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

Compiler vs Interpreter

Execution via Interpreter

program text

$\downarrow$

$\boxed{\text{Front End}}$

$\downarrow$

syntax tree

$\downarrow$

$\boxed{\text{Interpreter}}$

$\downarrow$

answer

Compiler vs Interpreter

Execution via Compiler

program text

$\downarrow$

$\boxed{\text{Front End}}$

$\downarrow$

syntax tree

$\downarrow$

$\boxed{\text{Compiler}}$

$\downarrow$

translated program

$\downarrow$

$\boxed{\text{Interpreter}}$

$\downarrow$

answer

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:

$$\begin{array}{c}\text{program text (list of characters)}\\ \downarrow \\ \boxed{\text{Scanner}} \\ \downarrow \\ \text{list of tokens}\end{array}$$

Compiler vs Interpreter

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

program text (list of characters)

$\downarrow$

$\boxed{\text{Scanner}}$

$\downarrow$

list of tokens

$\downarrow$

$\boxed{\text{Parser}}$

$\downarrow$

syntax tree

LET: A Simple Language – The Concrete Syntax

$$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$$

LET: A Simple Language – The Concrete Syntax

$$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Number} \rangle$$

LET: A Simple Language – The Concrete Syntax

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

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

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

LET: A Simple Language – The Concrete Syntax

$$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Number} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Identifier} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle \texttt{ - } \langle \mathit{Expression} \rangle$$

LET: A Simple Language – The Concrete Syntax

$$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Number} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Identifier} \rangle$$

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

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

LET: A Simple Language – The Concrete Syntax

$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$

$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Number} \rangle$

$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Identifier} \rangle$

$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$ - $\langle \mathit{Expression} \rangle$

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

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

LET: A Simple Language – The Concrete Syntax

$$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Number} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Identifier} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle \texttt{ - } \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \mathsf{zero?}\ (\langle \mathit{Expression} \rangle)$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \mathsf{if}\ \langle \mathit{Expression} \rangle\ \mathsf{then}\ \langle \mathit{Expression} \rangle\ \mathsf{else}\ \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \mathsf{let}\ \langle \mathit{Identifier} \rangle = \langle \mathit{Expression} \rangle\ \mathsf{in}\ \langle \mathit{Expression} \rangle$$

LET: A Simple Language – The Concrete Syntax

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

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

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

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

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

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

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

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

LET: A Simple Language – The Concrete Syntax

$$\langle \mathit{Program} \rangle \quad ::= \quad \langle \mathit{Expression} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Number} \rangle$$

$$\langle \mathit{Expression} \rangle \quad ::= \quad \langle \mathit{Identifier} \rangle$$

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

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

$$\langle \mathit{Expression} \rangle \quad ::= \quad \begin{array}{l} \text{if } \langle \mathit{Expression} \rangle \\ \text{then } \langle \mathit{Expression} \rangle \text{ else } \langle \mathit{Expression} \rangle \end{array}$$

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

$$\langle \mathit{Expression} \rangle \quad ::= \quad (\langle \mathit{Expression} \rangle)$$

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

Non-examples

Examples of Programs in LET

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

Examples of Programs in LET

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 LET

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))
- zero 4

Examples of Programs in LET

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))
- zero 4
- 1 + + 2

LET: Abstract Syntax

```
type prog = AProg of expr

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

LET: Abstract Syntax (cont.)

```
type expr =
  | Var of string
  | Int of int
  | Sub of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
```

Examples in Abstract Syntax

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

- Concrete syntax:

  $$55 \text{ -(x-11)}$$

- 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

```
AProg
    (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.
    $$\text{DenVal} = \text{Int} + \text{Bool}$$
  - For now they coincide with Expressed Values
  - So we'll just use the latter
- ρ ranges over environments.
- $[\,]$ denotes the empty environment.

Shorthands

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

More Shorthands for Environments

$$
\begin{array}{l}
[i = 1] \\
\quad [v = 5] \\
\quad\quad [x = 10] \\
\quad\quad\quad [\,]
\end{array}
$$

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 ρ (Int n) = n
```

- We must lookup the value of variables in the environment

```
eval_expr ρ (Var var) = ρ(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`

```
let rec eval_expr (en:env) (e:expr):exp_val =
  match e with
  | Int n          -> NumVal n
  | Var id          ->
    (match apply_env en id with
    | None -> failwith @@ "Variable "^id^" undefined"
    | 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

$$
\begin{array}{l}
\texttt{eval\_expr}\ \rho\ (\texttt{Sub}\ exp_1\ exp_2) \\
\quad = (\texttt{eval\_expr}\ \rho\ exp_1)\ \texttt{--}\ (\texttt{eval\_expr}\ \rho\ exp_2)
\end{array}
$$

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

Specifying the Behavior of the Interpreter for Expressions

```
eval_expr ρ (Sub exp1 exp2)
  = (eval_expr ρ exp1) -- (eval_expr ρ exp2)
```

Code for `Sub`

```
let rec eval_expr (en:env) (e:expr) :exp_val =
    match e with
     ...
    | Sub(e1, e2)     ->
      let v1 = eval_expr en e1 in
      let v2 = eval_expr en e2  in
      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`

```
eval_expr ρ (IsZero exp1) =
     (eval_expr ρ exp1)==0

eval_expr  ρ (ITE exp1 exp2 exp3) =
   if (eval_expr ρ exp1)
   then eval_expr ρ exp2
   else eval_expr ρ exp3
```

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

```
eval_expr ρ (Let var exp₁ body) =
    let val=eval_expr ρ exp₁
    in eval_expr [var = val]ρ 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";;`.