An Interpreter for a Simple Functional Language (LET)

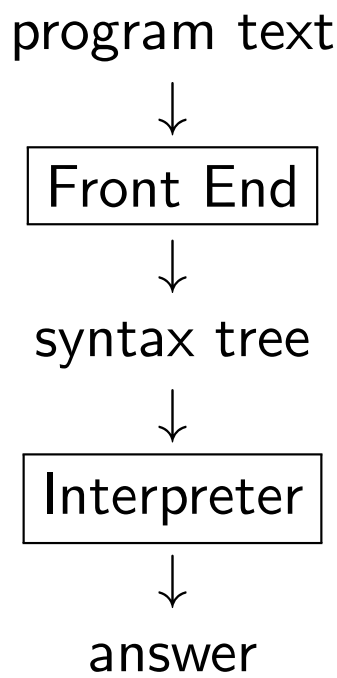
CS510

Expressions and Interpreters

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

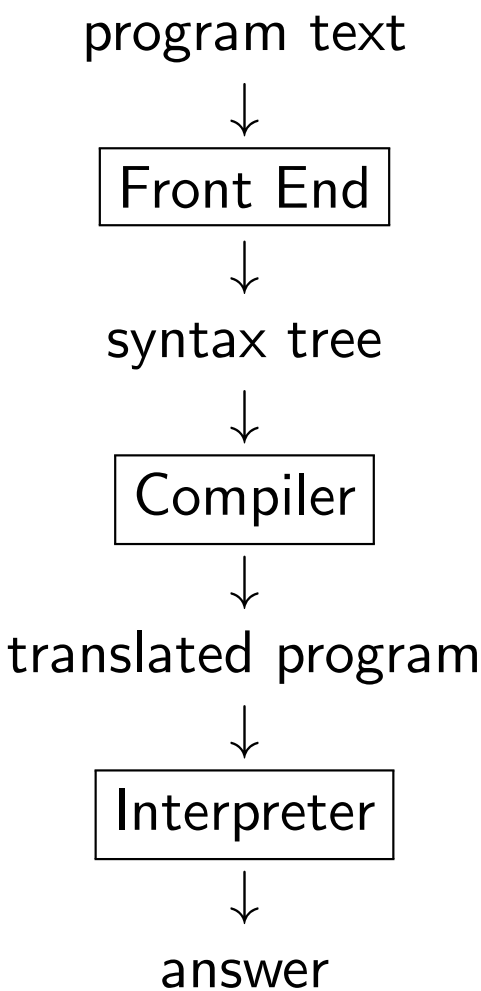
Compiler vs Interpreter

Execution via Interpreter



Compiler vs Interpreter

Execution via Compiler

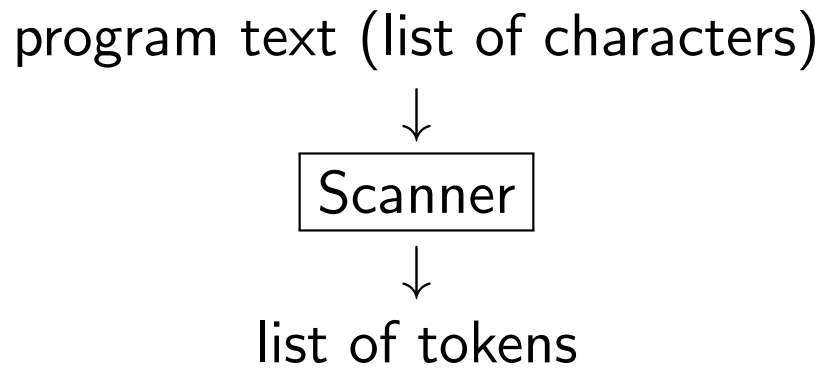


Compiler vs Interpreter

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

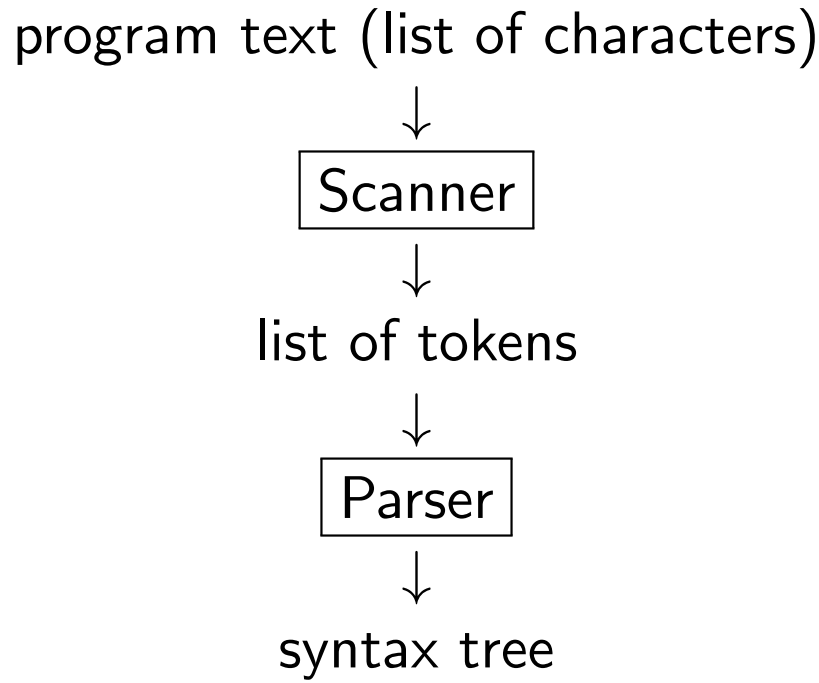
Compiler vs Interpreter

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



Compiler vs Interpreter

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



LET: A Simple Language – The Concrete Syntax

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

LET: A Simple Language – The Concrete Syntax

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

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

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

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{- (} \langle \textit{Expression} \rangle \text{ , } \langle \textit{Expression} \rangle \text{)}
\end{array}
$$

LET: A Simple Language – The Concrete Syntax

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{- } (\langle \textit{Expression} \rangle \text{ , } \langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{zero? } (\langle \textit{Expression} \rangle)
\end{array}
$$

LET: A Simple Language – The Concrete Syntax

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{- } (\langle \textit{Expression} \rangle \text{ , } \langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{zero? } (\langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{if } \langle \textit{Expression} \rangle \\
 & & \text{then } \langle \textit{Expression} \rangle \text{ else } \langle \textit{Expression} \rangle
\end{array}
$$

LET: A Simple Language – The Concrete Syntax

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{- } (\langle \textit{Expression} \rangle \text{ , } \langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{zero? } (\langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{if } \langle \textit{Expression} \rangle \\
 & & \text{then } \langle \textit{Expression} \rangle \text{ else } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{let } \langle \textit{Identifier} \rangle = \langle \textit{Expression} \rangle \text{ in } \langle \textit{Expression} \rangle
\end{array}
$$

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

