Recursion

- I claimed last lecture that λ-calculus was as expressive as any programming language, e.g. it is Turing-complete
- But for Turing completeness, we need to write recursive functions in λ-calculus

Recall: Named Function

- Write function definition as
  fun f with x in e ≡ let f = λx.e in
- Function call is now just application (f e₁) → (λx.e)e₁
- What about recursion?

Recursion

- Let us try to define a function that computes the factorial of a number
- Recall recursive factorial definition:
  - Factorial of 0 is 1
  - Factorial of n is n× Factorial of (n − 1)
- Let’s try to write this in λ-calculus:
  - fun f with n = (if n = 0 then 1 else n × (f(n − 1))) in ...
  - Does this work?
Recursion

- Next, expand the let binding:
- Recall: let \( z = e_1 \) in \( e_2 \) defined as \( e_2[e_1/x] \)
- let \( f = \lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \ast (f(n-1))) \) in \( (f\ 3) \rightarrow (\lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \ast (f(n-1))))\ 3 \)
- Left with undefined symbol \( f \)
- Conclusion: We cannot encode recursion using named functions

What about Recursion?

- On the face of it, \( \lambda \)-calculus does not seem to allow recursion
- But this would make \( \lambda \)-calculus very boring; not many interesting functions can be computed without recursion
- Amazing fact: It is possible to encode recursion in \( \lambda \)-calculus
- It is just a little bit involved (but very instructive to understand)
- Any ideas?

Encoding Recursion

- Recall again the factorial function we would like to compute:
  - fun \( f \) with \( n \) = (if \( n = 0 \) then 1 else \( n \ast (f(n-1)) \))
- Next, expand the let binding:
- Recall: let \( x = e_1 \) in \( e_2 \) defined as \( e_2[e_1/x] \)
- let \( f = \lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \ast (f(n-1))) \) named
- functions

Encoding Recursion Cont.

- Now, we can use a \( \lambda \)-abstraction to remove \( n \) from the left-hand side: \( (f\ n) = (\text{if } n = 0 \text{ then } 1 \text{ else } n \ast (f(n-1))) \)
- \( f = \lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \ast (f(n-1))) \)
- Consider defining another function \( G \) by moving \( f \) to the right-hand side:
  - \( G = \lambda f. \lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \ast (f(n-1))) \)
- To see that this is correct, show that \( f = G(f) \) at home

Fixed Points

- A fixed point of function \( h \) is value \( v \) such that \( v = h(v) \)
- Intuition: The fixed point of \( h \) is applying \( h \) until \( v = h(v) \), i.e., the base case of the recursion is hit
- But completely unclear how we can compute fixed-point of \( h \)
- An expression that computes a fixed point is called a fixed point operator

The \( Y \)-combinator

- We can define a fixed-point operator in \( \lambda \)-calculus as follows:
  - \( Y = \lambda f. (\lambda x. f(x\ x))(\lambda x. f(x\ x)) \)
- This bizarre expression is called \( Y \)-combinator
- Recall property of fixed-point: \( v = h(v) \) for any function \( h \).
- Lets confirm that \( Y \) has this property:
  - \( Y \ h \rightarrow \lambda f. (\lambda x. f(x\ x))(\lambda x. f(x\ x)) \ h \rightarrow (\lambda x. h(x\ x))(\lambda x. h(x\ x)) \rightarrow (h(x\ x))[(\lambda x. h(x\ x))/x] \rightarrow h(\lambda x. h(x\ x))(\lambda x. h(x\ x)) \rightarrow h(Y h) \)
Using the Y-combinator

- Let’s see how we can use the Y—combinator to compute factorial:
  
  - Recall: \( G = \lambda f . \lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \times (f \ (n - 1))) \)
  
  - Claim: Factorial of \( n \) can be computed as \((YG)\ n\)
  
  - Example:
    
    \[
    \begin{align*}
    (YG)\ 2 & \rightarrow (G(YG))\ 2 \\
    & \rightarrow (Af.\lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \times ((YG)\ (n - 1))))\ 2 \\
    & \rightarrow \lambda n. (\text{if } n = 0 \text{ then } 1 \text{ else } n \times ((YG)\ (n - 1))))\ 2 \\
    & \rightarrow \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 \times ((YG)\ 1) \\
    & \rightarrow 2 \times ((YG)\ 1) \\
    & \rightarrow \ldots
    \end{align*}
    \]

Fixed points Summary

- We can compute recursive functions in \(\lambda\)-calculus using fixed-point operators

- We have seen the most famous fixed-point operator, the Y—combinator

- However, there are other \(\lambda\) expressions that also compute fixed points.

