CSE130 Discussion Section

Week 7 - Interpreters

2021/11/12

Interpreters

What is an Interpreter?

An interpreter is a program that executes other programs (it can interpret / understand source code) without the need of compiling them.

Usually, it consists of an ***evaluation loop*** that recursively resolves the arguments to an operator from expressions to values.

```
parse :: String → Expr
eval  :: Env → Expr → Value
```

What is an Interpreter?

An interpreter is a program that executes other programs (it can interpret / understand source code) without the need of compiling them.

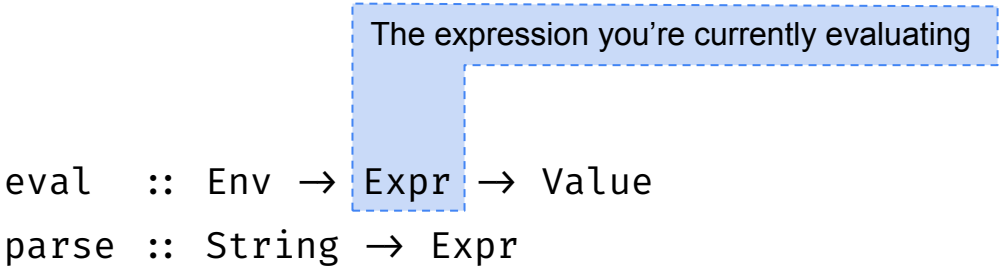
Usually, it consists of an ***evaluation loop*** that recursively resolves the arguments to an operator from expressions to values.

```
eval  :: Env → Expr → Value
parse :: String → Expr
```

What is an Interpreter?

An interpreter is a program that executes other programs (it can interpret / understand source code) without the need of compiling them.

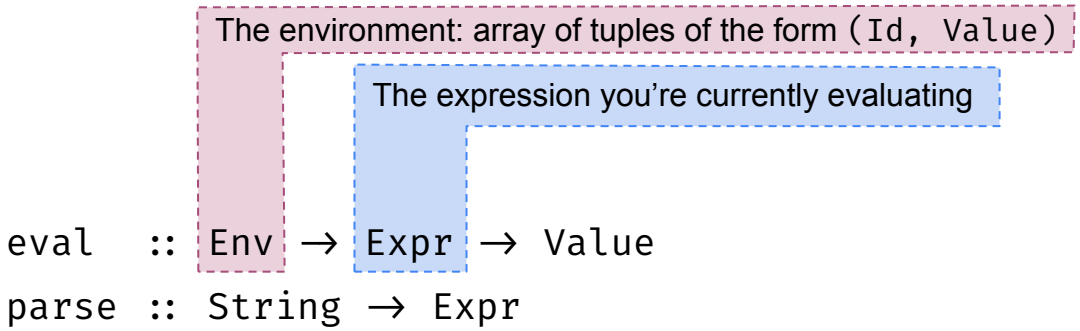
Usually, it consists of an ***evaluation loop*** that recursively resolves the arguments to an operator from expressions to values.



What is an Interpreter?

An interpreter is a program that executes other programs (it can interpret / understand source code) without the need of compiling them.

Usually, it consists of an ***evaluation loop*** that recursively resolves the arguments to an operator from expressions to values.



How to implement an interpreter?

Pattern match `Expr` with the data constructors and handle each case

Sometimes add a new variable to `env`

Also check that types are correct: cannot do 4 + “Burger”, for example

Environments

Let’s run some code in our heads!

```
let a = 1 in

  let b = 2 in

    let a = a + 1 in

      a + b
```

Let’s run some code in our heads!

```
let a = 1 in
  let b = 2 in
    let a = a + 1 in
      a + b
```

What’s the value of the final expression?

Let’s run some code in our heads!

```
let a = 1 in
  let b = 2 in
    let a = a + 1 in
      a + b
```

a and a are different!

Let's run some code in our heads!

```
let a = 1 in
  let b = 2 in
    let a = a + 1 in
      a + b
```

Environment

```
[
    ("a", VInt 1)
]
```

Let’s run some code in our heads!

```
let a = 1 in
  let b = 2 in
    let a = a + 1 in
      a + b
```

Environment

```
[
    (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

Let’s run some code in our heads!

