Basic Syntax Macros

Basic syntax macros were implemented for the final project, which allow programmers to define their own syntactic sugar. Macros are similar to function calls, though they differ in the sense that their input and output are syntax. Macros are evaluated at compilation, unlike functions which are evaluated at run-time. The syntax which a macro represents is substituted into the AST where they appear, along with any arguments which were passed by the user. They offer a form of syntactical code replacement, as the code which they represent is substituted immediately where they are invoked. After parsing code which involves macros, their evaluation is complete and equivalent parsed code is substituted, so it is as if the macro never existed from that point forward, appearing nowhere as a function definition or application within the AST.

The Iniquity source code was used as a starting point for the project. In order to support macro definitions, the AST and method of parsing required structural changes. The project can be broken down into three distinct phases: initial naive parsing, substitution, and cleanup. The parse function itself reflected these three phases, as substitution of macro definitions was done on the AST resulting from naive parsing, followed by removal of macro definitions. The output of parse is an AST with substituted macro syntax and no trace remaining of any macros.

The naive parsing of these statements is nearly the same as with normal function definitions, though a new pattern match was added to parse-define that was the same as the code to handle ‘define statements, with two key differences. First, rather than matching only on ‘define, matching was done on ‘define-macro as well and second, the function name used was the symbol for the original function with a ‘macro prefix in order to flag it as a macro for later processing. A function named naive-parse was defined which was nearly identical to the original parse function, with the key difference that the match on the list of definitions was replaced with a match on either a list of definitions or macro definitions - the underlying code remained the same otherwise. In this treatment of macro-definitions, they are parsed in quite the same way as function definitions, with the difference lying in the prefix “m” appearing before macro definition “function” names.

Substitution was a bit trickier and required two new auxiliary functions, substitute-e and substitute. Substitution was initiated within parse by calling substitute-e on the expression within the naively parsed AST. It takes two arguments: ds and e which represent the definitions to potentially by substituted and expression to perform substitution on. Substitute-e recursed

through the subexpressions within any expressions, such as the subexpressions of Prim1 or If. In the case of an App, if the function symbol with the ‘macro prefix appended matched any function symbols inside of ds (assuming the right number of arguments was provided), substitution commenced by calling substitute on the expression(s) of the App, otherwise substitute-e recursed within the subexpression(s) of the App. Substitute takes two arguments: vs and e which represent the bound variables and expression to substitute the variables in. The vs parameter was created by using the same zip function that appears in interp.rkt, applied to the variables and passed arguments. As with substitute-e, substitute recursed through any possible subexpressions of larger expressions, and in the case of a match on a Var, the substituted value was returned using an auxiliary vars-lookup function that mirrors defns-lookup, with the difference being the match-lamba of (cons x1 x2) and checking whether x1 equals the variable being looked up. This concludes substitution, since any bound variables that could exist in any subexpression within the macro is replaced with its passed value.

Finally cleanup commenced in a fairly straightforward manner. The cleanup function is called on ds, the definitions of the AST, after necessary substitution is done on the function bodies. Cleanup then iterates through the resulting list checking at each element whether each function definition symbol has the ‘macro prefix, and if so discarding that element. If the prefix was not found, the list itself was returned. This works because all macro-definitions occur before function definitions, so no macro definitions remain once any function definition is found that doesn’t have the macro flag. If ds is at any point empty, the empty list is simply returned.

Testing was done by checking whether various expressions involving macros of different forms would first parse, then compile and interpret as expected. Test cases were chosen so that every single distinct possible pattern match was tested. Additionally, conceptual test cases were chosen to ensure that multiple invocations of the same macro, invocations of different macros, function calls involving macros, and macros involving function calls all behaved as expected. The test cases are contained within the test-runner.rkt file, which leverages the existing testing framework of Iniquity.

Macros allow for programmers to define their own syntactic sugar, as they are not explicitly evaluated but the equivalent syntax they represent is substituted after parsing. Incorporating them requires significant changes to how the AST is handled, as a larger reference frame is needed to perform the necessary substitution.