### 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 ::= F1 F2...Fn E

F ::= I (I1,I2...In) = E

E ::= N | I | E1 O E2 | if E1 then E2 else E3 |
     let I = E1 in E2 | I (E1,E2,...En)

O ::= plus | minus | mult | divide | equal | greater | less
```

A program is a bunch of function definitions, F, and a 'starter' expression which will call possibly other functions

Each function definition has a name, I, a number of parameters, 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, E₁ in an expression E₂
- 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
  \( \pi : I \rightarrow \mathbb{Z} \)
  \( \phi : I \rightarrow \mathbb{F} \)
- Together they from a pair: \([\pi, \phi]\)

**Inference Rules for Basic Expressions**
- Numbers: \(N, [\pi, \phi] \rightarrow n\) (n is the value of n)
- Identifiers: \(I, [\pi, \phi] \rightarrow \pi(1)\)
- Arithmetic: \(E₁, [\pi, \phi] \rightarrow u\), \(E₂, [\pi, \phi] \rightarrow v\)
- \(E₁ + E₂, [\pi, \phi] \rightarrow u + v\) etc.
- Relations: \(E₁, [\pi, \phi] \rightarrow u\), \(E₂, [\pi, \phi] \rightarrow v\)
- \(E₁ > E₂, [\pi, \phi] \rightarrow 1\) (u > v) and
- \(E₁ \leq E₂, [\pi, \phi] \rightarrow 0\) (u ≤ v)
Inference Rules for the Conditional

- Two rules for the two branches:

\[
\begin{align*}
E_1, [\pi, \phi] & \rightarrow u, \quad E_2, [\pi, \phi] \rightarrow v \quad (u \neq 0) \\
\text{if } E_1 \text{ then } E_1, [\pi, \phi] \rightarrow v \\
\text{if } E_1 \text{ else } E_2, [\pi, \phi] \rightarrow w \quad (u = 0)
\end{align*}
\]

Note that only one expression need be evaluated and its value returned as the value of expression

Inference Rule for let

- Adds a new identifier and its value to the environment

\[
E_1, [\pi, \phi] \rightarrow u, \quad E_2, [I \rightarrow u, \pi, \phi] \rightarrow v
\]

\[
\text{let } I = E_1 \text{ in } E_2, [\pi, \phi] \rightarrow v
\]

- 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:

\[
\begin{align*}
E_1, [\pi, \phi] & \rightarrow u, \quad E_2, [\pi, \phi] \rightarrow v \\
&\text{let } I = E_1 \text{ in } E_2, [\pi, \phi] \rightarrow v
\end{align*}
\]

\[
\text{let } I = E_1 \text{ in } E_2, [\pi, \phi] \rightarrow v
\]

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

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
`nameof` is a function that extracts the name of the function from the definition

Examples of valid programs

- `f(x)=x+1`
- `let y=f(3)-2 in y*5`  
  - E mentions a variable y, but it is bound to a value by `let` before the enclosed expression is evaluated
- `f(x)=if x=0 then 1 else x*f(x-1)`  
  - Recursion is perfectly allowable (why?)
- `f(4)`  
  - We could have no definitions, then E must be all constants
- `1+2*5-3`

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

$$function\ call: \left[ E_{1 \rightarrow \pi_{1}}, \ldots, E_{n \rightarrow \pi_{n}}, \pi_{1}, \ldots, \pi_{n}, \phi \right] \rightarrow \nu (\pi) = \nu (E_{1}, \ldots, E_{n})$$

**Evaluation of lazy parameters**

- If the parameter needs evaluation:

  $$identifiers: \left[ E_{x \rightarrow \pi_{x}} \right] \rightarrow \nu (where \ \pi(1) = [E_{x}])$$

- The same should be done for let variables