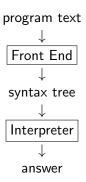
# An Interpreter for a Simple Functional Language (LET)CS496

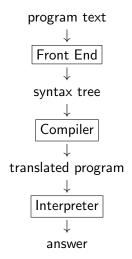
## Expressions and Interpreters

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

## **Execution via Interpreter**



#### **Execution via Compiler**

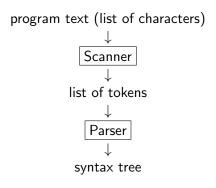


Both cases need a front end. The front end has two phases:

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



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



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

```
\langle Program \rangle ::= \langle Expression \rangle
\langle Expression \rangle ::= \langle Number \rangle
```

```
\langle Program \rangle ::= \langle Expression \rangle

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

\langle Expression \rangle ::= \langle Identifier \rangle
```

```
\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 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 ::= zero? (\langle Expression \rangle)
```

```
\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 ::= zero? (\langle Expression \rangle)
\langle Expression \rangle ::= if \langle Expression \rangle
                                   then \( \int Expression \)\) else \( \int Expression \)\
```

```
\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 ::= zero? (\langle Expression \rangle)
\langle Expression \rangle ::= if \langle Expression \rangle
                                     then \(\langle Expression \rangle \) else \(\langle Expression \rangle \)
\langle Expression \rangle ::= let \langle Identifier \rangle = \langle Expression \rangle in \langle Expression \rangle
```

```
\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 ::= zero? (\langle Expression \rangle)
\langle Expression \rangle ::= if \langle Expression \rangle
                                    then \(\langle Expression \rangle \) else \(\langle Expression \rangle \)
\langle Expression \rangle ::= let \langle Identifier \rangle = \langle Expression \rangle in \langle Expression \rangle
\langle Expression \rangle ::= (\langle Expression \rangle)
```

```
\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 ::= zero? (\langle Expression \rangle)
\langle Expression \rangle ::= if \langle Expression \rangle
                                    then \(\langle Expression \rangle \) else \(\langle Expression \rangle \)
\langle Expression \rangle ::= let \langle Identifier \rangle = \langle Expression \rangle in \langle Expression \rangle
\langle Expression \rangle ::= (\langle Expression \rangle)
```

## Examples of Programs in $\operatorname{LET}$

Examples of programs in concrete syntax:

X

Examples of programs in concrete syntax:

- X
- ▶ 55 (x 11)

Examples of programs in concrete syntax:

- X
- ▶ 55 (x 11)
- ▶ zero? (55 (x 11))

Examples of programs in concrete syntax:

- X
- ▶ 55 (x 11)
- ► zero? (55 (x 11))
- let y = 23 in if zero?(y) then 4 else 6

### Examples of programs in concrete syntax:

- X
- ▶ 55 (x 11)
- zero? (55 (x 11))
- let y = 23 in if zero?(y) then 4 else 6

#### Non-examples

► (zero? 55 - (x - 11))

#### Examples of programs in concrete syntax:

- X
- ▶ 55 (x 11)
- zero? (55 (x 11))
- let y = 23 in if zero?(y) then 4 else 6

- ► (zero? 55 (x 11))
- zero 4

## Examples of programs in concrete syntax:

- X
- ▶ 55 (x 11)
- zero? (55 (x 11))
- let y = 23 in if zero?(y) then 4 else 6

- ► (zero? 55 (x 11))
- zero 4
- **▶** 1 + + 2

## LET: Abstract Syntax

```
type prog = AProg of expr

type expr =
(* ... continues in next slide ... *)
```

# LET: Abstract Syntax (cont.)

```
type expr =
lint of string
lint of int
lint of expr*expr
let of string*expr*expr
lisZero of expr
litt of expr*expr
```

## Examples in Abstract Syntax

Let't revisit our earlier examples to translate them into the corresponding abstract syntax trees.

Concrete syntax:

Abstract syntax (type prog):

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

## Examples in Abstract Syntax

Concrete syntax:

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**

```
type env =
| EmptyEnv
| 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.
     DenVal = Int + Bool
  - ► For now they coincide with Expressed Values
  - So we'll just use the latter
- ightharpoonup 
  ho ranges over environments.
- [] denotes the empty environment.

#### Shorthands

- $[var = val]_{\rho}$  denotes ExtendEnv(var,val, $\rho$ ).
- $[var_1 = val_1, var_2 = val_2]_{\rho} \text{ abbreviates}$   $[var_1 = val_1]([var_2 = val_2]_{\rho})$
- ▶  $[var_1 = val_1, var_2 = val_2, ...]$  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) = n
```

▶ We must lookup the value of variables in the environment

```
eval\_expr \rho (Var var) = \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

## 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 
ho (Sub exp_1 exp_2) = (eval_expr 
ho exp_1) -- (eval_expr 
ho exp_2)
```

Operation "--" checks if its arguments are numbers and, if so, subtracts them

# Specifying the Behavior of the Interpreter for Expressions

#### Code for Sub

## 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 ρ (IsZero exp<sub>1</sub>) =
2          (eval_expr ρ exp<sub>1</sub>) == 0
3
4  eval_expr ρ (ITE exp<sub>1</sub> exp<sub>2</sub> exp<sub>3</sub>) =
5          if (eval_expr ρ exp<sub>1</sub>)
6          then eval_expr ρ exp<sub>2</sub>
7          else eval_expr ρ exp<sub>3</sub>
```

## Specifying the Behavior of let

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

- ▶ 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., Interp.interp "let x=2 in x+3";;.