COMP 321: Principles of Programming Languages, Fall 2021
Homework 2: Variables

In this assignment, you will do progress and preservation proofs and an implementation for a language with variable binding.

In particular, the language we work with will have

- number constants and addition (like last week's homework)
- string contstants (written like `"abc"`) and string append (written `s1 ^ s2`, e.g. `"A" ^ "BC"` should compute the string `"ABC"`)
- variable uses and let-binding, as discussed in this week's lectures

This language is called **E** in Chapter 4 and Section 5.2 and and Chapter 6 of PFPL, though

1 Progress and Preseravtion

Suppose we extend **E** with a primitive two-binding `let`, for example

```
let x be 7
    y be 8
 in
    x + y
end
```

should evaulate to 15. This is mostly the same as two nested single-binding `lets`:

```
let x be 7 in
  let y be 8 in
     x + y
  end
end
```

but for the two-binding version, the intended scoping rule is that `x` and `y` are both only in scope in the body of the `let`, and `x` is not in scope in the definition of `y`. For example, the nested single-binding lets

```
let x be 7 in
  let y be x + 1 in
     x + y
  end
end
```

are well-scoped, but the two-binding let

```
let x be 7
    y be x + 1 in
     x + y
end
```

is not. (This kind of `let` allows for parallel evaluation of the two let-bound expressions, if you heard about that in COMP 212, but we won't do that here.) As a formal abstract syntax tree, we write $\mathsf{let2}(e_1, e_2, x.y.e_3)$ for `let x be e1 y be e2 in e3 end`.

Task 1 (5%). Give a typing rule for let2, i.e. fill in the premises of

$$\frac{?}{\Gamma \vdash \mathsf{let2}(e_1, e_2, x.y.e_3) : \tau_3} \ \texttt{typing-let2}$$

Hint: use the typing rule for single-binding let in Figure 2 as a starting point.

Task 2 (15%). Give operational semantics rules for $\mathsf{let2}(e_1, e_2, x.y.e_3)$. Your semantics should evaluate the two bindings in left-to-right order, and should be call-by-value.

Task 3 (15%). For a language with variables, the progress theorem is:

For all e, τ, if $\cdot \vdash e : \tau$ then either e done or there exists an e' such that $e \mapsto e'$.

Here, the notation $\cdot \vdash e : \tau$ means that e has type τ in the empty context.

Progress is proved by rule induction on the derivation of $\cdot \vdash e : \tau$. Prove the case of progress for your typing rule `typing-let2` for $\mathsf{let2}(e_1, e_2, x.y.e_3)$ from Task 1.

Task 4 (15%). For a language with variable binding, the preservation theorem is:

For all e, e', τ, if $e \mapsto e'$ and $\cdot \vdash e : \tau$ then $\cdot \vdash e' : \tau$.

Preservation is proved by rule induction on $e \mapsto e'$. Prove the cases of preservation for your operational semantics rules from Task 2.

You may use the following lemmas:

- Inversion of typing: For all $e_1, e_2, x, y, e_3, \tau$, if $\cdot \vdash \mathsf{let2}(e_1, e_2, x.y.e_3) : \tau$ then *[the premises of the rule in your answer to Task 1]*.
- Weakening: For all $\Gamma, \Gamma', e, \tau, \tau_1, x$, if $\Gamma, \Gamma' \vdash e : \tau$ and $x \notin \Gamma, \Gamma'$ then $\Gamma, x : \tau_1, \Gamma' \vdash e : \tau$.
- Substitution: For all $\Gamma, \Gamma', e, x, \tau, \tau'$, if $\Gamma, x : \tau, \Gamma' \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$ then $\Gamma, \Gamma' \vdash e'[e/x] : \tau'$

2 Implementing Variables

In this problem, you will implement a type checker and operational semantics for **E**.

2.1 Substitution

The `syntax.sml` provides an implementation of the abstract syntax of the language.

Variable names are represented by strings. There is a function `freshName` that uses a global mutable memory cell to generate a "fresh" name, one that has never been returned by this function before; this is used for α-converting.

The types of the language (`typ`) are numbers and strings.

The expressions are variables, number constants, string constants, addition of numbers, concatenation of strings, and let-bindings.

Task 1 (20%). Implement a function

```
(* subst (e, (e',x)) implements
   the capture-avoiding substitution e[e'/x] *)
val subst : exp * (exp * name) -> exp
```

that implements capture-avoiding substitution for expressions in this language.

For reference, the mathematical definition of this function is in Figure 1. For these definitions, we use the formal abstract-syntax-tree style notation, rather than the concrete-syntax notation that I use in class. The abstract syntax is

$$e ::= x \mid \mathsf{num}[k] \mid \mathsf{str}[s] \mid \mathsf{plus}(e_1, e_2) \mid \mathsf{cat}(e_1, e_2) \mid \mathsf{let}(e_1, y.e_2)$$

which corresponds to the concrete syntax

```
e ::= x | k | "s" | e1 + e2 | e1 ^ e2 | let x be e1 in e2 end
```

You will receive 15/20 for this problem if your substiuttion function works correctly for substituting closed (no free variables) expressions, with no shadowing (i.e. every `let` binds a distinctly named variable — this is what the parser produces when it reads in a file), with the assumption that the variable being substituted for is different from all of the bound variables.

