CSE130 Discussion Section
Week 7 - Interpreters

11/12/2021

Interpreter

```
interpret   :: String → Value¹
```

(1) Or throws an exception

Interpreter

```
interpret    :: String → Value

parse        :: String → Expr

eval         :: Env → Expr → Value
```

```
eval :: Env → Expr → Value
```

Pattern match on expressions

Check Types are correct (lazily)

Lookup variables from the environment

(Sometimes) **Add** values to the environment

```
eval :: Env → Expr → Value
```

Lazy Type Checking

```
1 + "Burger"   →   throw (Error "type error")
```

```
eval :: Env → Expr → Value
```

Lazy Type Checking

Use a more meaningful message

```
1 + "Burger"   →   throw (Error "type error")
```

```
eval :: Env → Expr → Value
```
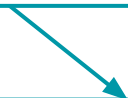
Environment:

```
type Env = [(Id, Value)]

type Id  = String
```

```
eval :: Env → Expr → Value
```

Environment:

```
type Env = [(Id, Value)]

type Id  = String

lookup :: Id → Env → Value
```
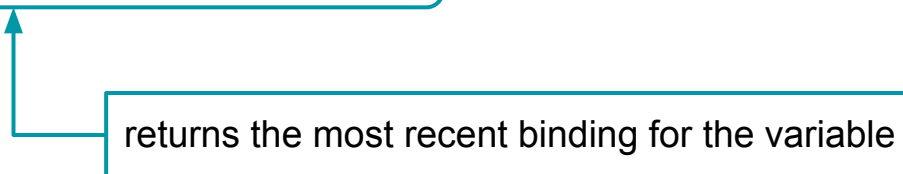
```
eval :: Env → Expr → Value
```

Environment:

```
type Env = [(Id, Value)]

type Id  = String

lookup :: Id → Env → Value
```

returns the most recent binding for the variable

eval :: Env → Expr → Value

What is the result of evaluating this program?

```
let a = 1 in

  let b = 2 in

    let a = a + 1 in

      a + b
```

a) 3
b) 4
c) 5
d) Error

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
let a = 1 in

  let b = 2 in

    let a = a + 1 in

      a + b
```

Environment

```
[
   (“a”, VInt 1)
]
```

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
let a = 1 in

  let b = 2 in

    let a = a + 1 in

      a + b
```

Environment

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

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
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)
]
```

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
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)
]
```

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
let a = 1 in

  let b = 2 in

    let a = a + 1 in

      a + b

      2 + 2 ⇒ 4
```

Environment

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

```
eval :: Env → Expr → Value
```

Closures

- Dynamic vs. Lexical Scoping

- Recursive Functions

eval :: Env → Expr → Value

What is the result of evaluating this program?

```
let a = 1 in

  let f = \x → x + a in

    let a = 3

      in f a
```

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
➡ let a = 1 in

   let f = \x → x + a in

     let a = 3

       in f a
```

Environment

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

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
let a = 1 in

  let f = \x → x + a in

    let a = 3

      in f a
```

```
Environment

[
  ("f", VClos [ ... ])
, ("a", VInt 1)
]
```

eval :: Env → Expr → Value

What is the result of evaluating this program?

```
let a = 1 in

  let f = \x → x + a in

    let a = 3

      in f a
```

Environment

```
[
  ("a", VInt 3)
, ("f", VClos [ ... ])
, ("a", VInt 1)
]
```

eval :: Env → Expr → Value

What is the result of evaluating this program?

```
let a = 1 in

  let f = \x → x + a in

    let a = 3

      in f a
```

Environment

```
[
  ("a", VInt 3)
, ("f", VClos [ ... ])
, ("a", VInt 1)
]
```

```
eval :: Env → Expr → Value
```

What is the result of evaluating this program?

