Final Assignment

I will be writing about the compilers course (CS 327) I attended the last semester.

This is a CSE elective that falls into the computer systems basket.

Domain back-ground

Programming languages (PL) such as Python, C, Java, etc. are high level languages (HLL). Code written in a HLL is either interpreted and executed right away (such as for JavaScript) or is converted to a lower level language made of instruction 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 (MC) written according to the Instruction Set Architecture (for example MIPS32) that the device (for example, Nintendo) uses.

A compiler (for example `gcc`) is a stand-alone (dependency free) program written, usually written in machine code that converts code written in a HLL into byte-code.

An interpreter is also a stand-alone program that converts executes the code in the HLL without conversion to byte-code.

A virtual machine is another stand-alone program which executes a Low Level Language (LLL) rather than a HLL.

Learning how to create interpreters and eventually compilers for PLs allows a programmer to create his own custom PLs.

While the set of possible PLs that are possible to interpret is very big, the PLs that can be “sanely” compiled are only a subset of the full space and the PLs that can be used for general purpose tasks and provide intuitive and necessary features are even less. Thus, PL design is an important task. The theories used to design “good” PLs is called PL theory.

Learning goals of Course

The main domain knowledge that was to be learned is

- basic PL theory
- Interpreters and compiler design (in detail)
- virtual machines
- writing code for compilers in a HLL

The control strategies to be learnt were

- Teamwork and efficient distribution of work that would allow for quicker development of the product
- Time management
- Managing the trade-off between usability/practicality and creativity

Apart from that, the one of the main goals of a course on Compilers in an institute where the majority of students go towards industry should be to *enable* the students to both actually take on compiler related tasks in their careers and have the competence needed to finish them.

Design features of the course

On the first day, we were made to create teams and were shown the PLs built by the teams in the last year.

The weightage of examinations for the final grade was only 20%, with programming assignments getting another 30% and the course project getting most importance.

Every week, we had 2 lectures and one lab. The lectures were partly doubt sessions and discussions on the developing compilers and were partly used to introduce new concepts at the moments when they might be useful. There was no grading based on attendance and we were only expected to be up-to-date with all the concepts that were introduced in the lectures. There was a common page on a GitHub repository where any written material created (slides, notes, etc.) by the proffessor and relevant code and links were posted. At the start of that page this was written :

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

Still, a “good textbook” was mentioned with the disclaimer that the professor disagrees with some parts from it, again reminding us that “there is no textbook”.

The development (of the compiler) of the teams was monitored in the labs by the TAs in one-on-one discussions which were mainly aimed for guidance and feedback (monitoring scaffolds).

We also had an active community. We would look at each other's PLs and approaches for the compiler and improve our own design based on that.

Learning Theories

The learning theory most applicable to this scenario is constructionism.

I think it is relevant since by the end of the course, I had learnt almost every concept that was needed for the end-sem written exam but had not referred to written sources at all and had attended only half the lectures. The learning happened because of the project and assignments.

As our team progressed in the project, i had to search concepts and methods for designing and implementing different components.

The assignments, although not directly linked to the project were meant for us to explore different ways of doing things.

The process by which the code-base and our knowledge grew cleanly resembles bricolage. I would design or implement a feature, see the restrictions and benefits that the feature provides after it is implemented, and based on that, either add another feature or improve the implementation. This to-and-fro between the product and the creator is typical of programming tasks and is by definition, bricolage.

Another learning theory applicable to this course's design was the cognitive apprenticeship. The features that were present were :

- Global before local : We were shown the PLs developed by past year students
- Modelling : We were also shown worked examples (code and pseudocode) for basic tasks so that we could internalise the mediating methods. But we were not spoon-fed; we still had to modify the basic ideas introduced to us in the lectures to write working code for our projects.
- Scaffolding :

  The concepts were introduced to us on-demand rather than beforehand. For example, the cactus-stack data structure used for implementing function closures was taught to us only after we had already started on implementing functions.

  There was also structuring present throughout the implementation process. The timeline was first basic design, then implementing the interpreter, then iteratively improving on the design and the interpreter (allowing us to master writing an interpreter before we could start on the code generation), then moving to bytecode generation and creating the VM, which again form an iterative cycle since the bytecode is highly linked to the VM. This structure allowed for us to focus on one step at one time rather than making changes chaotically.
- Articulation : The labs allowed the TAs to monitor our learning and thus the professor could give adaptive feedback.
- sequencing : The assignments allowed us to reach proficiency in implementation of particular parts of the process that would later be used (with more complexity) in the project.

