

# An Interpreter for a Simple Functional Language (LET)

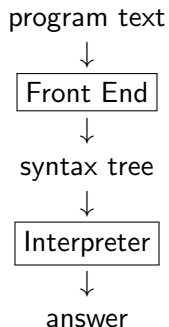
CS510

# Expressions and Interpreters

1. Compiler vs Interpreter
2. A simple programming language: LET
3. Specification and Evaluation

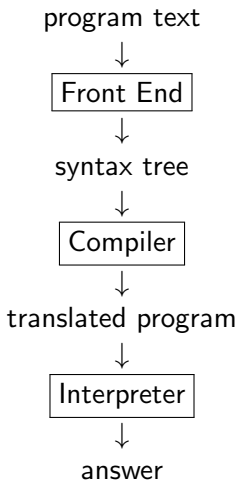
# Compiler vs Interpreter

## Execution via Interpreter



# Compiler vs Interpreter

## Execution via Compiler



# Compiler vs Interpreter

Both cases need a front end. The front end has two phases:

# Compiler vs Interpreter

Both cases need a front end. The front end has two phases:

program text (list of characters)



Scanner



list of tokens

# Compiler vs Interpreter

Both cases need a front end. The front end has two phases:

program text (list of characters)



Scanner



list of tokens



Parser



syntax tree

# LET: A Simple Language – The Concrete Syntax

$\langle Program \rangle ::= \langle Expression \rangle$



# LET: A Simple Language – The Concrete Syntax

$\langle Program \rangle ::= \langle Expression \rangle$

$\langle Expression \rangle ::= \langle Number \rangle$

# LET: A Simple Language – The Concrete Syntax

$\langle Program \rangle ::= \langle Expression \rangle$

$\langle Expression \rangle ::= \langle Number \rangle$

$\langle Expression \rangle ::= \langle Identifier \rangle$

# LET: A Simple Language – The Concrete Syntax

$\langle Program \rangle ::= \langle Expression \rangle$

$\langle Expression \rangle ::= \langle Number \rangle$

$\langle Expression \rangle ::= \langle Identifier \rangle$

$\langle Expression \rangle ::= \langle Expression \rangle - \langle Expression \rangle$

# LET: A Simple Language – The Concrete Syntax

$\langle \textit{Program} \rangle ::= \langle \textit{Expression} \rangle$

$\langle \textit{Expression} \rangle ::= \langle \textit{Number} \rangle$

$\langle \textit{Expression} \rangle ::= \langle \textit{Identifier} \rangle$

$\langle \textit{Expression} \rangle ::= \langle \textit{Expression} \rangle - \langle \textit{Expression} \rangle$

$\langle \textit{Expression} \rangle ::= \text{zero? } (\langle \textit{Expression} \rangle)$

# LET: A Simple Language – The Concrete Syntax

$\langle Program \rangle ::= \langle Expression \rangle$

$\langle Expression \rangle ::= \langle Number \rangle$

$\langle Expression \rangle ::= \langle Identifier \rangle$

$\langle Expression \rangle ::= \langle Expression \rangle - \langle Expression \rangle$

$\langle Expression \rangle ::= \text{zero? } (\langle Expression \rangle)$

$\langle Expression \rangle ::= \text{if } \langle Expression \rangle$   
                          then  $\langle Expression \rangle$  else  $\langle Expression \rangle$

# LET: A Simple Language – The Concrete Syntax

$\langle \text{Program} \rangle ::= \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Number} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Identifier} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle - \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \text{zero? } (\langle \text{Expression} \rangle)$

$\langle \text{Expression} \rangle ::= \text{if } \langle \text{Expression} \rangle$   
                          then  $\langle \text{Expression} \rangle$  else  $\langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \text{let } \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle \text{ in } \langle \text{Expression} \rangle$

# LET: A Simple Language – The Concrete Syntax

$\langle \text{Program} \rangle ::= \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Number} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Identifier} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle - \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \text{zero? } (\langle \text{Expression} \rangle)$

$\langle \text{Expression} \rangle ::= \text{if } \langle \text{Expression} \rangle$   
                          then  $\langle \text{Expression} \rangle$  else  $\langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \text{let } \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle \text{ in } \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= (\langle \text{Expression} \rangle)$

# LET: A Simple Language – The Concrete Syntax

$\langle \text{Program} \rangle ::= \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Number} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Identifier} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle - \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \text{zero? } (\langle \text{Expression} \rangle)$

$\langle \text{Expression} \rangle ::= \text{if } \langle \text{Expression} \rangle$   
                          then  $\langle \text{Expression} \rangle$  else  $\langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \text{let } \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle \text{ in } \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= (\langle \text{Expression} \rangle)$



# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶ `x`

Non-examples

# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶  $x$
- ▶  $55 - (x - 11)$

