PlcParserAux.fs

In the file PlcParserAux, I wrote the 4 functions which will be used in the Parser. makeFun function and makeAnon function are used in the production rules for function definition and they both use the auxiliary functions makeType and make makeFunAux.

PlcParserAux.fs | PlcLexer.fsl | PlcChecker.fs | runtests.fsx | PlcParser.fsy | PlcInterp.fs | Test.fs

```
1  module ParAux
2
3  open Absyn
4
5  let l =  "$list"
6  let elist =  "()"
7
8  let rec makeFunAux (n: int) (xs: (plcType * string) list) (e: expr) : expr =
9      match xs with
10     | [] -> e // replace with your implementation
11     | (t, x) :: [] -> Let(x, Item(n, Var l), e)
12     | (t, x) :: xt -> Let(x, Item(n,Var l), (makeFunAux (n+1) xt e))
13
14 let makeType (args: (plcType * string) list): plcType =
15     ListT (List.map (fun (x,y) -> x) args) // TODO
16
17 let makeFun (f: string) (xs: (plcType * string) list) (rt: plcType) (e1: expr) (e2: expr) : expr =
18     match xs with
19     | []          -> Letrec (f, ListT [], elist, rt, e1, e2)
20     | (t, x) :: [] -> Letrec (f, t, x, rt, e1, e2)
21     | _           ->
22       let t = makeType xs in
23       let e1' = makeFunAux 1 xs e1 in
24       Letrec(f, t, l, rt, e1', e2)
25
26 let makeAnon (xs: (plcType * string) list) (e: expr) : expr =
27     match xs with
28     | []          -> Anon (ListT [], elist, e)
29     | (t, x) :: [] -> Anon (t, x, e)
30     | _           ->
31       let t = makeType xs in
32       let e' = makeFunAux 1 xs e in
33       Anon (t, l, e')
34
```

PlcParser.fsy

In the file PlcParser.fsy which contains the parser for the PLC language, the parser uses the abstract syntax which defined in the file Absyn.fs and the function makeFun and makeAnon which defined in the file PlcParserAux.fs. I added all the tokens which can be used in the Lexer on the top of the file and wrote the whole production rules which describe the concrete syntax according to the requirements.

```
/*      Nil  Bool  Int   ->     */
%token  NIL  BOOL  INT  ARROW

/*      =>      fn  end  */
%token  DARROW  FN  END

/*      var  fun  rec  */
%token  VAR  FUN  REC

/*      if  then  else  */
%token  IF  THEN  ELSE

/*      match  with  |     _           */
%token  MATCH  WITH  PIPE  UNDERSCORE

/*      !    &&   */
%token  NOT  AND

/*      ::    hd    tl    ise  */
%token  CONS  HEAD  TAIL  ISE

/*      print  */
%token  PRINT

/*      +     -      *      /     */
%token  PLUS  MINUS  TIMES  DIV

/*      =   !=   <   <=   */
%token  EQ  NEQ  LT  LTE

/*      (     )     {       }       [       ]       */
%token  LPAR  RPAR  LBRACE  RBRACE  LBRACK  RBRACK

/*      ,      :      ;      */
%token  COMMA  COLON  SEMIC

%token EOF
```

```
Main:
    Expr                              { $1 }
  | AppExpr                           { $1 }
  | VAR NAME EQ Expr SEMIC Main       { Let($2, $4, $6)                      }
  | FUN REC NAME Args RetType EQ Expr SEMIC Main   { makeFun $3 $4 $5 $7 $9 }
  | FUN NAME Args EQ Expr SEMIC Main {Let($2,makeAnon $3 $5,$7) }
;

RetType:
  COLON Type  { $2 }
;


Args:
    LPAR RPAR           { []   }
  | LPAR Params RPAR    { $2   }          /* new rule */
;

/* Returns a list of variable/type pairs */
Params:
    TypedVar                { $1 :: [] }  /* new rule */
  | TypedVar COMMA Params   { $1 :: $3 }  /* new rule */
;

/* Returns a variable/type pair */
TypedVar:
    Type NAME  { ($1, $2) }      /* new rule */
;

Type:
    AtType              { $1       }
  | ListType            { ListT $1 }
  | LBRACK Type RBRACK  { SeqT $2  }
  | Type ARROW Type     { FunT( $1, $3) }
;
```

```
AtType:
    INT             { IntT     }
  | BOOL            { BoolT    }
  | NIL             { ListT [] }
  | LPAR Type RPAR  { $2       }
;

ListType:
  | AtType COMMA AtType    { [$1; $3] }
  | AtType COMMA ListType  { $1 :: $3 }
;

Expr:
    AtExpr                       { $1                    }
  | AppExpr                      { $1                    }
  | IF Expr THEN Expr ELSE Expr  { If ($2, $4, $6)       }
  | MATCH Expr WITH MatchExpr    { Match($2,$4)     }
  | NOT   Expr                   { Prim1 ("!", $2)       }
  | MINUS Expr                   { Prim1 ("-", $2)       }
  | HEAD Expr                    { Prim1 ("hd", $2)      }
  | TAIL Expr                    { Prim1 ("tl", $2)      }
  | ISE  Expr                    { Prim1 ("ise", $2)     }
  | PRINT Expr                   { Prim1 ("print", $2)   }
  | Expr PLUS  Expr              { Prim2 ("+",  $1, $3)  }
  | Expr MINUS Expr              { Prim2 ("-",  $1, $3)  }
  | Expr TIMES Expr              { Prim2 ("*",  $1, $3)  }
  | Expr DIV   Expr              { Prim2 ("/",  $1, $3)  }
  | Expr EQ    Expr              { Prim2 ("=",  $1, $3)  }
  | Expr NEQ   Expr              { Prim2 ("!=",  $1, $3) }
  | Expr LT    Expr              { Prim2 ("<",  $1, $3)  }
  | Expr LTE   Expr              { Prim2 ("<=", $1, $3)  }
  | Expr CONS  Expr              { Prim2 ("::", $1, $3)  }
  | Expr SEMIC Expr              { Prim2 (";", $1, $3)   }
  | Expr AND Expr                { Prim2 ("&&", $1, $3)  }
  | Expr LBRACK CSTINT RBRACK    { Item ($3, $1)         }
;
```

```
AtExpr:
    Const              { $1      }
  | NAME               { Var $1  }
  | LBRACE Main RBRACE  { $2      }
  | LPAR Expr RPAR     { $2      }
  | LPAR Comps RPAR    { List $2 }
  | FN Args DARROW Expr END { makeAnon $2 $4 }
;


AppExpr:
    AtExpr AtExpr   { Call  ($1, $2) }
  | AppExpr AtExpr  { Call  ($1, $2) }
;

Const:
    CSTBOOL  { ConB ($1) }
  | CSTINT   { ConI ($1) }
  | LPAR RPAR          { List [] }
  | LPAR  LBRACK Type RBRACK LBRACK RBRACK RPAR {ESeq(SeqT $3)}

;

Comps:
    Expr COMMA Expr   { [$1; $3] }
  | Expr COMMA Comps  { $1 :: $3 }
;



Types:
    Type COMMA Type { [$1; $3] }
  | Type COMMA Types { $1 :: $3 }
;


MatchExpr:
    END { [] }
  | PIPE Expr ARROW Expr MatchExpr  {(Some($2),$4)::$5}
  | PIPE UNDERSCORE ARROW Expr MatchExpr {(None,$4)::$5}
;
```