```
let a = 1 in
  let b = 2 in
    let a = a + 1 in
      a + b
```

Environment

```
[
    (“a”, VInt 2)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

Let's run some code in our heads!

```
let a = 1 in
  let b = 2 in
    let a = a + 1 in
      a + b
```

Environment

```
[
    ("a", VInt 2)
  , ("b", VInt 2)
  , ("a", VInt 1)
]
```

`lookupId` finds the ***left-most*** definition of any variable in the environment.
So `a + b` will resolve to `2 + 2 = 4` instead of `1 + 2 = 3`

Closures

Closures

Construction: `VClos Env Id Expr`

This is the representation of a function in your environment

When is a closure created?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“a”, VInt 1)
]
```

When is a closure created?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

When is a closure created?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

App

How do you call a function?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

How do you call a function?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

1. We look up the function name in the environment
2. We take the environment inside the closure

```
VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a
```

```
[(“b”, VInt 2), (“a”, VInt 1)]
```

How do you call a function?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

1. We look up the function name in the environment
2. We take the environment inside the closure
3. Then you bind the parameter to the passed value

```
VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a
[(“b”, VInt 2), (“a”, VInt 1)]
(“x”, VInt 2)
```

How do you call a function?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

1. We look up the function name in the environment
2. We take the environment inside the closure
3. Then you bind the parameter to the passed value
4. Then you pass the new bind to the environment

```
VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a
[(“b”, VInt 2), (“a”, VInt 1)]
(“x”, VInt 2)
[(“x”, VInt 2), (“b”, VInt 2), (“a”, VInt 1)]
```

How do you call a function?

```
let a = 1 in
  let b = 2 in
    let f = \x -> x + a
      in f b
```

Environment

```
[
    (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“b”, VInt 2)
  , (“a”, VInt 1)
]
```

1. We look up the function name in the environment — `VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a`
2. We take the environment inside the closure — `[(“b”, VInt 2), (“a”, VInt 1)]`
3. Then you bind the parameter to the passed value — `(“x”, VInt 2)`
4. Then you pass the new bind to the environment — `[(“x”, VInt 2), (“b”, VInt 2), (“a”, VInt 1)]`
5. And you evaluate the body in the closure with the new environment — `eval [...] (x + a)`

How do you call a function?

```
let a = 1 in

    let f = \x -> x + a in

      let a = 3

        in f a
```

Environment

```
[
  , (“a”, VInt 3)
  , (“f”, VClos [(“b”, VInt 2), (“a”, VInt 1)] “x” x+a)
  , (“a”, VInt 1)
]
```

1. We look up the function name in the environment — `VClos [(“a”, VInt 1)] “x” x+a`
2. We take the environment inside the closure — `[(“a”, VInt 1)]`
3. Then you bind the parameter to the passed value — `(“x”, VInt 2)`
4. Then you pass the new bind to the environment — `[(“x”, VInt 2), (“a”, VInt 1)]`
5. And you evaluate the body in the closure with the new environment — `eval [...] (x + a)`

What about recursive functions?

Now what happens when you have a recursive function?

What about recursive functions?

Now what happens when you have a recursive function?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

What about recursive functions?

Now what happens when you have a recursive function?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment — `VClos [] “x” ITE`
2. We take the environment inside the closure — `[]`
3. Then you bind the parameter to the passed value — `(“x”, VInt 5)`
4. Then you pass the new bind to the environment — `[(“x”, VInt 5)]`
5. And you evaluate the body in the closure with the new environment — `eval [(“x”, VInt 5)] ITE`

What about recursive functions?

Now what happens when you have a recursive function?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

```
eval [(“x”, VInt 5)] ITE  ➡  eval [(“x”, VInt 5)] (sum (x-1))
```

1. We look up the function name in the environment
2. We take the environment inside the closure
3. Then you bind the parameter to the passed value
4. Then you pass the new bind to the environment
5. And you evaluate the body in the closure with the new environment

What about recursive functions?

Now what happens when you have a recursive function?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

```
eval [(“x”, VInt 5)] ITE  ➡  eval [(“x”, VInt 5)] (sum (x-1))
```

Variable name “sum” is not available in the environment!

1. We look up the function name in the environment ✖
2. We take the environment inside the closure
3. Then you bind the parameter to the passed value
4. Then you pass the new bind to the environment
5. And you evaluate the body in the closure with the new environment

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE
```

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE

VClos [(“sum”, VClos [] “x” ITE)] “x” ITE
```

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE

VClos [(“sum”, VClos [] “x” ITE)] “x” ITE

[(“sum”, VClos [] “x” ITE)]
```

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE

VClos [(“sum”, VClos [] “x” ITE)] “x” ITE

[(“sum”, VClos [] “x” ITE)]

(“x”, VInt 5)
```

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE

VClos [(“sum”, VClos [] “x” ITE)] “x” ITE

[(“sum”, VClos [] “x” ITE)]

(“x”, VInt 5)

[(“x”, VInt 5), (“sum”, VClos [] “x” ITE)]
```

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE
VClos [(“sum”, VClos [] “x” ITE)] “x” ITE
[(“sum”, VClos [] “x” ITE)]
(“x”, VInt 5)
[(“x”, VInt 5), (“sum”, VClos [] “x” ITE)]
eval [(“x”, VInt 5)] ITE
```

What about recursive functions?

