First-class vs. Second-class

- Values are first-class if they can be used in all the usual ways
  - assigned to local variables
  - passed as arguments to functions/methods
  - returned from functions
  - created at run-time

- *Iota*: modules, functions are denoted by expressions but are only usable in limited ways (uses, function call)

First-class functions

- Many languages allow functions to be used in a more first-class manner than in *Iota* or Java: C, C++, ML, Modula-3, Pascal, Scheme,...
  - Passed as arguments to functions/methods
  - Nested within containing functions (exc. C, C++)
  - Used as return values (exc. Modula-3, Pascal)

Function Types

- *Iota-Fp*: *Iota* with function values that can be passed as arguments (still not fully first-class)
- Need to declare type of argument; will use program notation \( \text{function}(T_1, T_2): T_3 \) to denote the function type \( T_1 \times T_2 \rightarrow T_3 \).
- Example: sorting with a user-specified ordering:
  ```
  sort(a: array[int],
     order: function(int, int):bool) {
    ... if (order(a[i], a[j])) { ... } ...
  }
  ```
Passing a Function Value

\[
\text{leq}(x: \text{int}, y: \text{int}): \text{bool} = x \leq y \\
\text{geq}(x: \text{int}, y: \text{int}): \text{bool} = x \geq y \\
\text{sort}(a: \text{array}[\text{int}], \text{order}: \text{function}(\text{int}, \text{int}): \text{bool}) \ldots
\]

- Allows abstraction over choice of functions

Objects subsume functions!

\[
\text{interface comparer} \\
\quad \text{compare}(x: \text{int}, y: \text{int}): \text{bool} \\
\text{sort}(a: \text{array}[\text{int}], \text{cmp}: \text{comparer}) [ \\
\quad \ldots \text{if} (\text{cmp}.\text{compare}(a[i], a[j])) \{ \ldots \} \ldots \\
\text{class leq} [ \\
\quad \text{compare}(x: \text{int}, y: \text{int}) = x \leq y; \\
\]  
\]

Type-checking functions

- Same rules as in Iota static semantics, but function invoked in function call may be a general expression

\[
f: T_1 \times \ldots \times T_n \rightarrow T_R \in A \\
\frac{A \vdash e_i: T_i}{A \vdash f(e_1, \ldots, e_n): T_R}
\]

- Subtyping on function types: usual contravariant/covariant conditions

Representing functions

- For Iota-F\(_1\), a function may be represented as a pointer to its code (cheaper than an object)

\[
\text{Old translation:} \\
E[e_1, \ldots, e_n] = \text{CALL}((\text{Name}(f), E[e_1], \ldots, E[e_n]))
\]

\[
\text{New:} \\
E[e_1, \ldots, e_n] = \text{CALL}(E[e_1], E[e_2], \ldots, E[e_n])
\]

\[
E[\text{id}] = \text{Name}(\text{id}) \\
\quad \text{(if id is a global fcn)}
\]

Nested Functions

- In functional languages (Scheme, ML) and Pascal, Modula-3, Iota-F\(_1\)
- Nested function can access variables of the containing lexical scope

\[
\text{plot_graph}(f: \text{function}(x: \text{float}): \text{float}) = ( \ldots y = f(x) \ldots )
\]

\[
\text{plot_quadratic}(a, b, c: \text{float}) = ( \\
q(x: \text{float}): \text{float} = a \times x + b \times x + c; \\
\text{plot_graph}(q) )
\]

- Nested functions may access, update free variables from containing scope! Must change rep.

Iteration in Iota-F\(_1\)

- Also useful for implementing iterators and other user-defined control flow constructs

\[
\text{interface set} \{ \text{members}(f: \text{function}(o: \text{object})) \}
\]

\[
\text{countAnimals}(s: \text{set}) = \{ \\
\quad \text{count}: \text{int} = 0; \\
\quad \text{loop_body}(o: \text{object}) = \{ \\
\quad \quad \text{if} (\text{cast}(o, \text{Animal})) \text{count} ++; \\
\quad \} \\
\quad s.\text{members}(\text{loop_body}); \\
\quad \text{return} \text{count}; \\
\}
\]
A subtle program

```plaintext
int f(n: int, 
g1: function(): int, 
g2: function(): int) = ( 
    int x = n+10; 
g1(): int = x; 
    if (n == 0) f(1, g, dummy) 
    else if (n==1) f(2, g1, g) 
    else g1() + g2() + g() 
) 
f(0,dummy,dummy) = ?
```

Lexical scope

- Assignment `g(): int = x` creates a new `function value` = `(let (g (lambda () x))) ...`
- Free variables (x) are bound to the variable lexically visible at time of evaluation of function expression

Closures

- Problem: nested function (g) may need to access variables arbitrarily high up on stack
- Before nested functions: function value was pointer to code (1 word)
- With nested functions: function value is a closure of code + environment for free variables (2 words)

Supporting Closures

```
E \[e_0, e_1, ..., e_n]\] = implicit static link argument
ESEQ(MOVE(t1, E \[e_0]\), CALL(MEM(t1), E \[e_1], ..., E \[e_n]\))
S \[id \ldots; T_e\] : T_e = e
```

- Can optimize direct calls
- Function variable takes 2 stack locations
- What about variable accesses?

Static Link Chains

```
f() = (a: int; 
g() = (b: int; 
h() = ( 
    c = a + b; 
) ... 
) ... 
```

...other function stack frames... 

...other function stack frames... 

...other function stack frames...
Variable access code

- Local variable access unchanged
- Free variable access: walk up $n$ static links before indexing to variable

Progress Report

- Passed as arguments to functions/methods
- Nested within containing functions as local variables
- Used as return values
- If no nested functions, functions are just pointers to code; can be used as return values (C)
- Problem: interaction with nested functions

Iota-F$_2$ (first-class functions)

- Augment Iota-F$_1$ to allow the return type of a function to be a function itself.

  ```
  make_counter(): (function(): int) = (
    // returns a new counter function
    int count = 0;
    inc(): int = (count++);
    return inc
  )
  make_counter() + make_counter() = ?
  c = make_counter(); c() + c() + c() = ?
  ```

Dangling static link!

- Heap-allocate the make_counter activation record (at least count) so that it persists after the call

Heap allocation

- Activation record ≠ stack frame
- Even local variable accesses indierced

The GC side-effect

- With heap-allocated activation records, every function call creates an object that must be garbage collected eventually -- increases rate of garbage generation
- Activation records of all lexically enclosing functions are reachable from a closure via stack link chains
- Putting entire activation record in heap means a lot of garbage is not collectable
Summary

- Looked at 3 languages progressively making functions more first-class
- No lexical nesting (F₀, C)
  - Fast but limited
  - Function = pointer to code
- Lexical nesting, no upward function values or storage in data structures (F₁, Pascal, Modula-N):
  - Function value is closure
- Fully first-class: return values (F₂, Scheme, ML):
  - Lots of heap-allocation, more indirection
  - Functions roughly as powerful as objects (sometimes more convenient), but as expensive as objects... without optimization