# Compiling Functional Programming Languages

**Kesha Hietala (Advisor: Gopalan Nadathur), University of Minnesota Twin Cities**

## Project Objective

To implement two classical approaches to compiling functional programming languages and to compare their behavior with regard to efficiency

## Functional Programming Languages: What and Why

- A formalism that provides a high-level of abstraction, which allows for:
  - natural support for complex, structured data
  - the ability to treat functions (programs) themselves as data
  - a focus on problem solving rather than machine structure
- A powerful framework for developing complex programs correctly
  - abstraction mechanisms match the conceptual requirements of complex, data-oriented programming
  - mathematical structure facilitates reasoning about programs
  - low level details can be relegated to compilation
- A programming vehicle that is practical and growing in use
  - OCaml, Haskell, F#, and Swift are used in industry and gaining in popularity
  - offer competitive efficiency for all but extremely machine-oriented computations

## Approaches to Solving Compilation Problems

- Here we consider two approaches:
  - the Categorical Abstract Machine (CAM), which is the basis for the popular language OCaml and relies on the use of *categorical combinators*
  - compiling with continuations, which has been used in compilers for the languages Scheme and Standard ML and relies on *continuations* to make control flow explicit
- Both approaches use *closures* to associate code with an environment of variable bindings, allowing functions to be treated as first-class objects
- The most significant difference between the two approaches is how they handle control
  - consider code generated for the expression:

```
let j =
    let y = 3
    in let f x = x + y
        in (f 2) + y
```

### CAM Approach

- Evaluate expressions in the context of an environment
- Compile `j` into something of the following form:

```
<bind y to 3>
<bind f to a closure>
<evaluate (f 2) to v1>
<evaluate y to v2>
<apply + to v1 and v2>
```

- Requires a machine structure that correctly maintains the environment

### Continuations-based Approach

- Isolate where computations should take place next and extract this part into a new let expression
- The binding for `j` becomes:

```
let j =
    let y = 3
    in let f x = x + y
        in let w = (f 2)
            in w + y
```

- Translate the resulting expression into code with no special treatment for control

## Problems with Compiling Functional Languages

- *Compilation* is an essential component to closing the gap between a high-level language and what a machine can understand
- Compiling functional languages poses special difficulties because they treat functions as *first-class objects*

  ➢ Functions can be returned as values

```
fun f x =
    let g y = x + y
    in g
```

```
val h = (f 2)
val i = (f 3)
```

  **Problem:** `h` and `i` must be represented by the same code, but require different values for `x`

  ➢ Functions can be provided as arguments

```
fun j =
    let f x y = x + y
    in let g z = z 3 in g (f 2)
```

  **Problem:** How do we structure the evaluation of `g` and `(f 2)` in computing `g (f 2)`?

## Project Achievements

- Developed an understanding of the two different models of compilation
- Implemented both approaches for an expressive fragment of call-by-value functional languages
- Qualitatively characterized differences between the two models relevant to performance
  - in the CAM model the environment must be explicitly managed while in the continuations approach it grows linearly
  - control is built into the instruction sequence in the CAM model whereas explicit transfers are needed in the continuations approach

  e.g. consider the evaluation of the expression: `let x = 4 in ((let y = 2 in y) + x) + 3`

  ➢ CAM Approach

  - start with empty environment $e_0$
  - add `<x,4>` to $e_0$ to obtain $e_1$
  - add `<y,2>` to $e_1$ to obtain $e_2$
  - evaluate `y` to $v_1$ in $e_2$
  - restore $e_1$
  - evaluate `x` to $v_2$ in $e_1$
  - add $v_1$ and $v_2$

  ➢ Continuations-based Approach

  - start with an empty environment
  - add `<y,2>` to the environment
  - goto $c_1$
  - $c_1$ : add `<x,4>` to the environment
    goto $c_2$
  - $c_2$ : bind `z` to result of `x+y`
    goto $c_3$
  - $c_3$ : add `z` and `3` and return

- Current work is attempting to quantify the impact of these differences by running both implementations on large real-world programs