2.6 Practical Language

Note Title 1/10/2008

read 2.3 - 2.5 - not going to lecture on them

Kernel language idea: define a simple language and describe its semantics (2.3 and 2.4)

Define practical language by giving its translation to kernel language.
(note "kernel" here does not refer to OS kernel!)

- "syntactic sugar" — conveniences
- linguistic abstraction — new grammatical constructions

Sugar:

local <pattern> = <expression> in <stmt> end

local #1 T = [1 2 3] in ... end

expressions are sugar!

Case expr of
  <pattern> then <stmt>
[] <pattern> then <stmt>
[] <pattern> then <stmt> ...
[else <stmt>] end

Case L of
  nil then ...
[] H|T then ...
end

and then orelse short-circuiting boolean expression evaluation

functions

fun {F X1 ... XN}
  ≡
proc {F X1 ... XN ?R}

<expression> end

R = <expression> end

call: {Q {F X1 ... XN} {G Y1 ... YN} ...}
evaluate nested first, and left to right

declare

interactive only

adds new mappings in a single, global environment of the interactive system (only!)

{Browse <expr>}

p. 89 - data flow execution — play w/ it

Exceptions — Section 2.7

try $\langle s \rangle_0$ catch <x> then $\langle s \rangle_1$ finally $\langle s \rangle_2$ end

raise v

Functional programing is a restriction^ and tweak on this declarative model:

1) local X = V in <s> end
   local X = {P v_1 v_2 ...} in <s> end

2) allow use of only functions and not procedures

tweak: in constructy data structures, make nested calls before creating the data structure (so no unbound variables).

This FPL does not require a data flow store

Skim 2.8.2 on unification

Reuew 2.8.3 on static and dynamic strong typing

Read and think about Exercises 2.1 thru 2.12 } Should be reuied

Chapter 3

Compositionality — (1) Components of programs always mean the same thing. Allows independent design, testing, proof

(2) Relatively simple reasoning (order doesn't matter)

(1) • only connection between components is captured in the input-output relationship of each one and how they are connected

• Replace equals by equals

3.6 Higher-order programming

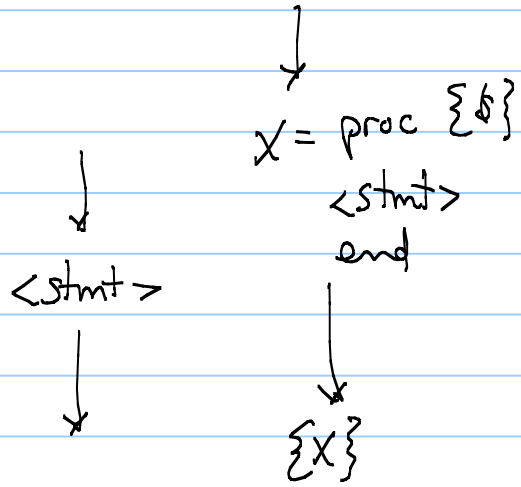
first-order: functions whose args. are not themselves functions

$n+1^{st}$ order: functions whose args. are maximum n^{th} order

HOP: functions can be of any order.

Basic concepts of HOP:

- Procedural abstraction
- Genericity
- Instantiation
- Embedding



Proc. Abst:

Any statement can be packaged in a procedure.

Limited proc. abst. in C, Java

Why limited in C? Implementation decision to put args and local vars on stack

- Genericity
  (ability to make generic)
  any specific entity in the function body (whether operation or value) can be made a parameter of the function

```
fun {SumList L}
   case L of
      nil then
   [] X|L1 then X {SumList L1}
   end
end
```

```
fun {FoldR L
   case L of
      nil then
   [] X|L1 {
   }}
   end
end
```

- Instantiation —
  A function may be returned as the result of another function.
  Static scoping rules continue to apply!

- Embedding —
  A function value (not only the result of a function evaluation) can be embedded in a data structure.

Note that all of these follow from adopting the viewpoint that functions are first-class values. (OR maybe this is what it means for functions to be first-class values).

Modules and components follow clearly from this.

3.6.6. Currying

Named for logician Haskell Curry:

instead of `fun {Max X Y} if X>=Y then X else Y end end`

write

```
fun {Max X}
   fun {$ Y} if X>=Y then X else Y end end
end
```

`{Max 10}` is now a function (instantiation); use it like this

`{{Max 10} 13}`

3.8 File I/O and GUI

module File

declare [File] = {Module.link ['File.ozf']}

operations to read & write files

3.9 Modules and functors

Aside about this binding notation.

Exercises:

2, 4, 5, 14