Constructionist Design in Compiler Education: A Critical Evaluation of CS 327

Pranav Joshi

2 Nov, 2025

Learning Goals

Domain Background

Programming languages (PL) such as Python, C, Java, etc. are high-level languages (HLL). Code written in an HLL is either interpreted and executed immediately (as with JavaScript) or is converted to a lower-level language consisting of instructions called bytecode (for example, Python bytecode written in `..pyc` files). The bytecode is either executed by a virtual machine (VM) or by a real device, in which case it is called machine code, written according to the Instruction Set Architecture (for example, MIPS32) that the device uses.

A compiler is a stand-alone program, usually written in machine code, that converts code written in an HLL into bytecode. An interpreter is also a stand-alone program that converts and executes code in an HLL without conversion to bytecode. A virtual machine is another stand-alone program which executes a low-level language rather than an HLL.

Learning how to create interpreters and eventually compilers for programming languages allows programmers to create their own custom languages. While the set of possible PLs that can be interpreted is very large, the PLs that can be "sanely" compiled are only a subset of this space, and the PLs that can be used for general-purpose tasks and provide intuitive and necessary features are even smaller. Thus, PL design is an important task, and the theories used to design "good" PLs are collectively known as PL theory.

Course Context

This course, CS 327, is a CSE elective that falls into the computer systems basket and is aimed at students who plan to pursue careers in industry, where understanding compiler design and systems programming is increasingly valuable.

Domain Knowledge:

- Understanding of basic PL theory
- Detailed knowledge of interpreter and compiler design
- Understanding of VM and bytecode
- Practical ability to write code for interpreters and compilers in a HLL

Control Strategies and Meta-Cognitive Skills:

- Teamwork and efficient distribution of work to accelerate product development
- Time management
- Managing the trade-off between usability/practicality and creative implementation

Overarching Career-Oriented Goal:

For a course on compilers in an institute where the majority of students pursue industry careers, a primary goal should be enabling students to not only understand compiler concepts but to actually take on compiler-related

tasks in their future careers and possess the genuine competence needed to finish them successfully. To accomplish this, students must develop two prerequisites for engagement: a genuine sense of competence and authentic interest in the domain; not merely achievement of pass-marks.

Thus, the core metrics that the students should improve on are:

- Actual competence (the ability to design and implement compilers)
- Sense of competence (confidence in ability to work on compilers)
- Genuine interest (intrinsic motivation beyond course grades)

Learning Design and Features

Team Formation and Community Exemplars

On the first day of the semester, students were organized into teams and shown the PL designs built by teams in the previous year, providing concrete models of what successful final products could look like.

Assessment Weightage and Incentive Structure

The grading scheme heavily emphasized practical work over theoretical testing. Examinations received only 20% of the final grade, programming assignments received 30%, and the course project received the remainder (50%). This weightage fundamentally shaped student effort allocation and signaled that practical competence and artifact creation were more valued than theoretical knowledge retention alone.

Lecture Structure and Opportunistic Knowledge Introduction

Every week, students attended 2 lectures and 1 lab session. The lectures served dual purposes: they functioned both as doubt-resolution sessions where ongoing compiler work was discussed and refined, and as moments for introducing new concepts precisely when they became useful to students' current work. There was no grading based on lecture attendance; students were only expected to stay current with concepts introduced during lectures.

All course materials such as slides, notes, code, and relevant links were written on a page in a GitHub repository. At the top of the page, the professor explicitly stated:

> There is no textbook. Follow the lectures. Code a lot. Read necessary theory.

Worked examples

The code and pseudocode served as worked examples along with demonstrations done during lectures. However, we were not "spoon-fed"; we still had to modify and adapt these examples to solve our specific project problems.

On-Demand Scaffolding Through Just-In-Time Concept Introduction

Concepts were introduced to students almost precisely when they needed them, rather than in advance. For example, the cactus-stack data structure used for implementing function closures, was taught only after students had already begun implementing functions in their projects. This just-in-time scaffolding prevented cognitive overload while maintaining authentic need for the knowledge being taught.

This level of precision was possible because the students' progress was monitored by the teaching assistants in the labs.

Structuring Through Sequential Implementation Pipeline

The course imposed a logical sequence on project work, designed to mirror the conceptual dependencies within compiler design itself:

1. Basic language design and parser design
2. Interpreter implementation
3. Iterative improvement of both design and interpreter (allowing mastery of interpreter writing before moving to code generation)
4. Bytecode generation
5. VM implementation (forming an iterative cycle since bytecode design is tightly coupled with VM implementation)

This structure prevented chaotic, simultaneous attempts at multiple implementation strategies. Instead, students focused intensively on one stage at a time, building competence sequentially rather than being overwhelmed by the full scope simultaneously.

One-on-One Monitoring and Adaptive Feedback Through Lab Sessions

Lab sessions featured one-on-one discussions between students and TAs. These sessions were explicitly framed as monitoring scaffolds; forums for providing personalized guidance and feedback rather than for grading performance. This qualitative monitoring allowed the professor to provide adaptive feedback based on the teams' progress and challenges, similar to scrum meetings in industry or research lab sessions.

