof	Gödel (1930)
Partial consistency proof	Herbrand (1931)
Incompleteness	Gödel (1931)
Untyped λ-Calculus	**Church (1932-33-41)**
Studies of primitive recursion	Péter (1932-36)
Non-standard models	Skolem (1933)
Functionality in Combinatory Logic	Curry (1934)
Grundlagen der Mathematik	Hilbert-Bernays (1934-39)
Natural deduction	Gentzen (1934)
Number-theoretic consistency & ϵ_0-induction	Gentzen (1934)
Inconsistency of Church's System	Kleene-Rosser (1936)
Confluence theorem	Church-Rosser (1936)
Finite combinatory processes	Post (1936)
Turing machines	Turing (1936-37)
Recursive undecidability	Church-Turing (1936)
General recursive functions	Kleene (1936)
Further completeness proofs	Maltsev (1936)
Improving incompleteness theorems	Rosser (1936)
Fixed-point combinator	Turing (1937)
Computability and λ-definability	Turing (1937)

1940s

Simple type theory & λ-calculus	Church (1940)
Primitive recursive functionals	Gödel (1941-58)

WW II---

Recursive hierarchies	Kleene (1943)

Theory of categories	Eilenberg-Mac Lane (1945)
New completeness proofs	Henkin (1949-50)

1950s

Computing and intelligence	Turing (1950)
Rethinking combinators	Rosenbloom (1950)
IAS Computer (MANIAC)	von Neumann (1951)
Introduction to Metamathematics	Kleene (1952)
IBM 701	Thomas Watson, Jr. (1952)
Arithmetical predicates	Kleene (1955)
FORTRAN	Backus et al. (1956-57)
ALGOL 58	Bauer et al. (1958)
LISP	**McCarthy (1958)**
Combinatory Logic. Volume I.	Curry-Feys-Craig (1958)
Adjoint functors	Kan (1958)
Recursive functionals & quantifiers, I. & II.	Kleene (1959-63)
Countable functionals	Kleene-Kreisel (1959)

1960s

Recursive procedures	Dijkstra (1960) **(CS 161)**
ALGOL 60	Backus et al. (1960)
Elementary formal systems	Smullyan (1961)
Grothendieck topologies	M.Artin (1962)
Higher-type λ-definability	Kleene (1962)
Grothendieck topoi	Grothendieck et al. SGA 4 (1963-64-72)
CPL	Strachey, et al. (1963)
Functorial semantics	Lawvere (1963)
Continuations (1)	van Wijngaarden (1964)

Adjoint functors & triples	Eilenberg-Moore (1965)
·Cartesian closed categories·	Eilenberg-Kelly (1966)
ISWIM & SECD machine	Landin (1966)
CUCH & combinator programming	Böhm (1966)
New foundations of recursion theory	Platek (1966)
Normalization Theorem	Tait (1967)
AUTOMATH & dependent types	de Bruijn (1967)
Finite-type computable functionals	Gandy (1967)
ALGOL 68	van Wijngaarden (1968)
Normal-form discrimination	Böhm (1966)
Category of sets	Lawvere (1969)
Typed domain logic	Scott (1969-93)
Domain-theoretic λ-models	Scott (1969)
Formulae-as-types	Howard (1969-1980)
Adjointness in foundations	Lawvere (1969)

1970s

Continuations (2)	Mazurkiewicz (1970)
Continuations (3)	F. Lockwood Morris (1970)
Continuations (4)	Wadsworth (1970)
Categorical logic	Joyal (1970+)
Elementary topoi	Lawvere-Tierney (1970)
Denotational semantics	Scott-Strachey (1970)
Coherence in closed categories	Kelly (1971)
Quantifiers and sheaves	Lawvere (1971)
Martin-Löf type theory	Martin-Löf (1971)
System F, Fω	Girard (1971)
Logic for Computable Functions	Milner (1972)

