Nathan Rabins

iEx6 – Programming Environment Survey

1. A programming environment exists to make development simpler for the developer. In light of this, every programming environment should provide certain services. The first of these is an editable code window. If there exists no such window, then code must be entered elsewhere, defeating the purpose of an integrated development environment. Next, a programming environment should provide some way of allowing input and output from and to the user. In Eclipse and Dr. Racket, this is done through the console. Last, a programming environment should facilitate error checking. There are numerous levels of said checking, from syntactical error notification to logic errors. In general, more error alerts and debugging assistance are more useful than fewer.

2. While Dracula features a difficult learning curve, there are a few things that are to be appreciated about it. First, the console window is functional and fairly prompt. This is important for its language, in which definitions are constructed and accessed through the console. Second, through use of the teachpacks, Dracula has access to a functional testing suite. Although we didn't use this much at the beginning of the semester, by the end it was a necessary and desired help. Third, Dracula features a "program contour" view where the overall form of the code can be seen in its entirety. This was helpful in honing in on sections of code which were "ugly" (too big/long, not encapsulated enough). Last – and this is more an appreciation of lisp than of the environment – Dracula enforced the structure of its programs well. The tab key was intuitive and useful in checking alignments; this ensured that the beautiful lisp structure was maintained.

3. It is true that Dracula is a functioning environment; however, it is not user-friendly. First, it is a very slow program. This is unfortunate in a programming environment, as programmers are the sort of people who like to get their thoughts down as quickly as possible. Often, Dracula lags behind, leading to typographical errors. Compounding this are Draculas graphical glitches in which deleted code shows through in part. Resizing clears these anomalies, but it is a step that shouldn't be necessary. Second, Dracula's error handling, notification, and debugging tools are abysmal. Simple mistakes, such as switched letters in a variable name or a missed edge case, were an order of magnitude more difficult to diagnose in this environment than one with an

easy-to-use and intuitive debugging system. Lastly, Dracula lacks autocompletion, which is quite common in other development environments. This small tool is incredibly helpful, especially when dealing with a large project. Its absence in Dracula means long periods of time scanning previous lines of code for reference.

4. In the hypothetical situation where I am to design the software for use in a dangerous situation, I would opt for a theorem-based language like ACL2 over an easier-to-implement language such as Java, C#, or C++. While the latter options would shorten the development cycle, there would be a much greater chance of code failure which, given the scenario, could be a life-threatening proposition. Coding in ACL2 and using a TDD approach (complete with its theorems, properties, and check-expects), I would be exceptionally confident in my code's correctness. When it has to be done right, the language that offers the most ways of checking correctness is the proper choice.