Yet another theory which might be applied here is that of cultural learning pathways; namely seeing this course as an episode in the learner's pathway for becoming a proficient compiler engineer. We don't wish the learning to just stop at the course but to cause a positive shift in identity, belonging and sense of competence too. In discussions with some of my peers I came to know of the various incentives they had for taking this course. I will point out some here :

- Industry relevance : Compilers and computer systems skills are in-demand in the current software development market.
- Personal career related incentives : Some of my batch-mates had secured internships in companies such as Qualcomm with roles highly centred around compilers.
- Research interest : One of my teammates was already doing research in PL theory. For him, taking this course was a natural decision.
- Curiosity, Challenge, Growth : Some students (such as me) were exposed to knowledge and challenges related to compilers before this and were thus going through centripetal participation. I particularly had heard that this is a "challenging" course and taking it would help me become a good computer engineer. That was my main reason for taking a project heavy course in an already heavy semester, potentially risking my SPI.
- Usage of already existing knowledge : Another classmate was doing a PhD in theoretical CS and had really good grasp on the Theory of Computation (TOC) which is what most of the interpreter designs are based on. She was not that good at coding and didn't want to improve either. It seems she took it merely because she had covered the pre-requisites and wanted to see that knowledge in action.

- Credits : This was one of the hardest possible systems basket courses. If the aim was just to complete credits, there were other courses better suited for that. Taking this course merely to complete credits requirements must be a mistake then; and from what I heard from my classmates, no one took it purely for the credits.

It should be kept in mind that this learning pathway p.o.v. allows us to project the effects of doing good or bad in the course onto the long term trajectories of the students. While this doesn't really changes design, it gives importance to building a sense of identity and competence for the students and preparing them for future participation. By including a learning pathway p.o.v. , I am also implicitly including activity theory and the community of participation lingo.

Appropriation of Design

I believe this course design is very appropriate for the learning goals. The semester that I took this course, it was conducted by Proff. Balgopal Komarath. This course is either conducted by him or by Proff. Abhishek Bischawat who conducts courses in the traditional lecture-then-assignment format with most weightage on theoretical exams. I was actually recommended to take this course under Proff. Balgopal Komarath by seniors.

I had already ramped up on compiler theory before starting the course but found no use of the theory for the most part since most of the course was practice based and we were using only a subset of the full compiler theory that is taught at the undergraduate level.

This course design had these merits over the traditional course design :

- Industry level knowledge *at the newcomer level*; thus not overwhelming the student with abstract knowledge
- Instrumentality of knowledge was present everywhere. As I said, merely by working on the project under guidance of the professor, TAs and peers, I was able to learn most of what was asked in the final written exam.
  Thus, I never felt the need to ask or answer "why am I learning this". It didn't *feel* like study.
- This course had excellent sequencing and scaffolding as compared to the assignments and projects given in some other CS courses where the student in entirely unguided and carries the full burden of literature review, design decisions and timely implementation. In such "unguided" projects and assignments, the students feel overwhelmed and often take shortcuts and even plagiarise if needed and beneficial; and are unable to develop identity and confidence.
- The practices of the exams, assignments and of the projects all relied on the same underlying domain knowledge and there was very little emphasis on heuristic and control strategies for the exam. Thus, doing good in the project would mean a very high probability of doing good in the exam.
- The difficulty of the tasks was correct for the newcomer level. As I described earlier, some CSE course projects cause heavy cognitive load on the students since they lack the necessary schema (as pointed out by Sweller). This course not only did not do that and provided guidance, but more than that; the tasks given were themselves of the correct difficulty for a beginner given the time we had.
- The labs were actually meaningful in this course. More often than not, the word "lab" is used synonymously with "test" since labs are often graded in courses. While continuous assessment is usually a good thing, often the skill being graded is very different from the skill being taught and the student thus has to shuffle between the two practices based on the weightage. For this course, the lab sessions were very similar to scrum meets in industry and to lab sessions held in research labs. Our progress and blocks were monitored qualitatively and not just graded.
- The freeloader problem was solved by giving a "total" amount of marks to each team for the project and having them split up the marks and justifying the split to the proff rather than giving the exact same number to each participant. A team-mate of mine who *was* a free loader was really angry when this was announced and I was rather happy since I had done around 70% of the work for the project. Thus, no negative perception against teamwork was developed. Due to this fact mainly, I believe that keeping a *group* project as opposed to just a project is justified since it allows the students to distribute the load, further reducing anxiety. It should be noted that the project, although being a group project was infact small enough to be doable by a single student in the allotted time.
  Another thing they did was that this rule was only made explicit at the very end of the course. If we had been told this at the start, then it would have become a rather unhealthy competition between the team-mates for contribution and a lot of students would choose to *not* form a team.
  Another peculiar thing I noticed was that the 50% share given to me was suggested not by me but rather my team-mates. At some point they had started to respect me as competent participant. This friction-less distribution of weight-age underlines an implicit power dynamic based on competence and participation; typical of a CoP. As I had already said, doing good in the project was directly correlated to doing good in the exam and because I had an excess of points at that point and had built a belonging towards the team, I willingly gave away some 10% of my share to others.