```
let sum = \x ->
  if x == 0
    then 0
    else x + sum (x - 1)
  in sum 5
```

Environment

```
[
    (“sum”, VClos [] “x” ITE)
]
```

1. We look up the function name in the environment
2. **Add the function value itself into the closure environment**
3. We take the environment inside the closure
4. Then you bind the parameter to the passed value
5. Then you pass the new bind to the environment
6. And you evaluate the body in the closure with the new environment

```
VClos [] “x” ITE
VClos [(“sum”, VClos [] “x” ITE)] “x” ITE
[(“sum”, VClos [] “x” ITE)]
(“x”, VInt 5)
[(“x”, VInt 5), (“sum”, VClos [] “x” ITE)]
eval [(“x”, VInt 5)] ITE
```

➡ `eval [(“x”, VInt 5), (“sum”, VClos [] “x” ITE)] (sum (x-1))`

Lists, Head and Tail

What about native functions?

Use `VPrim` and add them into `prelude`

What about native functions?

Use `VPrim` and add them into `prelude`

Constructor: `VPrim (Value -> Value)`

What about native functions?

Use `VPrim` and add them into `prelude`

Constructor: `VPrim (Value -> Value)`

Environment

```
[
    (“head”, VPrim primHead)
  , (“tail”, VPrim primTail)
]
```

1. We look up the function name in the environment
2. Check whether its a **`VClos`** or **`VPrim`**
3. If it is a **`VClos`**
    a. **Add the function value itself into the closure environment**
    b. We take the environment inside the closure
    c. Then you bind the parameter to the passed value
    d. Then you pass the new bind to the environment
    e. And you evaluate the body in the closure with the new environment
4. If it is a **`VPrim`**
    a. Apply the function inside the VPrim to the argument value

What about native functions?

Use `VPrim` and add them into `prelude`

Constructor: `VPrim (Value -> Value)`

1. We look up the function name in the environment
2. Check whether its a **`VClos`** or **`VPrim`**
3. If it is a **`VClos`**
   a. **Add the function value itself into the closure environment**
   b. We take the environment inside the closure
   c. Then you bind the parameter to the passed value
   d. Then you pass the new bind to the environment
   e. And you evaluate the body in the closure with the new environment
4. If it is a **`VPrim`**
   a. Apply the function inside the VPrim to the argument value

Environment

```
[
    (“head”, VPrim primHead)
  , (“tail”, VPrim primTail)
]
```

```
eval [] (head [1])
```

➡ `VPrim primHead`

➡ `primHead (VCons (VInt 1) VNil)`

➡ `VInt 1`

What about native functions?

Use `VPrim` and add them into `prelude`

Constructor: `VPrim (Value -> Value)`

1. We look up the function name in the environment
2. Check whether its a **`VClos`** or **`VPrim`**
3. If it is a **`VClos`**
   a. **Add the function value itself into the closure environment**
   b. We take the environment inside the closure
   c. Then you bind the parameter to the passed value
   d. Then you pass the new bind to the environment
   e. And you evaluate the body in the closure with the new environment
4. If it is a **`VPrim`**
   a. Apply the function inside the VPrim to the argument value

Environment

```
[
    (“head”, VPrim primHead)
  , (“tail”, VPrim primTail)
]
```

- Implement **`primHead`** and **`primTail`**
- Add **`head`** and **`tail`** as **`VPrim`** to **`prelude`**
- Support **`VPrim`** in **`eval (EApp e1 e2)`**

Summary

HW4 Tips

Problem #1a, 1b: evaluate variables and binary ops

- Use **`BinOp`**'s middle argument to find which binary operator (**`Plus`**, **`Minus`**, **`Cons`**, etc) is used
- Check for that values have the right type
  - Else, **`throw (Error "type error")`**

Problem #1c, 1d, 1e: Let and App

- For **`ELet`**, you'll be updating the environment
- For **`EApp`**, you'll be updating the environment with function parameter
- Remember to add the function value into the closure environment to support recursive functions

Problem #1f: List and native functions

- Implement primitive functions
- Add native functions as **`VPrim`** into prelude
- Modify **`eval EApp`** to support both **`VClos`** and **`VPrim`**