DLP

LAMBDA CALCULUS INTERPRETER

1. MULTI-LINE EXPRESSIONS

To enable multi-line expressions, we have modified main.ml to add a function read_command() that continuously executes read_line() until it detects a ';'. At that point, it returns all the read lines concatenated. This function is subsequently used in the main function of the main.ml file.

Example:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  sum 21 34
;;
```

2. PRETTY-PRINTER

New functions have been created: pretty_print and pretty_print_ty, inspired by the previous ones. In these new functions, only the strictly necessary parentheses are included.

Example:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  sum 21 34
;;
```

3. INTERNAL FIXED-POINT COMBINATOR

In lexer.mll, when a letrec is encountered, a LETREC token is passed to the parser, and when a fix is encountered, a FIX token is passed. In the parser, we define rules that allow recognizing the structure of a letrec or fix, triggering corresponding actions to return the associated term. In the case of letrec, a term TmLetIn is returned, with one of its arguments being a new term, TmFix. In lambda.ml, eval1 has been modified to add new cases to handle the new term, TmFix. Similarly, the typeof, subst, and free_vars functions have also been modified to include the new type.

Examples:

SUM:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  sum 21 34
```

;;

PROD:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n: Nat. lambda m : Nat.
          if iszero n then
          m
          else succ (sum (pred n) m)
  in
  letrec prod: Nat -> Nat -> Nat =
          lambda n: Nat. lambda m: Nat. if iszero m then 0 else sum n (prod n (pred m))
  in
prod 12 5;;
```

FIBONACCI:

```
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat.
  if iszero n then
          m
  else
          succ (sum (pred n) m)
  in
          letrec fib: Nat -> Nat =
          lambda n : Nat.
          if iszero n then
          0
          else
          if iszero (pred n) then
          1
          else
          sum(fib (pred (pred n))) (fib (pred n))
in fib 5;;
```

FACTORIAL:

```
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat.
  if iszero n then
          m
  else
          succ (sum (pred n) m)
  in
    letrec prod : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat.
```

```
    if iszero n then
      0
    else
      sum (prod (pred n) m) m
   in
     letrec fact: Nat -> Nat =
     lambda n : Nat.
     if iszero n then
       1
     else
       prod n (fact (pred n))
in fact 5;;
```

4. GLOBAL DEFINITIONS CONTEXT

Firstly, in lambda.mli and lambda.ml, we added the command type, which is used to represent the actions that the lambda interpreter can perform. Specifically, commands for evaluation (Eval) can be issued to evaluate a term in the current context, or assignment commands (Bind) can be used to bind a name to a term. The execute function has also been added to differentiate between these commands and execute the corresponding action. Now we have a context for definitions and another for types, so the types of functions in lambda.mli have been modified accordingly.

Additionally, in lambda.ml, the apply_ctx function has been added to substitute the name of the definition with its value. The eval and eval1 functions have also been modified to include the context and a case for TmVar to retrieve the value associated with the definition.

main.ml has been modified to include the new context.

In the parser, the grammar axiom now distinguishes between whether it is an assignment or simply a term for proper analysis.

Examples:

```
x = 5;;

op = letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  sum;;

op x x;;

id = lambda x : Bool. x

id x;;
```

5. STRING TYPE

Incorporated a new type named TyString and introduced a corresponding term, TmString. To seamlessly integrate the String type, essential pattern matching adjustments were made to functions in the lambda.ml file. To ensure recognition of the String type by the lexer.mll and parser.mly, new rules were added along with their appropriate definitions.

Furthermore, three new functions—concat, length, and compare—have been implemented. For these functions, corresponding terms TmConcat, TmLength, and TmCompare were created. Consequently, lambda.ml was modified to accommodate these changes. In eval1, the following implementations were added:

- **concat:** Concatenates two strings.
- **length:** Returns the length of a string.
- **compare:** Compares two strings, returning 0 if they have the same length, 1 if the first is longer, and 2 if the second is longer.

Examples:

```
"";;
"abc";;
concat "para" "sol";;
concat (concat "para" "sol") "es";
lambda s : String. s;;
(lambda s : String.s) "abc";;
letrec replicate : String -> Nat -> String =
  lambda s : String. lambda n : Nat.
        if iszero n then "" else concat s (replicate s (pred n))
  in
  replicate "abc" 3
;;
concat ((lambda s : String.s) "abc") "de";;
let s = letrec replicate : String -> Nat -> String =
  lambda s : String. lambda n : Nat.
        if iszero n then "" else concat s (replicate s (pred n))
  in
  replicate "abc" 3
in concat s s;;
;;
```