P4b: Grammar for MicroCaml AST

Expressions

```
e ::= v | x | e bop e | not e
    | let [rec]? x = e in e
    | if e then e else e
    | e e | fun x -> e

v ::= n | s | true | false | (A,λx.e)

bop ::= + | - | > | < | >= | <= | = | != | ^ | && | ||
```

Why a λ ?
It's just decoration, but it comes from the lambda calculus, which we will discuss in a few weeks

Mutop directives

```
d ::= def x = e;;
    | e;;
    | ;;
```

Expressions: Values, Not, Variables, Ifs

$$\frac{}{A;\ v \Rightarrow v}$$

$$\frac{A;\ e \Rightarrow \texttt{true}}{A;\ \texttt{not}\ e \Rightarrow \texttt{false}}$$

$$\frac{A;\ e \Rightarrow \texttt{false}}{A;\ \texttt{not}\ e \Rightarrow \texttt{true}}$$

$$\frac{A(x) = v}{A;\ x \Rightarrow v}$$

$$\frac{A;\ e1 \Rightarrow v1 \qquad A,x:v1;\ e2 \Rightarrow v2}{A;\ \texttt{let}\ x\ \texttt{=}\ e1\ \texttt{in}\ e2\ \Rightarrow v2}$$

Doesn't cover *recursion* – will discuss later

$$\frac{A;\ e1 \Rightarrow \texttt{true} \qquad A;\ e2 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3\ \Rightarrow v}$$

$$\frac{A;\ e1 \Rightarrow \texttt{false} \qquad A;\ e3 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3\ \Rightarrow v}$$

Expressions: Binops

$$\frac{A;\ e1 \Rightarrow v1 \qquad A;\ e2 \Rightarrow v2 \qquad v3 \text{ is } v1\ bop\ v2}{A;\ e1\ bop\ e2 \Rightarrow v3}$$

- Arithmetic ***bop*** operators +, -, /, * work on ints
  - as do relational operators <, >, …
- Operators =, != require both arguments to have the same type
  - … but only for ints, booleans, strings
  - Not supported on closures **(**A’**,λ*x*.*e*)**. Non-support makes logical sense, but also prevents implementation problems involving infinite loops, due to the use of references, shown later.
- Logical operators &&, || work on booleans only
- Concatenation ^ works on strings only

Expressions: Closures

$$\frac{}{A;\ \texttt{fun}\ x\ \texttt{->}\ e\ \Rightarrow (A, \lambda x.e)}$$

$$\frac{A;\ e1 \Rightarrow (A', \lambda x.e) \qquad A;\ e2 \Rightarrow v1 \qquad A', x{:}v1;\ e \Rightarrow v}{A;\ e1\ e2 \Rightarrow v}$$

- Implements lexical/static scoping
  - Captures the environment when closure is created; uses that (and nothing else) when called

Mutop Directives (no rec)

$$\frac{A;\ e \Rightarrow v}{A;\ \texttt{def}\ x\ =\ e\texttt{;;}\ \Rightarrow\ A,x{:}v\ ;\ v}$$

Doesn't cover *recursion* – we explain in 2 slides

$$\frac{A;\ e \Rightarrow v}{A;\ e\texttt{;;}\ \Rightarrow\ A;\ v}$$

$$\frac{}{A;\ \texttt{;;}\ \Rightarrow\ A;}$$

No value to return

- Judgment $A;\ d\ \Rightarrow\ A'\ ;\ vopt$
  - Returns updated environment, optional value

Let rec: Recursion

$$\frac{A;\ e1 \Rightarrow (A', \lambda x.e) \qquad v1 = (A'\{v1/x\}, \lambda x.e) \qquad A, x{:}v1;\ e2 \Rightarrow v2}{A;\ \texttt{let rec}\ x = e1\ \texttt{in}\ e2 \Rightarrow v2}$$

- We evaluate **e1** to a closure
  - If it's not a closure, it's the same as non-recursive let
- The second premise defines **v1** via a recursive equation
  - **v1** appears on the left and right-hand side of the =
- The solution is a fixed point: every occurrence of **x** in A' is **v1**, which can internally refer to itself

Implementing Recursion via References

$$\frac{\begin{array}{cc} r = \mathrm{newref}(0) & A, \mathbf{x}:r;\ \mathbf{e1} \Rightarrow \mathbf{v1} \\ \mathrm{update}(r, \mathbf{v1}) & A, \mathbf{x}:r;\ \mathbf{e2} \Rightarrow \mathbf{v2} \end{array}}{A;\ \texttt{let rec } \mathbf{x} = \mathbf{e1} \texttt{ in } \mathbf{e2} \Rightarrow \mathbf{v2}}$$

- We can implement the fixed point with OCaml references
  - Create a reference placeholder for ***x*** (use int *0* as a dummy val)
  - Extend A with that placeholder, and evaluate ***e1***
  - Update the placeholder to the resulting value ***v1***
- Once we do this, we can evaluate ***e2***
- We do something similar for recursive **`defs`**