- The exams being on the concepts gained while working on the project further highlighted the importance of the concepts and helped the students in identity building and developing belonging (to the field itself, not just the people involved in the course).
- A bad consequence of the group project setting was the underemployed worker issue. The students that were seen as less competent were not assigned heavy tasks and consequently didn't contribute a lot or develop perceived competence or even an internal sense of competence.

Overall, the course was a good learning experience for most. For me particularly, it eventually led to ideation of another project idea based around creating a parser for a practical subset of Oracle SQL. This was only possible because I had enough faith in my ability to be able to complete this massive task. Unfortunately, because of time constraints I was never able to continue working on that but that idea still sits in my bucket list and I will probably do it after I graduate.

Improvement

- I was fortunate enough to have a team-mate who was an old-timer in PL theory and could provide a lot of guidance to me timely. So, I didn't have to wait for the lab meeting to get guidance from the TA. Many teams were not so fortunate. So I believe pairing up an old timer with a new-timer is beneficial.
  The old time doesn't have to focus on the menial things such as writing code and details of design and can focus entirely on good PL and compiler design.
  Meanwhile the new-timer doesn't get overwhelmed and is given one-on-one feedback.
- The roles of the old timer and the new timer must not be explicitly stated to them so that agency is preserved and respect is earned naturally.
- While the assignments did help us a lot for learning new ways of doing thing, some of it felt unnecessary (abstract knowledge) since it wasn't always linked to the project.
  I suggest keeping assignments on a topic just before it might be used in the project.
- I also suggest that each new-timer should implement each stage in the pipeline (for the compiler) in a different way from others (for example, implementing a register based VM and bytecode rather than a stack based VM and bytecode). This allows for easier work distribution, a wider range of concepts that are learnt and articulated (so that the different new-times in the team can communicate) and a potential for problematisation (such as comparing run-times of different approaches or describing the difficulties to each other). The new-timer should only be graded based on the approaches he implemented, thus resolving the free-loader and underemployed worker problems entirely.
  So that all the new-timers learn all knowledge used by different approaches, they should explain not just their approach but also approach implemented by other new-timers in the team to the TAs in the lab meetings. This part can also be graded (for the full team) with random assignment of what approach to explain to the other team.
- From an activity theory P.O.V, each team is a sub-activity system with these subjects:
  - old timer with the objective of guiding new timers, monitoring progress, and communicating with the TA as well as all the objectives of the new-timers.
  - new timers with the objective of implementing their assigned approach for a stage, explaining it to other team-mates (both new-timers and old-timers), and understanding other approaches.
- While the course was excellent in teaching sufficient skills for working with compiler and PL design, the breadth of material completed was much lower than what would be in a traditional course. For example, by the end of the course we had excellent knowledge of recursive descent parsing but lacked any knowledge for LR or RR parsing algorithms.
  To solve this, we can use books as learning aids that would also introduce the students to a wider range of approaches not covered in the course.
  My teammate who was an old-timer had a wider range of concepts because he read all those things in books, even though he doesn't use any of that in practice.
  I believe keeping these things as "see also" sections would do them justice since

  1. The students are already motivated to read more since they have to choose different methods for implementation of parts and need to review the methods first, and secondly so that they can learn the lingo used by the old-timers when they give guidance.
  2. Any knowledge that is abruptly forced and is not instrumental to something enabling (and not oppressive, such as grades) is meaningless (abstract) for the learner and won't be retained in the long term.

  For the end-sem exam then, we can just take the intersection of approaches done by teams and ask theoretical questions on that, thus not causing a split between the practice of the exam and the practice of the project.

- Another way to increase experimentation and width of knowledge learnt, we can ask the students to justify why they chose one method over another and have very little weightage to this (think, 1%), thus *forcing* them to do read the literature.
  Students might hate this, but just as much as they will hate the assignments (which are also not directly helping them progress in the project).