From sheaves to logic	Reyes (1974)
Polymorphic λ-calculus	Reynolds (1974)
Call-by-name, call-by-value	Plotkin (1975)
Modeling Processes	Milner (1975)
SASL	Turner (1975)
Scheme	Sussman-Steele (1975-80)
Functional Programming & FP	Backus (1977)
First-order categorical logic	Makkai-Reyes (1977)
Edinburgh LCF	Milner et al. (1978)
Let-polymorphic type inference	Milner (1978)
Intersection types	Coppo-Dezani (1978)
ML	Milner et al. (1979)
*_Autonomous categories	Barr (1979)
Sheaves and logic	Fourman-Scott (1979)

1980s

Frege structures	Aczel (1980)
HOPE	Burstall et al. (1980)
The Lambda Calculus Book	Barendregt (1981-84)
Structural Operational Semantics	Plotkin (1981)
Effective Topos	Hyland (1982)
Dependent types & modularity	Burstall-Lampson (1984)
Locally CCC & type theory	Seely (1984)
Calculus of Constructions	Coquand-Huet (1985)
Bounded quantification	Cardelli-Wegner (1985)
NUPRL	Constable et al. (1986)
Higher-order categorical logic	Lambek-P.J. Scott (1986)
Cambridge LCF	Paulson (1987)

Linear logic	Girard et al. (1987-89)
HOL	Gordon (1988)
FORSYTHE	Reynolds (1988)
Proofs and Types	Girard et al. (1989)
Integrating logical & categorical types	Gray (1989)
Computational λ-calculus & **monads**	Moggi (1989)

1990s

HASKELL	Hudak-Hughes-Peyton Jones-Wadler (1990)
Higher-type recursion theory	Sacks (1990)
STANDARD ML	Milner, et al. (1990-97)
Lazy λ-calculus	Abramsky (1990)
Higher-order subtyping	Cardelli-Longo (1991)
Categories, Types and Structure	Asperti-Longo (1991)
STANDARD ML of NJ	MacQueen-Appel (1991-98)
QUEST	Cardelli (1991)
Edinburgh LF	Harper, et al. (1992)
Pi-Calculus	Milner-Parrow-Walker (1992)
Categorical combinators	Curien (1993)
Translucent types & modular	Harper-Lillibridge (1994)
Full abstraction for PCF	Hyland-Ong/Abramsky, et al. (1995)
Algebraic set theory	Joyal-Moerdijk (1995)
Object Calculus	Abadi-Cardelli (1996)
Typed intermediate languages	Tarditi, Morrisett, et al. (1996)
Proof-carrying code	Necula-Lee (1996)
Computability and totality in domains	Berger (1997)
Typed assembly language	Morrisett, et al. (1998)
Type theory via exact categories	Birkedal, et al. (1998)

Categorification	Baez (1998)

The New Millennium

Predicative topos	Moerdijk-Palmgren (2000)
Sketches of an Elephant	Johnstone (2002+)
Differential λ-calculus	Ehrhard/Regnier (2003)
Modular Structural Operational Semantics	Mosses (2004)
A λ-calculus for real analysis	Taylor (2005+)
Homotopy type theory	Awodey-Warren (2006)
Univalence axiom	Voevodsky (2006+)
The safe λ-calculus	Ong, et al. (2007)
Higher topos theory	Lurie (2009)
Functional Reactive Programming	Hudak, et al. (2010)

Reference

Minsky, Marvin, *The Society of Mind*

Church, Alonzo, *An Unsolvable Problem in Elementary Number Theory*

Church, Alonzo, *A Note on the Entscheidungsproblem*

Turing, Alan, *On Computable Numbers with an Application to the Entscheidungsproblem*

Turing, Alan, *Computability and λ-definability*

Ozy, *Short Coding*

Seibel, Peter, *Coders at Work*

Graham, Paul, *Hackers and Painters*

Sebesta, Robert, *Concepts of Programming Languages* **(CS 141)**

Scott, Dana, *λ-Calculus: Then & Now*

Rojas, Raul, *A Tutorial Introduction to the Lambda Calculus* **(iCS 33)**

Breshears, Clay, *The Art of Concurrency*

Dijkstra, Edsger, *GoTo Statement Considered Harmful*

Upenn, CIS 511-c-s15