Nano: Parsing and Eval

CSE 130

2.25.19

Parsing - A simple language

```
AExp ::= Int
       | String
       | Plus AExp AExp
       | Minus AExp AExp
       | Mul AExp AExp
       | Div AExp AExp
```

Goal: String -> AST

“12 + 2” `->  Plus 12 2`

“1 + (2 / “a”)” `-> Plus 1 (Div 2 “a”)`

“(3 / 4) * (2 / 5)” `-> Times (Div 3 4) (Div 2 5)`

Strategy

String -> **LEXER** -> [Token] -> **PARSER** -> AST

Strategy

String -> **LEXER** -> [Token] -> **PARSER** -> AST

Alex: Generates a Lexer in Haskell from .x file

Happy: Generates a parser in Haskell from .y file

Lexer :: String -> [Token]

Convert list of Chars to a high-level representation of same information

[‘5’,’0’,’0’,’ ‘,’+’,’ ‘,’1’,’2’] -> [500, Plus, 12]

[‘1’,’ ‘,’+’,’ ‘,’(‘,’3’,’ ‘,’*’,’ ‘,’2’,’)’] -> [1, Plus, LParen, 3, Times, 2, RParen]

Parser :: [Token] -> AST

[500,Plus,12] -> Plus 500 12

[1,Plus,LParen,3,Times,2,RParen] -> Plus 1 (Times 3 2)

Writing a Lexer

Need to define mappings from sequences of characters to tokens

```
data Token
  = NUM    AlexPosn Int
  | ID     AlexPosn String
  | PLUS   AlexPosn

....
```

Writing a Lexer

```
data Token
  = NUM    AlexPosn Int
  | ID     AlexPosn String
  | PLUS   AlexPosn
```

....

Define rules of the form | <regexp> {haskell-expr}

When <regexp> is matched, we evaluate {haskell-expr} to generate a token

Writing a Lexer

```
data Token
  = NUM    AlexPosn Int
  | ID     AlexPosn String
  | PLUS   AlexPosn
```

....

Define rules of the form | <regexp> {haskell-expr}

When <regexp> is matched, we evaluate {haskell-expr} to generate a token

`Haskell-expr :: AlexPosn -> String -> Token`

More lexing

```
data Token
  = NUM    AlexPosn Int
  | ID     AlexPosn String
  | PLUS   AlexPosn
```

Declare a mapping from patterns to a corresponding Haskell expression that returns a Token:

```
[\+]    { \p _ ->  PLUS p }

$digit+ { \p s -> NUM p (read s) }
```

Writing regexes

https://www.haskell.org/alex/doc/html/regexps.html

Parsing :: [Token] -> AST

Happy uses a **Context-Free Grammar** to define the tree structure

Terminal objects (leaf nodes of tree): TNUM and ID. Other token declarations simply map to values of the Token type. Tokens are re-defined

```
%tokentype { Token }

%token
    TNUM  { NUM _ $$ }
    ID    { ID _ $$  }
    '+'   { PLUS _   }
    '-'   { MINUS _  }
    '*'   { MUL _    } ...
```

Parsing :: [Token] -> AST

Non-terminals describe internal nodes of AST:

```
Aexpr : BinExp                          { $1             }
      | TNUM                            { AConst $1      }
      | ID                              { AVar   $1      }
      | '(' Aexpr ')'                   { $2             }

BinExp : Aexpr '*' Aexpr                { AMul   $1 $3 }
       | Aexpr '+' Aexpr                { APlus  $1 $3 }
       | Aexpr '-' Aexpr                { AMinus $1 $3 }
       | Aexpr '/' Aexpr                { ADiv   $1 $3 }
```

Parsing :: [Token] -> AST

Structure of rules corresponds to recursive structure of type definitions:

```
Aexpr : BinExp
      | TNUM
      | ID
      | '(' Aexpr ')'

BinExp : Aexpr '*' Aexpr
       | Aexpr '+' Aexpr
       | Aexpr '-' Aexpr
       | Aexpr '/' Aexpr
```

```
data Aexpr
  = AConst  Int
  | AVar    String
  | APlus   Aexpr Aexpr
  | AMinus  Aexpr Aexpr
  | AMul    Aexpr Aexpr
  | ADiv    Aexpr Aexp
```

Parsing :: [Token] -> AST

The hardest part of writing parsers is figuring out a recursive definition for the grammar.

A problem

```
evalString [] “2 * 5 + 5” = 20

evalString [] “2 - 1 - 1” = 2
```

A problem

```
evalString [] “2 * 5 + 5” = 20

Should be

(2 * 5) + 5
```

A problem

```
evalString [] “2 * 5 + 5” = 20

Should be

(2 * 5) + 5 = 15

Can be parsed as

(2 * 5) + 5

OR

2 * (5 + 5)
```

A problem

```
evalString [] “2 - 1 - 1” = 2

Should be

(2 - 1) - 1
```

A problem

evalString [] “2 - 1 - 1” = 2

Should be

(2 - 1) - 1

Can be parsed as

(2 - 1) - 1

OR

2 - (1 - 1)

A problem

We want to indicate that * has higher **precedence** than +

We want to indicate that - is **left-associative**

A solution

```
Aexpr : Aexpr '+' Aexpr2
      | Aexpr '-' Aexpr2
      | Aexpr2

Aexpr2 : Aexpr2 '*' Aexpr3
       | Aexpr2 '/' Aexpr3
       | Aexpr3

Aexpr3 : TNUM
       | ID
       | '(' Aexpr ')'
```

Why does this work?

“2 * 5 + 5”

Parser first looks for + or -

_ + 5 -> Plus _ 5

Why does this work?