For full credit, you should also implement a correctly capture-avoiding substitution that works for open (has free variables) expressions that potentially do have shadowing. To do this, you will need to implement α-conversion, i.e. changing the names of bound variables. The easiest way to do this is to implement a function `swap(e,(x,y))` that swaps the two names `x` and `y` *everywhere* where they occur in the expression, including binding sites. For example, swapping `x` and `y` in the expression

```
let x be 5 in let y be 6 in x + y end
```

should give

```
let y be 5 in let x be 6 in y + x end
```

Then you can use swapping to "freshen" bound variables to make the $x \neq y$ and $y \notin e$ conditions on the definition for let-binding be true.

$$\frac{}{x[e/x] = e} \qquad \frac{x \neq y}{y[e/x] = y}$$

$$\frac{}{\mathsf{num}[n][e/x] = \mathsf{num}[n]} \qquad \frac{}{\mathsf{str}[s][e/x] = \mathsf{str}[s]}$$

$$\frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{\mathsf{plus}(e_1, e_2)[e/x] = \mathsf{plus}(e_1', e_2')} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{\mathsf{cat}(e_1, e_2)[e/x] = \mathsf{cat}(e_1', e_2')}$$

$$\frac{e_1[e/x] = e_1' \quad x \neq y \quad y \notin e \quad e_2[e/x] = e_2'}{\mathsf{let}(e_1, y.e_2)[e/x] = \mathsf{let}(e_1', y.e_2')}$$

Figure 1: Substitution for **E**.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{num}[n] : \mathsf{number}} \qquad \frac{}{\Gamma \vdash \mathsf{str}[s] : \mathsf{string}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{number} \quad \Gamma \vdash e_2 : \mathsf{number}}{\Gamma \vdash \mathsf{plus}(e_1, e_2) : \mathsf{number}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{string} \quad \Gamma \vdash e_2 : \mathsf{string}}{\Gamma \vdash \mathsf{cat}(e_1, e_2) : \mathsf{string}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}(e_1, x.e_2) : \tau_2}$$

Figure 2: Typing rules for **E**.

$$\frac{}{\mathsf{num}[n]\ \mathsf{done}} \qquad \frac{}{\mathsf{str}[s]\ \mathsf{done}}$$

$$\frac{}{\mathsf{plus}(\mathsf{num}[m], \mathsf{num}[n]) \mapsto \mathsf{num}[m+n]} \qquad \frac{}{\mathsf{cat}(\mathsf{str}[s], \mathsf{str}[t]) \mapsto \mathsf{str}[st]}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{plus}(e_1, e_2) \mapsto \mathsf{plus}(e_1', e_2)} \qquad \frac{e_1\ \mathsf{done} \quad e_2 \mapsto e_2'}{\mathsf{plus}(e_1, e_2) \mapsto \mathsf{plus}(e_1, e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{cat}(e_1, e_2) \mapsto \mathsf{cat}(e_1', e_2)} \qquad \frac{e_1\ \mathsf{done} \quad e_2 \mapsto e_2'}{\mathsf{cat}(e_1, e_2) \mapsto \mathsf{cat}(e_1, e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{let}(e_1, x.e_2) \mapsto \mathsf{let}(e_1', x.e_2)} \qquad \frac{e_1\ \mathsf{done}}{\mathsf{let}(e_1, x.e_2) \mapsto e_2[e_1/x]}$$

Figure 3: Operational semantics for **E**.

2.2 Implementing the Type Checker

Task 2 (20%). Next, in `check.sml` you will implement a type checker, which implements the rules in Figure 2.

Like last week's homework, given an expression `e`, the type checker should return a `result`, which is either `WellTyped t` if `e : t` or `IllTyped` if `e` does not have a type. This week, you do not need to provide error messages for the type errors.

Like last week, the final function `check` only needs to work on "closed" expressions, ones with no free variable occurences. However, the main work will be in a function `checkOpen` that checks an "open" expression (an expression with free variables) relative to a given *context* Γ. We represent contexts as lists of variable name/type pairs. E.g. the context $(x : \mathsf{number}, y : \mathsf{string})$ might be represented by a list `[("x",Number), ("y",String)]`. The order of entries in the list is up to you. You should maintain the invariant that a particular name occurs at most once in the list, but you can also assume that `exp` you are given does not do any shadowing (the support code that reads in concrete syntax strings will α-convert all let-bindings to use a unique variable name).

2.3 Operational Semantics

Task 3 (10%). Next, in `step.sml`, implement the progress function.

Like last week, given a closed well-typed expression `e`, the function `progess` should return `Done` if `e` is done, or return `Stepped e'` if there is an `e'` such that `e |-> e'`. Much of the code will be similar to last week, with new cases for `let`.

In Figure 3, we include the operational semantics of **E** for reference.

2.4 How to test

In the support code, we have supplied a parser for a concrete syntax for **E** as well as several functions to help you test your code (see the `Top` module)

```
    val loop_print : unit -> unit (* print the same expression back *)
    val loop_type  : unit -> unit (* just type check *)
    val loop_eval  : unit -> unit (* type check and show the final value when done
    val loop_step  : unit -> unit (* type check and show all steps of evaluation *)

    (* similar, but read a .exp source file *)
    val file_print : string -> unit
    val file_type  : string -> unit
    val file_eval  : string -> unit
    val file_step  : string -> unit
```

The `loop` versions run an interactive input loop for **E**, like SMLNJ does for SML. The `file` versions are like `use`'ing a file. We have provided one **E** file for you to load, `examples.exp`. Note that the final expression in this file is ill-typed, so it is OK if your `progress` function fails on it.

For example, once you're done, you will be able to do this to type check and step through the execution of an expression:

```
- Top.loop_step();
Exp>let x be 4 in x + x end;

let x24 be 4 in
    x24 + x24
end : num;

Press return:

4 + 4 : num;

Press return:

8 : num;
```

To run all of the provided tests, do

```
- Top.file_step "examples.exp";
```

and hit enter repeatedly, and check that each step of evaluation and its type looks right. You can also do

```
- Top.file_eval "examples.exp";
```

to quickly see the final values and

```
- Top.file_rtype "examples.exp";
```

to run only run the type checker, not the operational semantics (to test that task before doing the next one).