- Remember: Not every recursive function has to terminate, so this means we can write \(\lambda\) terms that will reduce forever

Next: Your course project

Course Project Overview

- You will implement a lexer, parser, interpreter for the L language

- You can find a reference interpreter on the UT Austin machines to run L programs on (see the L Language handout for details)

- As the name suggests, L is very similar to \(\lambda\)-calculus, but still a useful language

- L has a bizarre property that is (almost) unique among programming languages. At the end of the semester, there will be many more interpreters for L than L programs

Language Overview

- In L, every expression evaluates to a value

- The result of running a L program is the value of the program

- Example: let \( x = 3 \) in \( x \) will evaluate to “3”

- In addition to integers, L also supports strings

- Example: let \( x = "cs312" \) in \( x \) will evaluate to “cs312”

Language Overview

- Of course, L has the \(\lambda\)-operator

- Example: \((\lambda x. \ x + 3)\ 2\) will evaluate to “5”

- Note: You must write parenthesis for any applications!

- This means \(\lambda x. \ x + 3\ 2\) is not a valid L program
More L Examples

- let g = lambda a. if a > 0 then 2*a else 3*a in
  let u = 12 in
  (g u)
- Value: "24"
- L also supports currying:
  - let x = lambda a, b. a + b in
    let y = (x 2) in
    (y 3)
- Value: "5"

Functions in L

- For convenience, L also has built-in function definitions:
  - fun compute_grade with percent =
    if percent > 90 then "A" else
    if percent > 80 then "B" else "F"
    in
    (compute_grade 73)
  - Result: "F"

Recursion in L

- Unlike \(\lambda\)-calculus, L allows you to write "naturally" recursive functions
  - fun factorial with n =
    if n <= 0 then 1 else n * (factorial (n-1)) in
  - Can also write "naturally recursive" anonymous functions:
    - let fact = lambda n. if n = 0 then 1 else n * (fact (n-1)) in
    (fact 6)
  - We will learn later how L allows natural recursion

Input/Output in L

- L has special operators for input/output:
  - let _ = print "Please enter an integer: " in
    let i = readInt in
    let _ = print "Please enter a string: " in
    let s = readString in
    let _ = print "Integer read: " in
    let _ = print i in
    let _ = print "String read: " in
    let _ = print s in
    0

Lists in L

- L also supports lists
- Lists are represented as pairs with a head and tail element
- This allows very generic data structures, no just linear lists
- L has the following list operators:
  - isNil: 1 if list is empty, 0 otherwise
  - e1@e2: Returns a list with e1 as head and e2 as tail
  - !e1: Returns head of e1 if e1 is list, e1 otherwise
  - #e1: Returns tail of e1 if e1 is list, Nil otherwise

List Examples

- let x = 1@2@3@4 in x
- Value: "[1, [2, [3, 4]]]"
- let x = 1@2@3@4 in !x
- Value: "1"
- let x = 1@2@3@4 in #x
- Value: "[2, [3, 4]]"
More List Examples

- How about computing the length of a list?
- fun length with l =
  if isNil l then 0 else 1 + (length (#l))
  in
  (length "A"@"B"@"C")
- Value: "3"

Run-time errors

- There are many run-time errors possible in L programs:
- Example: let x = "hello" in x+3
- Result: "Run-time error: Binop + can only be applied to expressions of same type"
- The L reference manual lists all possible errors

Course Project Details

- Your first four programming assignments will use L and built an L interpreter
  - Assignment 0: Develop and L program
  - Assignment 1: Lexer
  - Assignment 2: Parser
  - Assignment 3: Interpreter
  - Assignment 4: Type Inference
- You will complete these assignments in L and C++
- I posted a quick C++ introduction on the website
- But we will use only tiny subset of C++, easy to pick up