“2 * 5 + 5”

There is now only ONE unique way to generate this string from our grammar

Start by applying the “+” rule:

_ + 5

Then apply the “*” rule:

(2 * 5) + 5

Why does this work?

“2 - 1 - 1”

There is now only ONE unique way to generate this string from our grammar

Any expression with more than one subtraction operation must have the extra subtractions in the LEFT subtree of the AST:

(2 - 1) - 1 is valid, but 2 - (1 - 1) is not, since anything on the right side of a subtraction must be generated by the Aexpr2 rule.

Another solution

```
%left '+' '-'
%left '*' '/'
```

Tells parser generator that operators are left-associative

Operators declared on bottom have higher precedence

Another solution

```
%left '+' '-'
%left '*' '/'
```

Tells parser generator that operators are left-associative

Operators declared on bottom have higher precedence

Parsing :: [Token] -> AST

We could have defined our parser grammar exactly like the datatype:

```
Aexpr : TNUM
      | ID
      | '(' Aexpr ')'
      | Aexpr '*' Aexpr
      | Aexpr '+' Aexpr
      | Aexpr '-' Aexpr
      | Aexpr '/' Aexpr
```

```
data Aexpr
  = AConst  Int
  | AVar    String
  | APlus   Aexpr Aexpr
  | AMinus  Aexpr Aexpr
  | AMul    Aexpr Aexpr
  | ADiv    Aexpr Aexp
```

Parsing :: [Token] -> AST

It's generally easier to reason about the grammar if split into subtrees (AND you can deal with operator precedence):

```
Aexpr : BinExp
      | TNUM
      | ID
      | '(' Aexpr ')'

BinExp : Aexpr '*' Aexpr
       | Aexpr '+' Aexpr
       | Aexpr '-' Aexpr
       | Aexpr '/' Aexpr
```

```
data Aexpr
  = AConst  Int
  | AVar    String
  | APlus   Aexpr Aexpr
  | AMinus  Aexpr Aexpr
  | AMul    Aexpr Aexpr
  | ADiv    Aexpr Aexp
```

Extending our parser and lexer

What if we want to add boolean expressions to our language?

data AExpr = … | ITE BExpr AExpr AExpr

```
data BExpr = BTrue
           | BFalse
           | Eq AExpr AExpr
```

New tokens and matching regexes:

```
data Token = …
           | TRUE AlexPosn
           | FALSE AlexPosn
           | BEQ AlexPosn
           | IF AlexPosn
           | THEN AlexPosn
         …

“==” { \p _ -> BEQ p }
if   { \p _ -> IF p }
then { \p _ -> THEN p }
...
```

Extend the grammar

Declare more tokens in the .x file

```
...
then { THEN _ }
else { ELSE _ }
‘==’ {BEq _}
...
```

Extend the grammar

```
Aexpr : BinExp                                { $1           }
      | TNUM                                  { AConst $1    }
      | ID                                    { AVar   $1    }
      | '(' Aexpr ')'                         { $2           }
      | if BoolExp then Aexpr else Aexpr { ITE $2 $4 $6 }

BoolExp : true                   { BTrue }
        | false                  { BTrue }
        | Aexpr eq Aexpr         { BEq $1 $3 }
```

Breaking the grammar up makes it easier to extend!

More detail on this example:

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

Alex docs: https://www.haskell.org/alex/doc/html/index.html

Happy docs: https://www.haskell.org/happy/

Eval - Recursive let

```
let x = e1 in e2
```

If e1 is not a function, how do we implement eval?

Eval - Recursive let

```
let x = e1 in e2
```

```
let y = 3 in (1 + y)
```

Eval - Recursive let

```
let x = e1 in e2
```

What happens if e1 IS a function?

Eval - Recursive let

```
let sum = \n -> if n <= 0 then 0 else n + sum (n - 1)

in  sum 5
```

How do we evaluate this?

```
let sum = \n -> if n <= 0 then 0 else n + sum (n - 1)

in  sum 5
```

One approach

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

Add sum (as a closure) to the context, evaluate `sum 5`

```
[“sum”, <[],\n -> if n <= 0 then 0 else n + sum (n - 1)>]
sum 5 = ??
```

One approach

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

Add sum (as a closure) to the context, evaluate `sum 5`

```
[“sum”, <[], \n -> if n <= 0 then 0 else n + sum (n - 1)>]
sum 5 = (\n -> if n <= 0 then 0 else n + sum (n - 1)) 5
```

One approach

```
let sum = \n -> if n <= 0 then 0 else n + sum (n - 1)

in  sum 5
```

Add sum (as a closure) to the context, evaluate `sum 5`

```
[“sum”, <[], \n -> if n <= 0 then 0 else n + sum (n - 1)>]

sum 5 = (\n -> if n <= 0 then 0 else n + sum (n - 1)) 5
```

One approach

Add sum (as a closure) to the context, evaluate `sum 5`

```
[“sum”, <[], \n -> if n <= 0 then 0 else n + sum (n - 1)>]

sum 5 = (\n -> if n <= 0 then 0 else n + sum (n - 1)) 5
```

Will this work?

One approach

Add sum (as a closure) to the context, evaluate `sum 5`

```
[“sum”, <[], \n -> if n <= 0 then 0 else n + sum (n - 1)>]
```

```
sum 5 = (\n -> if n <= 0 then 0 else n + sum (n - 1)) 5
```

Will this work?

One approach

Add sum (as a closure) to the context, evaluate `sum 5`

```
[“sum”, <[], \n -> if n <= 0 then 0 else n + sum (n - 1)>]
```

```
if 5 <= 0 then 0 else n + sum (5 - 1)
```

Will this work?

What happens when we make the recursive call?