Imperative vs. functional languages

- Imperative – based on assignment of values to a variable
- Functional – no assignment, therefore no variables
- Standard imperative languages are C, Pascal, Ada
- Standard functional languages are ML, Scheme and the many versions of Lisp

Abstract syntax

- \( P ::= F_1 F_2 \ldots F_n E \)
- \( F ::= I \ (I_1, I_2 \ldots I_m) = E \)
- \( E ::= N \ | \ I \ | \ E_1 O E_2 \ | \ \text{if} \ E_1 \ \text{then} \ E_2 \ \text{else} \ E_3 \ | \ \text{let} \ I = E_1 \ \text{in} \ E_2 \ | \ I \ (E_1, E_2, \ldots E_n) \)
- \( O ::= \text{plus} \ | \ \text{minus} \ | \ \text{mult} \ | \ \text{divide} \ | \ \text{equal} \ | \ \text{greater} \ | \ \text{less} \)

A program is a bunch of function definitions, \( F_i \) and a 'starter' expression which will call possibly other functions.

Each function definition has a name, \( I \), a number of parameters, \( I_i \), and a body \( E \)

Expressions

- A number, \( N \) (taken from the integers)
- An identifier, \( I \)
- A binary operation, arithmetic or relational
- A conditional
- The let form, to introduce an identifier, \( I \), bound to a sub-expression, \( E_1 \) in an expression \( E_2 \)
- The function call, with a number arguments, each of which can be an expression

* The relational expressions will return 1 or 0, just like C, instead of true or false.
Environments

- Instead of a single store, we will have two *environments*, one to hold the values of function parameters, and one to hold function definitions
  
  \( \pi : I \rightarrow \mathbb{Z} \)
  
  \( \varphi : I \rightarrow F \)

- Together they from a pair: \( [\pi, \varphi] \)

Inference Rules for Basic Expressions

- **Numbers:** \( [N, [\pi, \varphi]] \rightarrow n \) (\( n \) is the value of \( n \))
- **Identifiers:** \( [1, [\pi, \varphi]] \rightarrow \pi(1) \)
- **Arithmetic:**
  - \( E_1, [\pi, \varphi] \rightarrow u \quad E_2, [\pi, \varphi] \rightarrow v \) etc.
  - \( E_1 \text{ plus } E_2, [\pi, \varphi] \rightarrow u + v \)
- **Relations:**
  - \( E_1, [\pi, \varphi] \rightarrow u \quad E_2, [\pi, \varphi] \rightarrow v \) \((u > v)\)
  - \( E_1 \text{ greater } E_2, [\pi, \varphi] \rightarrow 1 \)
  - \( E_1, [\pi, \varphi] \rightarrow u \quad E_2, [\pi, \varphi] \rightarrow v \) \((u \leq v)\)

Inference Rules for the Conditional

- Two rules for the two branches:

  \[
  \begin{align*}
  &\left[ E_1, [\pi, \varphi] \right] \rightarrow u \\
  &\quad \left[ E_2, [\pi, \varphi] \right] \rightarrow v \quad (u \neq 0) \text{ and} \\
  &\text{if } E_1 \text{ then } E_2 \text{ else } E_3, [\pi, \varphi] \rightarrow v
  \\
  &\left[ E_1, [\pi, \varphi] \right] \rightarrow u \\
  &\quad \left[ E_2, [\pi, \varphi] \right] \rightarrow w \quad (u = 0) \\
  &\text{if } E_1 \text{ then } E_2 \text{ else } E_3, [\pi, \varphi] \rightarrow w
  \\
  
  \text{Note that only one expression need be} \\
  \text{evaluated and its value returned as the value of} \\
  \text{expression}
  \end{align*}
  \]

Inference Rule for let

- Adds a new identifier and its value to the environment

  \[
  \left[ E_1, [\pi, \varphi] \right] \rightarrow u \\
  \left[ E_2, [I \rightarrow u, \pi, \varphi] \right] \rightarrow v
  \\
  \text{let } I = E_1 \text{ in } E_2, [\pi, \varphi] \rightarrow v
  \]

- The environment for \( E_2 \) is the initial environment, 
  augmented by the value of \( E_1 \) bound to \( I \)
- The value returned is that of \( E_2 \)
- let expressions can be nested any number of times
Inference Rule for Function Call

- The idea is to evaluate all the arguments, then retrieve the function definition from $\phi$, make a new environment by binding each parameter from the function to the corresponding argument value, and evaluate the body of the function:

$$
\begin{align*}
1. \text{Evaluate arguments} &: [E_i[x_i]] \mapsto u_i \quad (\text{for } i = 1, \ldots, r) \\
2. \text{Retrieve function definition} &: \phi \mapsto \phi' \\
3. \text{Make new environment} &: [E \mapsto u_1] \ldots [E \mapsto u_r] \mapsto [\phi'] \\
4. \text{Evaluate body in new environment} &: \phi'[E_i[x_i]] \mapsto v
\end{align*}
$$

- This gives “call by value” semantics where all the arguments are evaluated prior to applying the function.

Running the Program

$$
\begin{align*}
\text{nameof}(E_1) &\rightarrow I_1, \ldots, \text{nameof}(E_n) &\rightarrow I_n, \left[E_i\left[p_i, [I_i \mapsto E_i] \ldots [I_n \mapsto E_n] \cup \{\}ight]\right] &\rightarrow v \\
F_1 \cdots F_n &\rightarrow v
\end{align*}
$$

- E cannot contain any variables – only function calls, operations and constants.
- The initial parameter environment is empty, and the function environment only has definitions for those functions in the program.
- $\text{nameof}$ is a function that extracts the name of the function from the definition.

Examples of valid programs

- $f(x) = x + 1$
  - $\text{let } y = f(3) \text{- } 2 \text{ in } y^5$

- $f(x) = \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x-1)$
  - $f(4)$

- $1 + 2 \cdot 5 \text{- } 3$

- $E$ mentions a variable $y$, but it is bound to a value by let before the enclosed expression is evaluated.

- Recursion is perfectly allowable (why?)

- We could have no definitions, then $E$ must be all constants.

Lazy evaluation of parameters

- Eager evaluation evaluates all arguments before binding to parameters.
- Lazy evaluation binds the expression and the environment and only evaluates the argument if needed.
Rule for lazy function call

- Bind parameters to $[E, \pi]$ pairs

\[
\text{function call: } \left[ l_1 \mapsto [E, \pi], \ldots, l_n \mapsto [E, \pi] \right] \rightarrow v \\
\left( I = \left\{ l_1, \ldots, l_n \right\} = E \right)
\]

Evaluation of lazy parameters

- If the parameter needs evaluation:

\[
\text{identifiers: } \left[ E, \pi', \phi \right] \rightarrow v \\
\left( \text{where } \pi(1) = [E, \pi'] \right)
\]

- The same should be done for let variables