Alternative LS paradigms to base this course on

Learning Paradigm	What Students Do (Activities)	How Students Are Evaluated	Problem / Instruction Sequence	What Would Be Gained Compared to Constructionism	What Would Be Lost Compared to Project-Based Constructionism
Behaviorism / “Bad” Instructionism	Drill tasks, repetitive coding exercises, quizzes	Frequent testing, checklist-based grading, response accuracy	Instruction first, repeated practice, quizzes; minimal problem-solving	Clear expected outcomes, measurable skill mastery	Loss of deep problem solving, creativity, social interactions
Cognitivism / “Better” instructionism	Concept mapping, algorithm pseudocode exercises, case studies	Performance on problem sets, concept tests, transfer tasks	Instruction leads with schema explanations; problems applied after learning	Structured mental models, improved knowledge transfer	Less situated, artifact-centered learning; fewer social interactions
Pure Constructivism / personal project based course	Self-selected projects or experiments; building personal models	Reflective journals, portfolio reviews, self-assessment	Problem exploration leads; instruction minimal and formative	Meaningful individual knowledge construction, exploration	Less teamwork, shared artifact creation, external guidance
Social Constructivism	Group discussions, collaborative design sessions, peer critiques	Participation grades, group project deliverables, peer evaluation	Instruction scaffolds social dialogue; problems occur within collaborative contexts	Rich peer discourse, collaborative meaning making	Less artifact focus; more diffuse team accountability
Discovery Learning	Open-ended labs, unguided problem exploration	Observation of discovery process, final exploration report	Problem first, no instruction until reflection or final	Exploration-driven motivation, creativity	Lack of guidance may lead to gaps or misconceptions
Inquiry-Based Learning	Investigations guided by study questions, research presentations	Quality of inquiry process, presentation skills, reports	Instruction sets inquiry questions; problem exploration follows	Deeper analytic questioning, research skills	Less focus on artifact creation and ownership.
Problem-Based Learning	Research and propose solutions to specific problems, group collaboration	Problem solution quality, group process reflection, peer assessment	Problem presented first; instruction based on identified gaps post-solution	Enhanced problem-solving skills in authentic contexts	Less sustained artifact work; episodic problems
PSI (Problem-Solving then Instruction)	Attempt problem sets individually or in groups; follow-up lectures	Problem resolution, conceptual tests post-instruction	Problem first (productive failure), instruction follows systematically	Effective conceptual change and correction	Reduced continuous team collaboration and artifact iteration
Situated Learning	Participate in real or simulated professional contexts	Performance in situated tasks, community feedback	Learner immersion in practice precedes instruction or guided reflection	Authentic engagement with practice communities	Less emphasis on technical mastery or product delivery
Experiential Learning	Hands-on labs and reflection on that but no long-term project	Reflective essays, skill demonstration	Experience leads; reflection and theory follow	Rich personal reflection on lived experience	Less collaboration and shared knowledge construction

Learning Paradigm	What Students Do (Activities)	How Students Are Evaluated	Problem / Instruction Sequence	What Would Be Gained Compared to Constructionism	What Would Be Lost Compared to Project-Based Constructionism
Case-Based Learning	Analyze and discuss cases individually or in groups	Case analysis quality, participation, presentation	Cases present complex problems; instruction embedded in analyses	Exposure to varied authentic cases; analytical reasoning	Reduced iterative design and artifact creation
Collaborative Learning	Group projects, peer teaching, collaborative problem solving	Group and individual assessments, peer feedback	Instruction embedded; problems tackled collaboratively	Enhanced communication, peer learning benefits	Potential diffusion of individual accountability
Active Learning (general)	Flash discussions, clicker questions, group problem-solving	Participation, problem solutions, immediate feedback quizzes	Problem and instruction are interleaved dynamically	Increased engagement and feedback loops	Risk of broad non-specific activities lacking coherence
Pure Cognitive Apprenticeship	Guided practice with mentors, observation, coached reflection	Mentor evaluations, progressive competence demonstrations	Instruction scaffolds practice; problems arise in context	Strong mentorship, situated cognition, community identity building	Less emphasis on ownership and cumulative product

It’s clear that the one common failure of all other theories is the sense of ownership and confidence. While some paradigms such as a pure cognitive apprenticeship might enable the student to work on complex real life projects and build a sense of competence by the end, the identity formation will be limited. On the other hand, paradigms such as situated learning and problem based learning overload the student potentially causing a negative episode (in terms of cultural learning pathways). Other paradigms such as inquiry learning, discovery learning and active learning are great for gaining in-depth theoretical knowledge but don’t necessarily give the control strategies that a real project gives and neither the sense of industry level competence which is needed for undertaking real engineering tasks.

Again, the main goal of a course on Compilers in an institute where the majority of students go towards industry should be to *enable* the students to both actually take on compiler related tasks in their careers. For them to take on such tasks, they need first a sense of competence, and second, a genuine interest. The students should also have the competence needed to finish these tasks and thus experience positive episode in their (learning) pathway in the future.

The only paradigm that seems to produce all three (sense of competence, actual competence, and interest) in this case is a mix of constructionism and cognitive apprenticeship. The constructionist approach is rather good at creating a sense of ownership, style, and thus identity and a *sense* of competence, and the cognitive apprenticeship approach is really good at creating actual competence.