Personal Statement: Jean Yang

Those passing by Maxwell Dworkin G135 one spring evening last year may have been surprised to see a caped figure dangling his strangely unfazed peer upside-down by the boots. Both characters were holding cardboard arrows; behind them on the board was a sea of `*`'s and `->`'s. This was the Computer Science 51's C review session, and my fellow teaching session leader (known to attendees as Guardian of the Heap) was holding me upside-down to demonstrate the conventions of stack and heap growth directions.

Each year the course holds a review of the essentials of memory management before diving into C++; this year I had convinced the other teaching fellow to join me in costume to demonstrate how and why C provides an abstraction over the data memory. My own discovery of the power that comes from languages' semantic constructs led me to emphasize this point to all who would listen. My students may recall—fondly or otherwise—my e-mails (sometimes after the course was over) about discoveries of papers such as Fateman's *Why C is Not my Favorite Language* and Stroustrup's *Evolving a language in and for the real world: C++ 1991-2006*.

What first drew me to programming languages was a class research project on parallel computation models. Because physical limitations have caused chip manufacturers to begin increasing the number of cores per processor rather than transistors per core, continuing the trend of exponential processor speed growth involves correctly and efficiently parallelizing tasks across growing numbers of cores. Solving this problem could eventually eliminate many of today's computational limitations. This convinced me that leveraging computational tools was potentially much more powerful than any speedups from algorithmic analysis alone.

I have come to believe that the tool innovation will come from developing high-level language constructs. In implementing algorithms for my research in computational biology, I saw how useful it was to build my systems out of modular, reusable components. In creating the vision and control systems for my RoboCup team's autonomous soccer-playing robots, abstraction was crucial to maintaining sanity, both in the organization of the code and in the organization of the team. As a technical director of our team, I saw how crucial interfaces were for allowing us to forget about previously solved problems and for allowing us to work separately on interacting code. After taking courses on computer hardware, compilers, and programming languages, I came to see how language abstraction has allowed many the advances of the last few decades.

My experiences have made me increasingly convinced of the importance of having the proper language tools. When I interned at Google last summer, I discovered how much time people put into implementation, bug hunting, and proving code correctness. Though I enjoyed the craftsmanship involved with writing disciplined C++ code that was required for producing correct, efficient and readable code, I noticed a glaring need for language constructs with better error prevention and better ways of demonstrating code correctness. In the hours I spent waiting for my code to compile, I thought about how to build language tools that would better support separate compilation and discourage the production of so many dependencies.

Not only did my experience in industry convince me that there are real problems that I should solve, but it motivated me to return to academia to solve them. At Google I noticed that production pressures tended to make tool development less important than product development and that the need for backwards compatibility limits the possibility of language innovation. While tool innovation certainly does occur in industry, the greater experimental freedom that academic settings allow is more conducive to producing projects such as Gprof, a C profiling tool developed at UC Berkeley, and Fortress, Sun's revision of the Fortran programming language.

My own undergraduate experience contributes to my desire to remain at the university. Because of the influence of my professors and activities in my decision to continue studying computer science at all, I have felt strongly about increasing undergraduate accessibility to resources, academic and otherwise. As president of the Harvard College Engineering Society, I have organized community-building and advising events such as lunches with professors, freshman advising events, and Women in Computer Science activities. I have also worked to make it possible for students to pursue their own projects: I have helped to acquire thousands in funding for our RoboCup team and critical lab space to test our soccer-playing robots.

Of course, one of the main reasons to stay in academia is to teach. Teaching has been the best thing I have done as an undergraduate: not only has it helped reinforce my understanding of the material, but it has also given me perspective on the purpose of my own coursework. Selfish reasons aside, I love teaching because I find the concepts interesting and relevant; my goal is for other people to gain enough understanding to be able to feel the same. I want to teach to provide accessibility to beautiful theoretical results such as the proof of the Halting Problem, to impart necessary practical knowledge such as the proper construction of a Makefile, and to convince others that solving problems in computational tools is interesting and important.

It would be ideal to remain in academia as a professor because I could pursue my research interests while training and recruiting others to solve relevant problems in programming languages. The NSF fellowship will provide me with the flexibility to work on such problems of my interest while I pursue this goal.