

Project  
 SNU 4910.210, Programming Principles Fall 2020  
 Chung-Kil Hur  
**due: 12/19(Sat) 23:59**

**Problem 1 (50 Points)** In Scala, implement an interpreter `interp` for the programming language E given below.

`interp : E → V`

$A ::=$	$x$	call by value
$B ::=$	$(\text{def } f (A^*) E)$	def
	$ $	
	$(\text{val } x E)$	val
$E ::=$	$n$	integer
	$ $	
	$x$	name
	$ $	
	$\text{nil}$	list nil
	$ $	
	$(\text{cons } E E)$	pair constructor
	$ $	
	$(\text{fst } E)$	the first component of a product type value
	$ $	
	$(\text{snd } E)$	the second component of a product type value
	$ $	
	$(\text{inl } E)$	the left tagged value of sum type
	$ $	
	$(\text{inr } E)$	the right tagged value of sum type
	$ $	
	$(\text{nil? } E)$	is nil
	$ $	
	$(\text{int? } E)$	is int
	$ $	
	$(\text{prod? } E)$	is product type
	$ $	
	$(\text{sum? } E)$	is sum type
	$ $	
	$(\text{let } (B^*) E)$	name binding of def/val
	$ $	
	$(\text{app } E E^*)$	function call
	$ $	
	$(\text{match } E ((x) E) ((x) E))$	pattern matching for inl and inr values
	$ $	
	$(+ E E)$	integer addition
	$ $	
	$(- E E)$	integer subtraction
	$ $	
	$(* E E)$	integer multiplication
	$ $	
	$(/ E E)$	integer division
	$ $	
	$(\% E E)$	integer remainder
	$ $	
	$(= E E)$	integer equality
	$ $	
	$(< E E)$	integer less than
	$ $	
	$(> E E)$	integer greater than

- For ill-typed inputs, you can return arbitrary values, or raise exceptions.
- $X^*$  denotes that  $X$  can appear 0 or more times.
- **let** clauses create a new scope like a ‘block’ in Scala. Name bindings **def** and **val** work the similar way as in Scala.
  - **(def f (A\*) E)** assigns name **f** to expression **E** with arguments  $A^*$ .  
Examples include **(def f (a (by-name b)) (+ a b))** and **(def g () 3)**.
  - **(val x E)** assigns name  $x$  to the value obtained by evaluating  $E$ .
  - We do not allow the same name to be defined twice in the frame.
  - You do not have to consider forward reference in **val**. For example, **(val x (cons 1 x))**.
  - Hint: Implement environment with mutable data structure for **lazyness**.
- **Environment** is collection of **Frames**. **Frame** is created when a new scope is created.
- **(inl v)** and **(inr v)** are sum type values.
- **nil** and **(cons v<sub>1</sub> v<sub>2</sub>)** are product type values.
- **(match E<sub>1</sub> ((x<sub>1</sub>) E<sub>2</sub>) ((x<sub>2</sub>) E<sub>3</sub>))** first evaluates  $E_1$  into value  $v$ . If  $v$  is **(inl v<sub>1</sub>)**, it evaluates  $E_2$  with binding  $x_1 := v_1$  to get the final value. If  $v$  is **(inr v<sub>2</sub>)**, it evaluates  $E_3$  with binding  $x_2 := v_2$  to get the final value.
- **true** and **false** are encoded as **inr 0** and **inl 0**, respectively.
- **(nil? E)** first evaluates  $E$  into value  $v$ . If  $v$  is **nil**, it returns **true**.
- **(int? E)** first evaluates  $E$  into value  $v$ . If  $v$  is **integer**, it returns **true**. Otherwise, it returns **false**.
- **(prod? E)** first evaluates  $E$  into value  $v$ . If  $v$  is product type value, it returns **true**. Otherwise, it returns **false**.
- **(sum? E)** first evaluates  $E$  into value  $v$ . If  $v$  is sum type value, it returns **true**. Otherwise, it returns **false**.
- For additional information, post questions on the GitHub course webpage.
- examples in `src/test/scala/TestMain.scala`.

**Problem 2 (15 Points)** Optimize `interp` to handle tail recursive input programs, such as the example code shown below.

```
(let ((def f (x sum) (match (x > 0) ((_) sum) ((_) (app f (- x 1)
(+ x sum)))))) (app f 10 0))
```

**Problem 3 (15 Points)** Add lazy evaluation to `interp` by implementing `by-name` and `lazy-val` following.

$$\begin{array}{ll} A ::= & \dots \\ & | \text{ (by-name } x \text{) } \quad \text{call by name} \\ B ::= & \dots \\ & | \text{ (lazy-val } x \text{ } E \text{) } \quad \text{lazy val} \end{array}$$

- Name bindings `lazy-val` work the similar way as in Scala.
  - `(lazy-val x E)` assigns name  $x$  to the value obtained by evaluating  $E$  lazily.
  - Hint: Implement environment with mutable data structure for `lazyness`.

**Problem 4 (20 Points)** Implement the function to find the  $n$ th prime number in the language defined above.  
Hint: See `nthPrime` function in the lecture note.