```
let a = 1 in

  let f = \x → x + a in

    let a = 3

      in f a
```

```
(\x → x + a) a
⇒ a + a
⇒ 3 + 1
⇒ 4
```

Environment

```
[
  ("a", VInt 3)
, ("f", VClos [ ... ])
, (“a”, VInt 1)
]
```

```
eval :: Env → Expr → Value
```

Closures

```
data Value
   = …
   | VClos Env Id Expr
   | …
```

The environment when the closure is define

eval :: Env → Expr → Value

Recursive Closures

- Need to know function **Name** and **Definition**.

- Where in `eval` do we have this information?

    - `ELet`
    - Anywhere else?

eval :: Env → Expr → Value

VPrim

- "Built-in" or "Primitive" functions, provided in `prelude`

- VPrim (Value → Value)

- We require two:

    - `head :=` Return the first element of a list
    - `tail :=` Return the last element of a list

```
eval :: Env → Expr → Value
```

That's all for `eval`!

Questions?

Lexing & Parsing

- Read this *carefully*:

    https://github.com/cse130-sp18/arith

- But, here's a summary:

    - Lexing with `Alex` (and Regular Expressions)

    - Parsing with `Happy`

Lexing & Parsing

- Lexing

  Converting `Strings` (list of `Chars`) to Tokens

$$['f', '\ ', '*', '\ ', '1', '2', '3']$$
$$\Rightarrow [\text{ID p "f", MUL p, NUM p 123}]$$

Lexing & Parsing

- Lexing: Regular Expressions

  a   the *letter* "a"

  [ab]   the letter "a", *or* the letter "b"

  [a-z]   *any* lowercase letter

  R1 R2   a string where *some* first part matches R1 and the *rest* matches R2

  R+   *one or more* Rs

  R*   *zero* or more Rs

Lexing & Parsing

- Lexing: Alex

```
-- Lexer.x

$digit = 0-9

tokens :-
  [\*]          { \p s → MUL p           }
  $digit+       { \p s → NUM p (read s) }
```

Lexing & Parsing

-   Lexing: Alex

    ```
    -- Lexer.x

    $digit = 0-9

    tokens :-
      [\*]          {  \p s → MUL p           }
      $digit+       {  \p s → NUM p (read s)  }
    ```

    ```
    AlexPosn → String → Token
    ```

Lexing & Parsing

- Lexing: Alex

```
-- Lexer.x

$digit = 0-9

tokens :-
  [\*]        { \p s → MUL p           }
  $digit+     { \p s → NUM p (read s) }
```

```
          ['f', ' ', '*', ' ', '1', '2', '3']
```

Lexing & Parsing

- Lexing: Alex

```
-- Lexer.x

$digit = 0-9

tokens :-
  [\*]           { \p s → MUL p              }
  $digit+        { \p s → NUM p (read s) }
```

```
        ['f', ' ', '*', ' ', '1', '2', '3']
```

Lexing & Parsing

- Lexing: Alex

```
-- Lexer.x

$digit = 0-9

tokens :-
  [\*]          { \p s → PLUS p          }
  $digit+       { \p s → NUM  p (read s) }
```

```
['f', ' ', '*', ' ', '1', '2', '3']
```

Lexing & Parsing

IDs must start with a letter, but otherwise can contain any letter, digit or underscore ("_"). Which of the following is a valid Regular Expression for matching IDs?

a) $alpha+ $digit* $alpha* _*

b) $alpha [$alpha $digit _]+

c) $alpha [$digit $alpha _]*

d) None of the above

```
$digit = 0-9
$alpha = [a-zA-Z]
```

Lexing & Parsing

IDs must start with a letter, but otherwise can contain any letter, digit or underscore ("_"). Which of the following is a valid Regular Expression for matching IDs?

a) $alpha+ $digit* $alpha* _*

b) $alpha [$alpha $digit _]+

c) $alpha [$digit $alpha _]*

d) None of the above

```
$digit = 0-9
$alpha = [a-zA-Z]
```

Lexing & Parsing

That's all for lexing!

Questions?

Lexing & Parsing

- Parsing: Happy

  Convert list of `Tokens` to an `Expr`

```
       [ID p "f", MUL p, NUM p 123]
    ⇒ EBin Mul (EVar "f") (EInt 123)
```

Lexing & Parsing

- Parsing: Grammar

  Recursive Definition of a set of trees

  1. Terminals:
     - Leaf nodes of the tree (Tokens!)

  2. Non-terminals:
     - Internal nodes of the tree

  3. Production Rules:
     - Rules for building the tree
     - What you will define in the assignment!

Lexing & Parsing

- Parsing: Grammar

```
Aexpr : BinExp                { $1              }
      | TNUM                  { EInt $1         }
      | ID                    { EVar $1         }

