LECTURE 03

Theory and Design of PL (CS 538)

January 31, 2020

MORE ON HW1

UPDATES

- Make sure to compile with `-Wall` before submitting
    - If there are warnings in starter code, please fix
- Make sure to run `hlint` before submitting
    - **Don't** need to do all changes; use your judgment

SMALL CONTEST

- We will run all solutions on several new puzzles
- Fastest solutions get a small prize (not for grade)
- Details:
  - Run on instructional machines
  - One solution, and first N solutions

> *Will grade solutions for **given** functions*

MORE HIGHER-ORDER

EXAMPLE: APPLICATION

- Apply a function to an argument, get result:

```
($) :: (a -> b) -> a -> b
fun $ arg = f arg
```

- Why use this? One use: avoiding parentheses

```
plusOne :: Int -> Int

val = plusOne $ plusOne 42
-- SAME AS: val = plusOne (plusOne 42)
-- BUT NOT: val = plusOne plusOne 42
```

EXAMPLE: COMPOSITION

- Chain two functions, get another function:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) sndFun fstFun x = sndFun (fstFun x)
-- NOTE: order matters!
```

- Example: repeat functions:

```
doTwice :: (a -> a) -> a -> a
doTwice fun = fun . fun

plusTwo = doTwice plusOne
```

EXAMPLE: FLIP

- Swap arguments of a two-argument function. Type?

```
flip :: (a -> b -> c) -> b -> a -> c
--- SAME AS: (a -> b -> c) -> (b -> a -> c)
```

- How can we implement this function?

```
flip f y x = f x y
```

EXAMPLE: UNTIL

- Repeat fn from init until condition holds. Type?

```haskell
until :: (a -> Bool) -> (a -> a) -> a -> a
```

- How can we implement this function?

```haskell
until stop f cur
  | stop cur = cur
  | otherwise = until stop f (f cur)
```

EXAMPLE: CURRYING

MULTIPLE ARGUMENTS

- Given *two* integers, produce integer
- First possible type (*uncurried*):

```
myBinaryFn :: (Int, Int) -> Int

foo = myBinaryFn (7, 42)
```

A BETTER TYPE

- Given *one* integer, produce *function* from int to int
- Second possible type (*curried*):

```
myBinaryFn' :: Int -> Int -> Int
-- SAME AS: myBinaryFn' :: Int -> (Int -> Int)
-- BUT NOT: myBinaryFn' :: (Int -> Int) -> Int

foo = myBinaryFn' 7 42
```

PARTIAL APPLICATION

- Don’t need to provide all arguments at once:

```
plus :: Int -> Int -> Int
plus x y = x + y

plusOne :: Int -> Int
plusOne = plus 1           -- SAME AS: plusOne y = 1 + y
```

- Only works for *curried* functions, not uncurried

```
plus' :: (Int, Int) -> Int
plus' (x, y) = x + y

plusOne' = plus' ???
```

CURRY/UNCURRY

- From uncurried to curried:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)
```

- From curried to uncurried:

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (x, y) = f x y
```

WHAT IS A VALID PROGRAM?

A VALID PROGRAM...

- Doesn’t crash when you run it
- Applies functions to arguments of the right types
- Has properly nested parentheses (…), braces {…}
- …

BASIC CRITERIA: SYNTAX

- Can check *statically*, without running program
- If syntax is wrong, program is definitely wrong
- If syntax is right, program could still be wrong

WORDS AND PHRASES

- Different kinds of words
    - Constants (`0`, `true`), operations (`+`, `-`, `*`)
    - Variable names (`x`)
    - Keywords (`if`, `then`, `else`, `let`, `where`)
- Compound words (phrases)
    - Expressions (`2*x+1`)
    - If-statements (`if b then 3 else 4`)

HOW TO SPECIFY SYNTAX?

GRAMMARS

- List of *production rules*: different kinds of phrases
- Terminals written `"..."` or `'...'`
- Pipe `|` means or
- Each rule ended by semicolon
- Example:

```
digit-0-to-4 = "0" | "1" | "2" | "3" | "4" ;
digit-5-to-9 = "5" | "6" | "7" | "8" | "9" ;
digit        = digit-0-to-4 | digit-5-to-9 ;
```

REPEATING, OPTIONAL

- Braces for *repetition*, zero or more times:

```
num        = digit { digit }
```

- Brackets for *option*, zero or one times:

```
signed-num = [ "-" ] num
```

- EBNF grammars, Extended Backus-Naur Form

BASIC EXAMPLES

BOOLEANS

Begin with Boolean constants:

```
bool-cons = "true" | "false"         ; (* constants  *)
```

Then add logical combinations:

```
bool-expr = bool-cons                  (* constants  *)
          | "!" bool-expr              (* negation   *)
          | "(" bool-expr ")"          (* paren term *)
          | bool-expr "==" bool-expr   (* equals     *)
          | bool-expr "&&" bool-expr   (* and        *)
          | bool-expr "||" bool-expr ; (* or         *)
```

NUMBERS

Integers and arithmetic operations

```
num-expr = signed-num              (* constants  *)
         | "-" num-expr            (* negate     *)
         | "(" num-expr ")"        (* paren term *)
         | num-expr "+" num-expr   (* add        *)
         | num-expr "-" num-expr   (* minus      *)
         | num-expr "*" num-expr ; (* multiply   *)
```

EXAMPLE: LAMBDA CALCULUS

WHY A CORE LANGUAGE?

- Simple enough to fully model
  - Remove all unnecessary features
  - Easier to study without extra noise
- Clarify key language similarities/differences

BRIEF HISTORY

- Universal model of computation
- Equivalent to Turing machines in power
- Common ancestor of *all* functional languages

STARTING POINT

Begin with variable names and constants:

```
var  = "x" | "y" | "z" | ... ;

expr = var                    (* variables  *)
     | bool-cons | num-cons   (* base const *)
     | "(" expr ")"         ; (* paren expr *)
```

DEFINING FUNCTIONS

```
expr = var                    (* variables  *)
     | bool-cons | num-cons   (* base const *)
     | "(" expr ")"           (* paren expr *)
     | "λ" var "." expr     ; (* functions  *)
```

Functions have input variable, body expression

CALLING FUNCTIONS

```
expr = var                    (* variables   *)
     | bool-cons | num-cons   (* base const  *)
     | "(" expr ")"           (* paren expr  *)
     | "λ" var "." expr       (* functions   *)
     | expr " " expr        ; (* application *)
```

Call function with argument by separating with space

ADD PRIMITIVES AS NEEDED

Adding in some Boolean operations...

```
expr = ...
     | expr "==" expr
     | expr "&&" expr
     | expr "||" expr
     | "!" expr ;
```

...and some other operations

```
expr = ...
     | expr "+" expr
     | expr "*" expr
     | "-" expr
     | "if" expr "then" expr "else" expr ;
```

EXAMPLE

CONCRETE VERSUS ABSTRACT SYNTAX

TWO KINDS OF SYNTAXES

- Both can be described by grammars
- Concrete: string of characters
  - Source code from a file
  - Data sent over a network
- Abstract: tree with labeled nodes

CONCRETE IS GOOD, BUT...

- Keeps a lot of irrelevant details
  - Parentheses, spaces, ...
- Some important features are hard to see
  - Ambiguity: `1+2*3` is `(1+2)*3`? or `1+(2*3)`?
  - Where is the scope of variables?

ABSTRACT SYNTAX TREES

- Represent program code as a *labeled tree*
- Each node has:
  - a label (an *operation*)
  - some number of child trees (maybe 0)
- Different representation of actual code

> *Code is more than a just list of characters*

EXAMPLE

CONCRETE VS. ABSTRACT?

- Concrete: closer to what programmers *write*
  - Useful when parsing actual programs
- Abstract: closer to what a program *means*
  - Useful when representing code in compilers
  - Useful when performing optimizations
  - Useful when proving things about programs

We will mostly work with abstract syntax