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
 ;;  ... continues in next slide ...
```

LET: Abstract Syntax (cont.)

```
(define-datatype expression expression?
 (const-exp (num number?))
 (var-exp (var symbol?))
 (diff-exp
   (exp1 expression?)
   (exp2 expression?))
 (zero?-exp
   (exp1 expression?))
 (if-exp
   (exp1 expression?)
   (exp2 expression?)
   (exp3 expression?))
 (let-exp
   (var symbol?)
   (exp1 expression?)
   (body expression?)))
```

LET: Abstract Syntax (cont.)

```
1  (define-datatype expression expression?
2   (const-exp (num number?))
3   (var-exp (var symbol?))
4   (diff-exp
5     (exp1 expression?)
6     (exp2 expression?))
7   (zero?-exp
8     (exp1 expression?))
9   (if-exp
10    (exp1 expression?)
11    (exp2 expression?)
12    (exp3 expression?))
13  (let-exp
14    (var symbol?)
15    (exp1 expression?)
16    (body expression?)))
```

LET: Abstract Syntax (cont.)

```
1  (define-datatype expression expression?
2   (const-exp (num number?))
3   (var-exp (var symbol?))
4   (diff-exp
5     (exp1 expression?)
6     (exp2 expression?))
7   (zero?-exp
8     (exp1 expression?))
9   (if-exp
10    (exp1 expression?)
11    (exp2 expression?)
12    (exp3 expression?))
13  (let-exp
14    (var symbol?)
15    (exp1 expression?)
16    (body expression?)))
```

LET: Abstract Syntax (cont.)

```
(define-datatype expression expression?
 (const-exp (num number?))
 (var-exp (var symbol?))
 (diff-exp
   (exp1 expression?)
   (exp2 expression?))
 (zero?-exp
   (exp1 expression?))
 (if-exp
   (exp1 expression?)
   (exp2 expression?)
   (exp3 expression?))
 (let-exp
   (var symbol?)
   (exp1 expression?)
   (body expression?)))
```

LET: Abstract Syntax (cont.)

```
(define-datatype expression expression?
 (const-exp (num number?))
 (var-exp (var symbol?))
 (diff-exp
   (exp1 expression?)
   (exp2 expression?))
 (zero?-exp
   (exp1 expression?))
 (if-exp
   (exp1 expression?)
   (exp2 expression?)
   (exp3 expression?))
 (let-exp
   (var symbol?)
   (exp1 expression?)
   (body expression?)))
```

LET: Abstract Syntax (cont.)

```
1  (define-datatype expression expression?
2   (const-exp (num number?))
3   (var-exp (var symbol?))
4   (diff-exp
5     (exp1 expression?)
6     (exp2 expression?))
7   (zero?-exp
8     (exp1 expression?))
9   (if-exp
10    (exp1 expression?)
11    (exp2 expression?)
12    (exp3 expression?))
13  (let-exp
14    (var symbol?)
15    (exp1 expression?)
16    (body expression?)))
```

LET: Abstract Syntax (cont.)