PlcLexer.fsl

In the file PlcLexer.fsl which contains the Lexer for the PLC language, I added different keywords and tokens.

```
let keyword s =
  match s with
   | "Bool"  -> BOOL
   | "else"  -> ELSE
   | "end"   -> END
   | "false" -> CSTBOOL false
   | "fn"    -> FN
   | "fun"   -> FUN
   | "hd"    -> HEAD
   | "if"    -> IF
   | "Int"   -> INT
   | "ise"   -> ISE
   | "match" -> MATCH
   | "Nil"   -> NIL
   | "print" -> PRINT
   | "rec"   -> REC
   | "then"  -> THEN
   | "tl"    -> TAIL
   | "var"   -> VAR
   | "true"  -> CSTBOOL true
   | "with"  -> WITH
   | _       -> NAME s
}
```

```
 rule Token = parse
    | "->"            { ARROW }
    | "=>"            { DARROW }
    | "|"             { PIPE }
    | "&&"            { AND }
    | "::"            { CONS }
```

```
| '}'               { RBRACE }
| '['               { LBRACK }
| ']'               { RBRACK }
| ';'               { SEMIC }
| ','               { COMMA }
| ':'               { COLON }
| eof               { EOF }
| _                 { UNDERSCORE }
```

PlcInterp.fs

I wrote the whole file PlcInterp.fs which contains an interpreter. The interpreter converts expr terms to plcValue terms. It offers a function eval : expr -> plcValue env -> plcValue that evaluates the value of a well-typed expression e in a value environment which is for the free variables of e. The interpreter supports the following types:
Nil type; Boolean type; integer type; List types; Function types; Sequence types and Equality types.

PlcChecker.fs

I wrote the whole file PlcChecker.fs which contains a type checker. It offers a function teval : expr -> plcType env -> plcType which can check whether the type is correct or not. The type checker should return an error with failwith in the following cases: undefined operators; functions applied to an argument whose type that differs from the declared one; function definitions whose body has a type that differs from the function's declared return type; terms whose types are different; The equality expressions whose types are different.