BinExp : Aexpr '*' Aexpr   { EBin Mul $1 $3 }
```
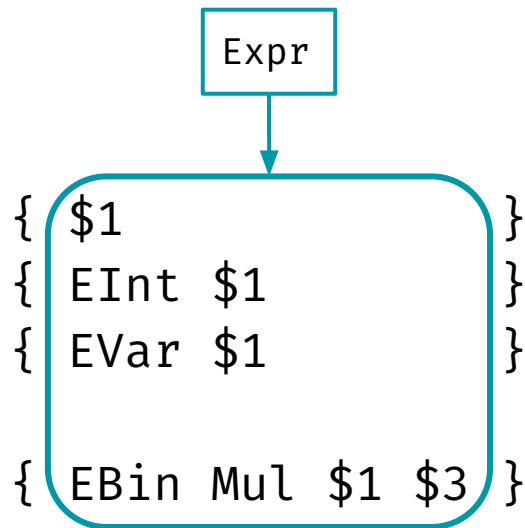
Lexing & Parsing

- Parsing: Grammar

```
Aexpr : BinExp         { $1            }
      | TNUM           { EInt $1       }
      | ID             { EVar $1       }

BinExp : Aexpr '*' Aexpr { EBin Mul $1 $3 }
```

Expr

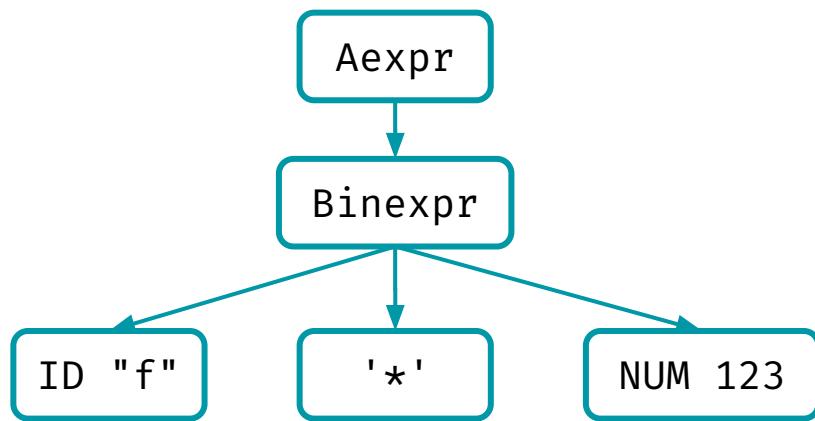# Lexing & Parsing

- Parsing: Grammar

```
[ID p "f", MUL p, NUM p 123]
```



```
Aexpr : BinExp              { $1              }
      | TNUM                { EInt $1         }
      | ID                  { EVar $1         }

BinExp : Aexpr '*' Aexpr    { EBin Mul $1 $3 }
```

Lexing & Parsing

Which of the following *can't* be parsed with our grammar?

a) `1 * 2 * 3`

b) `x * 2 * z`

c) `1 * y * z * 1`

d) They can all be parsed

```
Aexpr : BinExp              { $1              }
      | TNUM                { EInt $1         }
      | ID                  { EVar $1         }

BinExp : Aexpr '*' Aexpr  { EBin Mul $1 $3 }
```

Lexing & Parsing

Which of the following *can't* be parsed with our grammar?

a) `1 * 2 * 3`

b) `x * 2 * z`

c) `1 * y * z * 1`

d) **They can all be parsed**

```
Aexpr : BinExp              { $1                }
      | TNUM                { EInt $1           }
      | ID                  { EVar $1           }


BinExp : Aexpr '*' Aexpr  { EBin Mul $1 $3 }
```

Lexing & Parsing

Parsing: Operator Precedence

- Grammars can be *ambiguous* (multiple ways to parse a string)

- We can disambiguate by:
    a.   Splitting the grammar into more non-terminals
    b.   Using parser "directives" that specify operator precedence

- More in the Arith repo and lecture slides:
    a.   https://github.com/cse130-sp18/arith#precedence-and-associativity
    b.   https://nadia-polikarpova.github.io/cse130-web/lectures/06-parsing.html#precedence-and-associativity

Lexing & Parsing

That's all for parsing!

Questions?

Hints for HW4

- Start early!

- Type-check lazily

- Use meaningful error messages

- Test early, and test often:
  - `eval        :: Env    → Expr → Value`
  - `parseTokens :: String → Either String [Token]`
  - `parse       :: String → Expr`
- Run `make` before `ghci`

- Don't be afraid to split the grammar into more non-terminals for associativity