Nafis Upshur
CS 242
October 5, 2002

SUMMARY OF REVIEW SESSION (9/30/02)

• FUNCTIONS AND COMPUTABILITY

Partial functions are functions that are not defined on some arguments. Programs are partial functions because they might have undefined behavior on some inputs (e.g. division when the denominator is zero) or they may not terminate. The class of partial recursive functions is the class of functions on the natural numbers (1,2,3,....) that can be computed. Basically, a partial recursive function is a function that can be computed by a Turing Machine. All standard programming languages are Turing complete, which means that they can express the class of partial recursive functions and therefore programs in these languages can be computed by a Turing machine. The "recursive" part of this definition refers to the fact that all computation can be expressed as recursion. No, that fact is not obvious.

Why do we care about computability at all? Because it saves us from wasting our time on problems such as determining whether a program will halt on a given input. This is known as the hating problem, and we can prove that there is no program that we can ever write that will decide it. In fact,, we can show that there are other problems that are not solvable by showing that they can be reduced to solving the halting problem. As an example, we showed in the review session that it is not possible to construct an algorithm that determines whether a given program halts on all inputs. We proved that if such a construction were possible, then we could also solve the halting problem.

• LISP

Computability is all well and good, but if we cannot see how it maps to actual programs that we implement in some particular language, then it's of little use to us. If you're a C programmer, then it might be hard to get you to think about your programs in terms of functions that compute something. Lisp on the other hand, is a functional programming language, so programs can be readily viewed in functional terms. This makes it a little easier (we hope) to relate our somewhat abstract notions of computabiliy to real programs.

We discussed some basic features of Lisp (atoms, lists, cons cells, etc.). In pure Lisp, there are no side effects, which means that the state of the machine (a particular assignment of values to memory locations) does not change when expressions are evaluated. This might make it easier to reason about what is happening in a Lisp program. We also talked briefly about garbage collection, as in "Lisp has garbage collection." I noted that Lisp supports higher-order functions, which means that functions can be passed as arguments and returned from functions. Function pointers can also be passed as arguments and returned from functions in C, but we cannot create a function within the body of a function and then pass it to another function as an argument. We also cannot create a function and return it from another function. This is because functions can only be defined in global scope in C. We will talk more about higher order functions and scoping, as well as the structures needed to support these features, later in the course.

As an exrecise for gaining more insight into the link between computation and recursion, you might try writing a few simple programs in Lisp. Notice how the absence of while loops and assignment forces you to express even simple computations in recursive terms. Indeed, the challenge in writing some Lisp programs is solely in thinking about how to compute something without the use of assignments and loops. Remember, though, that all programming languages express precisely the class of partial recursive functions and are therefore equivalent in expressive power (e.g. there are no functions that you can compute in C but not Lisp, or in Lisp but not C).

• LAMBDA CALCULUS

Things are more believable when expressed in a mathematically rigorous fashion, so we'd like to find a general language for talking about functions and computation. Lambda calculus is such a language. All lambda calculus expressions are one-argument functions. The system posits some basic rules for function applications and the like, and from these rules we gain a structure that is equivalent in expressive power to Turing machines (not obvious). Since Turing machines can compute the class of partial recursive functions, and all standard programming languages express exactly that class of functions, we can view expressions in the programs we write as lambda calculus expressions. Ultimately, this enables us to prove things about programs using lambda calculus.

We noted that it's especially easy to see the link between Lisp and lambda calculus (the "lambda" syntax in Lisp used to denote anonymous functions). We talked briefly about equivalence and substitution rules, and I wrote some rules on the board for renaming bound variables in lambda expressions. Due to a series of increasingly poor explanations, we spent quite a bit of time talking about the concept of currying. Expressions in lambda calculus are functions of one

argument. But that doesn't mean that we can't represent functions of more than one argument. To represent a function of two arguments, for example, we write a function of a single argument, which, when applied to the first argument, returns another function that accepts the second argument. This is known as currying. We care about currying because it shows that we don't have to add rules to lambda calculus for dealing with n-argument functions; they can all be expressed in terms of one-argument functions.