Non-examples

# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶  $x$
- ▶  $55 - (x - 11)$
- ▶  $\text{zero? } (55 - (x - 11))$

Non-examples

# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶  $x$
- ▶  $55 - (x - 11)$
- ▶  $\text{zero? } (55 - (x - 11))$
- ▶  $\text{let } y = 23 \text{ in if zero?}(y) \text{ then } 4 \text{ else } 6$

Non-examples

# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶  $x$
- ▶  $55 - (x - 11)$
- ▶  $\text{zero? } (55 - (x - 11))$
- ▶  $\text{let } y = 23 \text{ in if zero?}(y) \text{ then } 4 \text{ else } 6$

Non-examples

- ▶  $(\text{zero? } 55 - (x - 11))$

# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶  $x$
- ▶  $55 - (x - 11)$
- ▶  $\text{zero? } (55 - (x - 11))$
- ▶  $\text{let } y = 23 \text{ in if zero?}(y) \text{ then } 4 \text{ else } 6$

Non-examples

- ▶  $(\text{zero? } 55 - (x - 11))$
- ▶  $\text{zero } 4$

# Examples of Programs in LET

Examples of programs in concrete syntax:

- ▶  $x$
- ▶  $55 - (x - 11)$
- ▶  $\text{zero? } (55 - (x - 11))$
- ▶  $\text{let } y = 23 \text{ in if zero?}(y) \text{ then } 4 \text{ else } 6$

Non-examples

- ▶  $(\text{zero? } 55 - (x - 11))$
- ▶  $\text{zero } 4$
- ▶  $1 + + 2$

# LET: Abstract Syntax

```
1 type prog = AProg of expr
2
3 type expr =
4   (* ... continues in next slide ... *)
```



# LET: Abstract Syntax (cont.)

```
1  type expr =  
2    | Var of string  
3    | Int of int  
4    | Sub of expr*expr  
5    | Let of string*expr*expr  
6    | App of expr*expr  
7    | IsZero of expr  
8    | ITE of expr*expr*expr
```

# Examples in Abstract Syntax

Let's revisit our earlier examples to translate them into the corresponding abstract syntax trees.

- ▶ Concrete syntax:

$55 - (x - 11)$

- ▶ Abstract syntax (type prog):

```
AProg  
  (Sub (Int 55, Sub (Var "x", Int 11)))
```

# Examples in Abstract Syntax

- Concrete syntax:

zero? (55 - (x - 11))

- Abstract syntax

```
1 AProg
2   (IsZero (Sub (Int 55, Sub (Var "x", Int 11))))
```

- Exercise: write the abstract syntax tree for this LET expression:

let y = 23 in if zero?(y) then 4 else 6

# Interpreter for Expressions

- ▶ The next step is to define an interpreter for expressions

`eval_expr: expr -> ???`

- ▶ What should the return type of the interpreter be?
- ▶ Since we can write programs such as 2 and zero?(4) then either a number or a boolean
  - ▶ At least for now
- ▶ Let us define a new type for the return type of the interpreter that includes constructors for these two cases
- ▶ We'll call it the type of **Expressed Values**

# Interpreter for Expressions – The Need for Environments

- ▶ Now that we know the type of the interpreter for expressions

`eval_expr: expr -> exp_val`

we must move on to defining the interpreter itself

- ▶ Before doing so, however, one final observation
- ▶ The value of `5-1` should clearly be `(NumVal 4)`
- ▶ That of `if zero?(4-4) then 2 else 1` should clearly be `2`
- ▶ What should the value of `x-2` be?

# Interpreter for Expressions – The Need for Environments

- ▶ What should the value of  $x-2$  be?
- ▶ We need the value of  $x$  to be able to answer
- ▶ Hence we need **environments**
- ▶ The final type of the interpreter is therefore

`eval_expr: env -> expr -> exp_val`

# Environments

```
1 type env =  
2   | EmptyEnv  
3   | ExtendedEnv of string*exp_val*env
```

## ► Two constructors

- EmptyEnv: constructs an empty environment
- ExtendedEnv: extends a previous environment with a new association pair

# Environment

- ▶ Function whose domain is a finite set of variables and whose range is the **denoted values**.
  - ▶ **Denoted Values**: values bound to variables.  
 $\text{DenVal} = \text{Int} + \text{Bool}$
  - ▶ For now they coincide with **Expressed Values**
  - ▶ So we'll just use the latter
- ▶  $\rho$  ranges over environments.
- ▶  $[]$  denotes the empty environment.

## Shorthands

- ▶  $[var = val]_{\rho}$  denotes  $\text{ExtendEnv}(var, val, \rho)$ .
- ▶  $[var_1 = val_1, var_2 = val_2]_{\rho}$  abbreviates  
 $[var_1 = val_1]([var_2 = val_2]_{\rho})$
- ▶  $[var_1 = val_1, var_2 = val_2, \dots]$  denotes the environment in which the value of  $var_1$  is  $val_1$ , etc.



