Overview

For my extensions, I chose to implement new atomic types and operators. In this case, I opted to add floats, strings, chars, and units to my program, and I chose to both extend some of the already existing operators to include these new types, as well as create some new ones. For the floats, this included all of the existing Num operators applied to floats, such as Plus, Minus, Equals, etc. For the chars and strings, I extended both the Equals and LessThan operators to them, and I also redefined the Plus operator to work as concatenation for the strings. A unit simply returns itself, so I did not make any new operators for it.

Float

I first expanded the definition of an expression to include Float, which takes in a float data type as its value. In addition, I expanded the definition of binops to include Fplus, Fminus, and Ftimes, in order to distinguish them from the integer arithmetic operators. I also extended the unop definition to include a RoundtoInt unop that rounds floats to the closest integer while converting them to a Num type expression in the process. Next, I edited my definition of the functions that handle substitution and dynamical evaluations by including the new Float type as one of the match cases, returning itself when evaluated. In addition, I added the new Fplus, Fminus, and Ftimes to the match cases under the Binops pattern matching, along with the Equals and LessThan operators, appropriately applying the correct float operator to each one. Then, I added the RoundtoInt unop to the pattern matching of the unop expression, which behaves similarly to the Negate unop, but can only be applied to Floats for obvious reasons, raising an error when one tries to apply it to any other type. I programmed the RoundtoInt operator by defining a function rounder that takes in a float and rounds floats with decimals >= .5 up and the rest down for positive floats, and vice versa for negative floats.

Then, I extended the lexical analyzer to include floats. I did this by allowing decimal numbers, like 1.5, 2., 0.9, to be written in concrete syntax and then converted to an abstract representation. In addition, I extended the lexical analyzer to include the float operator symbols +., –., *., ∼– by adding them to the sym hashtable, and I made the symbol ∼= represent the RoundtoInt operator, as the tilde is often used to signify approximations. Also, I made the float negation ∼– instead of the usual ∼–. because I often found it confusing when I wrote negative floats and they looked like they had two decimals, and I do not think using the ∼– for both ints and floats causes any issues. Instead, it can lead to less confusion and it makes it easier to type a negative float (less key strokes).

I then added the new float type and operators to the MiniML parser using the int/Num types as an example. Finally, I edited my previously defined exp_to_abstract_string, exp_to_concrete_string, free_vars, and subst functions to handle the new Float type and operators, acting much like the Num type and

its operators in each of these functions.

Examples of usage:
4.5 +. 3.83 returns 8.33
3. -. 3.83 returns -0.83
3.5 *. 2.5 returns 8.75
2.5 < 2.6 returns true
2.5 = 2.6 returns false
∼– 5.5 returns -5.5
∼= 5.5 returns 6

Char

Like Float, I expanded the definition of an expression to include Char, which takes a char data type as its value, and edited all of the functions in expr.ml to account for the Char type. Similarly, I modified the evaluation functions in evaluation.ml to handle the Char type, which is an atomic data type so it just returns itself when evaluated. In addition, I made it so two existing binops extend to chars, the Equals operator and the LessThan operator. The former behaves as you would expect it to, checking to see if two chars are the same, and the LessThan operator takes two chars and checks which one has a lower ASCII value, so it can be used to sort things alphabetically.

Then, I extended the lexical analyzer to include chars, making it so the concrete syntax requires that a char be defined using a single quote ' followed by one character and then closed using another single quote '. However, in order to make it so the char value that is stored into the char expression does not include those single quotes, I had to use String.get on the inputted concrete syntax to extract only the one character. Also, I could not figure out how to raise an error in the case that multiple characters are inserted between the single quotes, so instead MiniML will always just ignore any characters besides the first one if placed in between single quotes. Finally, I added Char as a token to the MiniML parser.

Examples of usage:
'a' returns 'a'
'a' = 'a' returns true
'a' < 'b' returns true
'c' < 'b' returns false
'a' = 27 returns an error
'abba' returns 'a'

String

Much of how I defined String is the same as how I defined Char, only it takes in a string data type as its value. In addition, I extended the Plus binop to work on

strings, acting as an operator that concatenates two strings. I chose to reuse the Plus binop instead of defining an entirely new one for concatenation, one that might use the carrot symbol instead of the plus sign to represent concatenation, because I thought it was more efficient. I did not see any reason to define an entirely different binop for concatenation when Plus conveys the same meaning for strings, and it would have gone unused if I had defined a separate concatenation operator, simply raising a type error if one tried to apply it to strings. Also, the Equals operator checks if two strings are the same, but I altered the LessThan operator to compare the lengths of strings instead of their ASCII values. I did this using the String.length function to compare the lengths of two strings in the evaluation functions. I did this because I found that I often need to compare the lengths of strings more often than I need to compare their ASCII values.

I altered the lexical analyzer and parser much the same way as I did for Char to handle the new String type. Again, since I made the concrete syntax require that a string start and end with quotation marks, I had to cut these quotation marks out of the inputted string so what is stored in the string expression does not include the quotes, as this complicated the concatenation process. Therefore, I took the substring, using String.sub, of what the user entered to include every character besides the first and last (the quotation marks).

Examples of usage:
"Abstraction" returns "Abstraction"
"Abstraction" = "Abstraction" returns true
"Abstraction" < "is" returns false
"Abstraction" + " is " + "Mahogany" returns "Abstraction is Mahogany"

Unit

I defined Unit in the expressions definition, and it takes no values, acting similar to Unassigned. I then modified all the functions in expr.ml to handle the Unit expression. In the evaluation file, I had Unit return itself whenever it is evaluated. Then in the lexical analyzer, I included unit in the concrete syntax by representing it as (). I did this by extending id and the keyword_table to include "()", which would map to Unit when inputted. I then added Unit as a token to the MiniML parser. I did not add any new operators or extend any of the preexisting ones to Unit because you cannot do anything to a unit; it just gets returned when called on.

Examples of usage:
() returns ()

Bool operators

I created the Or and And operators for bools, including them in the definition of binops, and implementing them in the evaluation functions to only work when

called on Bools, raising an error otherwise. They work as you would expect them to, with And returning true only when both bools are true, and Or returning true if either of the two bools are true.

In the lexical analyzer, I added the | and & characters to sym, and then I extended the sym_table hashtable to map || to Or and && to And. Next, I added both Or and And nonassociative tokens to the MiniML parser.

Examples of usage:

true && false returns false
true && true returns true
true || false returns true
false || false returns false