Learning Through Peer Examination

Student teams examined each other's PL designs and compiler approaches, using insights from peers to refine their own work. This created a some-what active community of practice where students were both learners and resources for one another.

Fair Mark Distribution and Social Accountability

The course implemented an innovative solution to the "freeloader problem" common in group projects: rather than assigning identical marks to all team members, the total marks for the project were given to the team, and team members negotiated and justified the distribution of marks to the professor. Critically, this rule was announced only at the end of the semester; had it been explicit from the start, it would have created unhealthy competition and discouraged team formation.

Learning Theories or Pedagogical Models Used

Constructionism: Learning Through Artifact Creation

Constructionism, developed by Seymour Papert, posits that learning occurs most powerfully when learners design, build, and refine tangible, meaningful artifacts in the world.

Evidence of constructionist learning can be seen from the course outcomes: by the end of the semester, I had learned nearly every concept required for the final written exam while referring to no written sources and attending only about half the lectures. The learning happened primarily through the project and assignments, not through instruction.

Learning followed a bricolage model, characteristic of authentic programming work. I would design or implement a feature, observe the restrictions and benefits that the feature provided after implementation, and based on that observation, either add another feature or improve the existing implementation.

Cognitive Apprenticeship: Guided Development of Complex Skills

Cognitive apprenticeship, articulated by Collins, Brown, and Newman, describes how learners acquire complex skills through observation, guided practice, and progressively increased responsibility within an authentic domain. The compilers course exhibits these components of cognitive apprenticeship:

- Global Before Local (Modeling at Scale) : On the first day, students were shown complete PL implementations from previous years.
- Modeling Through Worked Examples : Code and pseudocode for basic tasks
- Scaffolding
  - Timely adaptive help through monitoring in lab and concept introduction based on that
  - Structuring via the implementation pipeline (basic design, interpreter, bytecode, VM cycle)
- Articulation Through Lab Sessions

Cultural Learning Pathways: Identity, Belonging, and Long-Term Trajectories

Beyond immediate course knowledge, both constructionist and apprenticeship approaches operate within the broader framework of cultural learning pathways. This perspective views the course not as an isolated event but as an episode within each student's trajectory toward becoming a competent compiler engineer or systems programmer. In discussions with peers, I discovered diverse motivations for taking the course, each representing different entry points into the domain:

- Industry relevance: Compilers and computer systems skills are in-demand in the software development market.
- Career incentives: Some classmates had secured internships at companies like Qualcomm with roles centered on compiler work.
- Research interest: One teammate was conducting research in PL theory; this course was a natural continuation.
- Challenge and growth: Some students (including myself) had heard the course was challenging and saw taking it as central to becoming a strong computer engineer.
- Theory into practice: A classmate with strong theoretical CS background wanted to see her knowledge of computation theory applied in practice.

The course design respects and cultivates these diverse entry points. By producing genuine competence and belonging, not just grades, it creates positive episodes that shape long-term trajectories. My own experience illustrates this: the course's success led me to ideate a new independent project (creating a parser for Oracle SQL).

Assessment of Learning Theory or Pedagogical Model Appropriateness

The theories employed in the design of this course have these benefits :

1. Constructionism supports development of style and a sense of ownership which leads to **idenity formation**.
2. Cognitive apprenticeship supports both **domain knowledge** and **meta-cognitive skills** through scaffolded, guided practice leading to practical competency **without overwhelming** the students as Sweller warns against.
3. The course creating a positive episode moves the students upwards in the cutural learning path by giving them a **sense of competence**. This "good episode" occurs mainly because of the constructionist approach which is known to be "fun" and often leaves the creator proud of his creation.

On the other hand, a comparative analysis of alternative pedagogical paradigms reveals that while many excel in some dimensions, only the combination of constructionism and cognitive apprenticeship produces all three critical outcomes for the over-arching career-oriented goals simultaneously (competence, sense of competence, and idenity formation) :

- Behaviorism and traditional instructionism produce procedural competence but lack motivation and ownership. Students can execute procedures but don't develop identity or genuine interest.
- Cognitivism based instructionism supports mental model development but can remain abstract and disconnected from authentic practice, limiting transfer and motivation.
- Discovery learning can produce motivation but risks leaving critical conceptual gaps, especially in a domain as complex as compilers, thus not preparing the students for future.
- Problem-based learning engages students but often involves episodic problems rather than sustained artifact development, limiting the sense of ownership and long-term trajectory effects that come from building something persistent.
- Situated learning can overwhelm students with authentic complexity before they have sufficient schemas as Sweller warns about.
- Cognitive apprenticeship alone, while effective for competence development, doesn't necessarily produce the sense of ownership and identity from having built something personally meaningful.

Evaluation of Learning Design

Merits

Industry-level knowledge without abstraction overload Before taking this course, I had already studied basic compiler theory. But during the course, I found little use for most of that theoretical knowledge. The course operates on the implementable subset of PL and compiler theory, not the abstract formalism. This prevents cognitive overload from unnecessary theory. In contrast, the traditional format introduces full compiler theory, which many students find overwhelming and abstract.