## More Shorthands for Environments

```
[i = 1]
  [v = 5]
    [x = 10]
      []
```

abbreviates

```
ExtendEnv("i", NumVal 1,
  ExtendEnv("v", NumVal 5,
    ExtendEnv("x", NumVal 10,
      EmptyEnv)))
```

- We'll call this environment `init_env`

# Specifying the Behavior of the Interpreter for Expressions

- ▶ The value of a constant is the constant itself, as an expressed value

```
eval_expr  $\rho$  (Int  $n$ ) = NumVal  $n$ 
```

- ▶ We must lookup the value of variables in the environment

```
eval_expr  $\rho$  (Var  $var$ ) = apply_env  $\rho$   $var$ 
```

Note:

- ▶ This is **not** executable code (hence the shadow in the frame)
- ▶ It specifies the behavior of `eval_expr` in terms of equations
- ▶ On the next slide we show sample code for `eval_expr` that satisfies these equations

## Implementing the first cases of `eval_expr`

```
1 let rec eval_expr (en:env) (e:expr):exp_val =  
2   match e with  
3   | Int n          -> NumVal n  
4   | Var id         ->  
5     (match apply_env en id with  
6     | None -> failwith @@ "Variable "^id^" undefined"  
7     | Some ev -> ev)
```

## Specifying the Behavior of the Interpreter for Expressions

- ▶ Difference is computed by first computing the values of the arguments and then performing the difference itself

```
eval_expr  $\rho$  (Sub  $exp_1$   $exp_2$ )  
  = NumVal  
    (numVal_to_num (eval_expr  $\rho$   $exp_1$ )) -  
    (numVal_to_num (eval_expr  $\rho$   $exp_2$ ))
```

# Specifying the Behavior of the Interpreter for Expressions

```
eval_expr  $\rho$  (Sub  $exp_1$   $exp_2$ )  
  = NumVal  
    (numVal_to_num (eval_expr  $\rho$   $exp_1$ )) -  
    (numVal_to_num (eval_expr  $\rho$   $exp_2$ ))
```

## Code for Sub

```
1 let rec eval_expr (en:env) (e:expr) :exp_val =  
2   match e with  
3     ...  
4   | Sub(e1, e2)    ->  
5     let v1 = eval_expr en e1 in  
6     let v2 = eval_expr en e2 in  
7     NumVal ((numVal_to_num v1) - (numVal_to_num v2))
```

# Specifying the Behavior of the Interpreter for Programs

- ▶ A program is an expression that may contain free variables.
  - ▶ These represent the top-level declarations
- ▶ The value of the program is the value of the expression in a suitable environment.
- ▶ We assume the initial environment `init-env` defined above

```
eval_prog exp = eval_expr init_env exp
```

- ▶ This initial environment allows us to write examples involving variables  $i$ ,  $v$  and  $x$  in our program without having to declare them

## Specifying the Behavior of IsZero and ITE

```
1 eval_expr  $\rho$  (IsZero  $exp_1$ ) =  
2   BoolVal (zero? (expval->num (eval_expr  $\rho$   $exp_1$ )))  
3  
4 eval_expr  $\rho$  (ITE  $exp_1$   $exp_2$   $exp_3$ ) =  
5   if (boolVal_to_bool (eval_expr  $\rho$   $exp_1$ ))  
6   then eval_expr  $\rho$   $exp_2$   
7   else eval_expr  $\rho$   $exp_3$ 
```

## Specifying the Behavior of let

The right-hand side of the `let` is also an expression, so it can be arbitrarily complex. For example,

```
1 let x = 7
2 in let y = 2
3     in let y = let x = x-1
4               in x-y
5         in (x-8)-y
```

- ▶ Here the `x` declared on the third line is bound to 6
- ▶ So the value of `y` is 4
- ▶ The value of the entire expression is  $((-1) - 4) = -5$ .
- ▶ We can write down the specification as an equation.



# Behavior of the Interpreter for let-expressions

- Note how the body is evaluated in an extended environment

```
1 eval_expr  $\rho$  (Let var  $exp_1$  body) =  
2   let val=eval_expr  $\rho$   $exp_1$   
3   in eval_expr [var = val] $\rho$  body
```

- Important: static scoping is implemented by extending environments

# The Interpreter for LET

- ▶ Code available in Canvas Modules/Interpreters
- ▶ Directory `let-lang`
- ▶ Compile with `ocamlbuild -use-menhir interp.ml`
- ▶ Make sure the `.ocamlinit` file is in the folder of your sources
- ▶ Run `utop`
- ▶ Type, for eg., `Main.interp "let x=2 in x+3";;`