```
(define-datatype expression expression?
 (const-exp (num number?))
 (var-exp (var symbol?))
 (diff-exp
   (exp1 expression?)
   (exp2 expression?))
 (zero?-exp
   (exp1 expression?))
 (if-exp
   (exp1 expression?)
   (exp2 expression?)
   (exp3 expression?))
 (let-exp
   (var symbol?)
   (exp1 expression?)
   (body expression?)))
```

Examples in Abstract Syntax

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

- Concrete syntax:

  -(55, -(x, 11))

- Abstract syntax:

```
(a-program
 (diff-exp
  (const-exp 55)
  (diff-exp
    (var-exp x)
    (const-exp 11))))
```

Examples in Abstract Syntax

- Concrete syntax:

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

- Abstract syntax

```
(a-program
  (zero?-exp
    (diff-exp
      (const-exp 55)
      (diff-exp
        (var-exp x)
        (const-exp 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

  `value-of: expression ->` ???

- 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

Specification of Values (1/2)

Expressed Values: possible values of expressions.

$$ExpVal = Int + Bool$$

- Here is a datatype declaration for ExpVal

```
1 (define-datatype expval expval?
2    (num-val
3      (value number?))
4    (bool-val
5      (boolean boolean?)))
```

- Example expressions of this type are `(num-val 2)` and `(bool-val #t)`

Specification of Values (2/2)

Interface for the datatype of expressed values

`num-val`	:	Int	$\rightarrow$	ExpVal
`bool-val`	:	Bool	$\rightarrow$	ExpVal
`expval->num`	:	ExpVal	$\rightarrow$	Int
`expval->bool`	:	ExpVal	$\rightarrow$	Bool

```
1 ;; expval->num : ExpVal -> Int
2 ;; Page: 70
3 (define expval->num
4   (lambda (v)
5     (cases expval v
6       (num-val (num) num)
7       (else (expval-extractor-error 'num v)))))
```

Scheme Implementation

```
;; expval->bool : ExpVal -> Bool
;; Page: 70
(define expval->bool
  (lambda (v)
    (cases expval v
      (bool-val (bool) bool)
      (else (expval-extractor-error 'bool v)))))

(define expval-extractor-error
  (lambda (variant value)
    (eopl:error 'expval-extractors
                "Looking for a ~s, found ~s"
                 variant value)))
```

Interpreter for Expressions – The Need for Environments

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

  ```
  value-of: expression -> expval
  ```

  we must move on to defining the interpreter itself
- Before doing so, however, one final observation
- The value of `-(5,1)` should clearly be `(num-val 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

```
value-of: { expression, environment } -> expval
```

Environments

```
1 (define-datatype environment environment?
2   (empty-env-record)
3   (extended-env-record
4     (syms (list-of symbol?))
5     (vals (list-of expval?))
6     (env environment?)))
```

- Two constructors
  - `empty-env-record`: constructs an empty environment
  - `extended-env-record`: 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 `(extend-env var val` ρ`)`.
- $[var_1 = val_1, var_2 = val_2]_\rho$ 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

$$[i = 1]$$
$$\quad [v = 5]$$
$$\quad\quad [x = 10]$$
$$\quad\quad\quad [\,]$$

abbreviates

```
1 (extend-env 'i (num-val 1)
2  (extend-env 'v (num-val 5)
3   (extend-env 'x (num-val 10)
4    (empty-env))))))
```

- 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

```
(value-of (const-exp n) ρ) = (num-val n)
```

- We must lookup the value of variables in the environment

```
(value-of (var-exp var) ρ) = (apply-env ρ var)
```

Note:

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

Implementing the first cases of `value-of`

```
(define value-of
    (lambda (exp env)
      (cases expression exp
        ;; corresponds to equation for constant
    expressions
        (const-exp (num) (num-val num))
        ;; corresponds to equations for
    variables
        (var-exp (var) (apply-env env var))

        ;; more cases still to come ....

 )))
```

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

```
1 (value-of (diff-exp exp1 exp2) ρ)
2   = (num-val
3       (-
4         (expval->num (value-of exp1 ρ))
5         (expval->num (value-of exp2 ρ))))
```

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

```
1 (value-of-program exp) = (value-of exp init-env)
```

- 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 zero? and if-exp

```
1 (value-of  (zero?-exp exp1) ρ) =
2     (bool-val
3         (zero? (expval->num (value-of exp1 ρ))))
4
5 (value-of (if-exp exp1 exp2 exp3) ρ) =
6    (if (expval->bool (value-of exp1 ρ))
7          (value-of (exp2 ρ))
8          (value-of (exp3 ρ))
9
```

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 (value-of (let-exp var exp1 body) ρ) =
2      (let ((val (value-of exp1 ρ)))
3           (value-of body [var = val]ρ))
```

- Important: static scoping is implemented by extending environments

The Interpreter for LET

- Code available from `http://www.eopl3.com`
- Directory `chapter3/let-lang`
- Open `top.scm` in Racket
- There are a number of tests in `tests.scm`
- You can run them with `run-one`. Eg.

```
> (run-one 'simple-let-1)
(num-val 3)
```