CS51, FINAL PROJECT

April 28, 2020

Report on the extensions of the MiniML interpreter

(1) Changes to the Interpreter

For the final project, I decided to modify the parser to allow for syntactic sugar and add extra operations for integer values. I specifically modified the parser to allow let-bound functions to be defined in the form "let f x y = x + y in f 1 3" and for unbound functions to take multiple arguments, for example "fun x y -> x + y." In addition, I added the greater than, greater than or equal, less than or equal, quotient, and mod functions to the syntax of the language.

(2) Adding syntactic sugar

In implementing the syntactic sugar, I modified the parser in miniml parser.mly. In this file, I created two new grammar structures called "sugfun" and "suglet", which would allow functions and let expressions to take in more variables before the arrow (->) or equal (=) signs, respectively. For this to happen, I had to modify the fun inputs so that after the variable, notated as ID, a sugfun expression would come. The sugfun expression can simply be the arrow (->) to what the function evaluates, or another ID followed by another sugfun expression which could now be the arrow (->) to what the function evaluates. This allows the parser to parse a function with any number of variables. Similarly the let and let rec expressions behave in the same way. The parser allows for a suglet expression after the variable (ID). The suglet grammar structure allows for more ID's or an equal sign (=) to indicate to the parser that it should now include what comes after the equal sign in the second part of the data type. In this way, the interpreter parses a function such as "fun x y -> x + y" as "let fun x -> fun y -> x + y" and a let "f x = x + 1 in f 1" as "let f = fun x -> x + 1 in f 1." For the interpreter to work as expected, I had to create a separate grammar structure because after the let or let rec should come a number of variables (ID's) followed by an equal sign and not an arrow (->). If I had implemented it as a single grammar structure, the parser would allow me to interpret expressions like "let f x -> x + 1 in f 3" or "fun x y = x + y," which should instead raise an error.

```
        | LET ID suglet IN exp          { Let($2, $3, $5) }
        | LET REC ID suglet IN exp      { Letrec($3, $4, $6) }
        | FUNCTION ID sugfun            { Fun($2, $3) }
        | RAISE                 { Raise }
        | OPEN exp CLOSE        { $2 }

sugfun: ID sugfun                 { Fun($1, $2) }
        | DOT exp                 { $2 }

suglet: ID suglet                 { Fun($1, $2) }
        | EQUALS exp              { $2 }
;
```

Figure 1. Code that allows for syntactic sugar in the parsing method.

```
Last login: Tue Apr 28 01:44:48 on ttys004

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) Martins-MacBook-Pro-2:project-mreyes2000 martinreyes$ ocamlbuild miniml.b
yte
Finished, 17 targets (17 cached) in 00:00:00.
(base) Martins-MacBook-Pro-2:project-mreyes2000 martinreyes$ ./miniml.byte
Entering ./miniml.byte...
<== fun x y z -> x + y + z;;
--> Fun(x, Fun(y, Fun(z, Binop(Plus, Binop(Plus, Var(x), Var(y)), Var(z)))))
s=> [function x -> [function y -> [function z -> [[x + y] + z]]]]
d=> [function x -> [function y -> [function z -> [[x + y] + z]]]] where []
<== let sum x y = x + y in sum 5 6;;
--> Let(sum, Fun(x, Fun(y, Binop(Plus, Var(x), Var(y)))), App(App(Var(sum), Num(
5)), Num(6)))
s=> 11
dx> evaluation error: Variable x not found in environment
<== let rec fact n = if n <= 1 then 1 else n * fact (n - 1) in fact 5;;
--> Letrec(fact, Fun(n, Conditional(Binop(LessEqual, Var(n), Num(1)), Num(1), Bi
nop(Times, Var(n), App(Var(fact), Binop(Minus, Var(n), Num(1)))))), App(Var(fact
), Num(5)))
s=> 120
d=> 120
<==
```

Figure 2. The interpreter parsing and evaluating three different types of usages of the syntactic sugar.

(3) **Adding more integer operations**

In addition to adding syntactic sugar to my interpreter, I also decided to add more binary operations for integers, such as more inequalities and the division and modulus operations. To do this, I had to modify the lexical analyzer in miniml lex.mll, where I defined new symbols such as "<=" (less than or equal), ">=" (greater than or equal), ">" (greater than), "/" (integer division), and "%" (modulus) and assigned them to their tokens. Then, I modified the parser in miniml parser.mlym where I defined the tokens and gave them an algebraic order. Finally, I had to define these operators as new types in expr.mli and implement them in borth expr.ml and evaluation.ml in most of the functions. Luckily, the functions are coded so that one can easily define and implement more types in the functions.

```
Last login: Tue Apr 28 01:44:54 on ttys005

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) Martins-MacBook-Pro-2:project-mreyes2000 martinreyes$ ocamlbuild tests.by
te
Finished, 11 targets (11 cached) in 00:00:00.
(base) Martins-MacBook-Pro-2:project-mreyes2000 martinreyes$ ./miniml.byte
Entering ./miniml.byte...
<== let x = 5 in x > 1;;
--> Let(x, Num(5), Binop(GreaterThan, Var(x), Num(1)))
s=> true
d=> true
<== 5 % 2;;
--> Binop(Mod, Num(5), Num(2))
s=> 1
d=> 1
<== 2 % 5;;
--> Binop(Mod, Num(2), Num(5))
s=> 2
d=> 2
<== 4 >= 4;;
--> Binop(GreaterEqual, Num(4), Num(4))
s=> true
d=> true
<== 4 > 4;;
--> Binop(GreaterThan, Num(4), Num(4))
s=> false
d=> false
<== 4 <= 4;;
--> Binop(LessEqual, Num(4), Num(4))
s=> true
d=> true
<==
```

Figure 3. The interpreter parsing and evaluating the newly implemented binary operations.