Meaningfulness of knowledge throughout the learning process In this course, knowledge was learned because it was needed to solve authentic problems. I was able to learn nearly every concept required for the final exam purely through working on the project under guidance of the professor, TAs, and peers. As a result, I never felt the need to ask "why am I learning this?". It never felt like study. In contrast, students in the traditional lecture-based format often experience knowledge as abstract and disconnected from practice, reducing motivation.

Excellent sequencing and scaffolding In many other CS courses, students receive vague project descriptions and must independently navigate literature review, design decisions, and implementation. This often leads to overwhelm, shortcuts, plagiarism, and failure to develop confidence. This course provided careful scaffolding at each stage; students were not dropped into chaos.

Alignment between assessment practices and domain practices Exams, assignments, and projects all relied on the same underlying domain knowledge, with minimal emphasis on exam-specific heuristics or strategies. Success in the project strongly predicted success on the exam, creating intrinsic motivation rather than forcing students to juggle disconnected skill sets.

Appropriate cognitive load calibration As cognitive load theory (CLT) demonstrates, task difficulty must match learner schemas. Many CS course projects impose heavy cognitive load because students lack necessary prerequisite schemas and receive insufficient guidance. This course calibrated task difficulty appropriately for beginners; we were challenged but not overwhelmed.

Meaningful lab sessions The term "lab" is often used synonymously with "test", namely sessions where disconnected skills are graded. This course's labs functioned more like industry scrum meetings or research lab sessions: qualitative monitoring of progress for adaptive feedback rather than grading disconnected skills. This alignment between the "exam" practice (activity) and the "lab" practice enhances transfer and reduces load.

Fair contribution and positive team dynamics The mark-splitting approach solved the free-rider problem while preserving motivation for teamwork. A classmate who had done minimal work was upset when this rule was announced; I, having done roughly 70% of the work, was satisfied. Critically, this rule was announced only at semester's end. Had it been explicit from the start, it would have created unhealthy competition and discouraged team formation.

Design Gaps

The Underemployed Worker Problem As already discusses, the course solved the freeloader problem well enough.

However, the course exhibited one significant failure mode: the "underemployed worker issue." Students perceived as less competent by their team were not assigned challenging tasks and consequently contributed less, developed lower perceived competence, and potentially suffered negative learning episodes.

This creates a negative feedback loop: initial differences in perceived competence, however small or arbitrary, become reinforced through differential task assignment, preventing less confident students from developing the competence and confidence that the course aimed to provide.

The Lack of Breadth Problem The constructionist approach is excellent for practical competency but not so much for gaining a wider range of knowledge. The breadth of material covered was narrower than traditional courses. For example, recursive descent parsing was thoroughly taught since it was used a lot, but alternatives such as LR or RR parsing were never touched.

Redesign Features

Pairing Old-Timers with Newcomers for Inclusive Mentorship

I was fortunate to have a team-mate who was an "old-timer" in programming language theory and provided timely guidance. Many teams were not as fortunate.

To address this, an explicit system pairing an old-timer with each new-timer could be introduced. The old-timer's role would focus on good PL and compiler design principles rather than detailed coding or menial tasks, allowing the newcomer to receive personalized one-on-one feedback.

Critically, roles must not be explicitly assigned to preserve newcomer agency and allow respect to be earned naturally.

Linking Assignments to Project Timeline to Maintain Instrumentality

While the assignments were helpful in exploring new approaches, some felt abstract and disconnected from the project.

I suggest sequencing assignments to directly precede the project topics they support. For example, assignments on parsing would be scheduled immediately before parser implementation begins.

This strengthens the principle of instrumentality by ensuring all knowledge learned is immediately applicable.

Distributed Implementation of Compiler Pipeline Stages

Each new-timer should implement a different approach or component for each pipeline stage. For example, one student could implement a register-based bytecode VM while another implements a stack-based VM. Students can be graded with some weightage A for only their own implemented approaches and other weightage B using the mark-split method for the final optimum compiler that the team will present, such that $A + B = C$ where C is the total weightage for the project component (currently 50%). This avoids the free-rider problem.

All team members should explain their own approach and the approaches of other teammates (randomly picked) to the TAs during lab meetings. There can be grading for this at the team level which will force the students to learn what the team-mates have implemented.

Benefits include:

- Easier distribution of workload resolving the under-employed worker problem
- Exposure to a wider range of concepts and design trade-offs resolving the Lack of Breadth problem
- Opportunities for rich articulation and communication among team members, allowing for reflection and exploration (from cognitive apprenticeship)

"See Also" Literature to Address Breadth Without Overload

- Introduce "see also" sections pointing to external literature covering these omitted methods
- Keep these optional to preserve instrumentality. Students will be motivated to read these either as they face implementation choices requiring each method's understanding or due to curiosity, and will undergo exploration (from cognitive apprenticeship).
- The final exam should only include concepts covered by approaches implemented by teams
- Additionally, students can be asked to justify their chosen methods in a brief assignment (around 3% weightage), encouraging lightweight engagement with broader literature despite some initial resistance