Homework 8: Operational Semantics

Due Thursday, April 16 at 9:55 am

Remember to write how long the assignment takes you on the top

This assignment is relatively short because we have an exam next week. Working on it over the weekend may help you prepare for the exam.

Progress rules reduce complex expressions to simpler ones. As described in PLAI chapter 23 and the lecture notes, semantics can be defined by a set of rules that look like either simple reductions of the form $a \Rightarrow a'$, or more complex conditional ones that look something like:

$$\frac{a \Rightarrow a' \qquad b \Rightarrow b'}{c \Rightarrow c'}$$

The statements above the bar are called antecedents. The rule can be applied whenever all of the antecedents are true. The statements below the bar are called consequents.

To evaluate an expression under an operational semantics, apply any rule whose pattern matches. If it is a conditional rule, the antecedents must also be satisfied. The program halts when there is no rule that can be applied. The only part of this syntax that is new to you is the bar in the conditional; we've been writing reductions all semester using the arrow notation.

Progress rules are useful both for defining a language (for automated evaluation), and for proving properties of the language (either by hand, or...as we'll see soon, automatically as well.) One can write a proof using progress rules by recursively nesting conditional rules or simply listing the non-conditional rules applied, along with the resulting expressions.

The questions on this assignment refer to the LIST language, which is defined in the appendix. Answer them according to a strict reading of the formal specification (i.e., not your intuition or something from a similar language like Scheme). Note that the specification is extended throughout the assignment.

1. What is the difference between **big-step** and **small-step** operational semantics? Which is used for the specification in the appendix? (6 pts)
2. Prove that the expression {cons {cons null null} null} reduces to $\left(\left(\hat{n},\hat{n}\right),\hat{n}\right)$. Proceed in the same way as an eval interpreter, from the outside in: start with the cons expression, and then in order to satisfy its antecedents, recursively apply other rules to the expression that you find. (8 pts)
3. It is silly that there are no literals for true and false in this language. Extend the specification (both syntax and semantics) to include these, using "true" and "false" as the syntax. (8 pts)
4. Extend the specification with a new form, "and", that accepts two subexpressions. This new form should evaluate to true if both subexpressions evaluate to true and false otherwise. (8 pts)

Challenge (optional; no points)

5. Let's extend the language with the new syntax:

```
<exp> ::= ... | { with <id> = <exp>  <exp>}  | <id>
```

The WITH expression is like a single-argument LET in Scheme: it creates a new environment that is a child of the current environment, binds the specified identifier to the evaluation of the first subexpression, and then evaluates the second subexpression in that environment. Assume

that <id> has been defined properly to exclude reserved keywords.

---

a) Complete the correct rules for evaluating a WITH expression by filling in the box:

$$\text{var:} \quad i,\mathcal{E}[i \quad v] \# \; v$$

$$\text{with:} \quad \frac{e,\mathcal{E} \Rightarrow v_1 \quad \boxed{?}}{\{\text{with } i = e_1\},\mathcal{E} \Rightarrow v_2}$$

Explain your solution and translate it into English.

b) Demonstrate correct application of WITH by proving that the following expression reduces to $\hat{t}$: {with x = {cons true null} {with y = {car x} y}}. (7 pts)

APPENDIX

LIST is defined by the following terms (a.k.a. syntax):

<exp> ::= { cons <exp> <exp>} | { car <exp> } | { cdr <exp> } | null | { empty? <exp> }

<val> ::= $\hat{n}$ | (<val>, <val>) | $\hat{t}$ | $\hat{f}$

and the following semantics:

car: $$\frac{L,\mathcal{E} \Rightarrow (f,r)}{\{\text{car } L\},\mathcal{E} \Rightarrow f}$$

cdr: $$\frac{L,\mathcal{E} \Rightarrow (f,r)}{\{\text{cdr } L\},\mathcal{E} \Rightarrow r}$$

cons: $$\frac{a,\mathcal{E} \Rightarrow a' \quad b,\mathcal{E} \Rightarrow b'}{\{\text{cons } a\ b\},\mathcal{E} \Rightarrow (a',b')}$$

null: $$\text{null},\mathcal{E} \Rightarrow \hat{\text{n}}$$

empty-true: $$\frac{e,\mathcal{E} \Rightarrow \hat{\text{n}}}{\{\text{empty? } e\},\mathcal{E} \Rightarrow \hat{\text{t}}}$$

empty-false: $$\frac{\neg\left(e,\mathcal{E} \Rightarrow \hat{\text{n}}\right)}{\{\text{empty? } e\},\mathcal{E} \Rightarrow \hat{\text{f}}}$$

The rotated "L" shape in the empty-false rule means "not," as in, "if *e* **does not** reduce to ..." One can continue to use the BNF quoting notation (which I'm using gray boxes for) within the rules as well, and doing so adds some clarity. It also adds a lot of notation and is tricky to typeset, so I'm not doing that here and don't expect you to quote terminals in your solutions. Do be careful about the hats on values, though: e.g., "null" is the null literal expression and "n" with a hat on it is the null value, and so on. This follows the syntax used in the textbook. If we had numbers, 2 would be an expression (numeral) and 2 with a hat would be a value (number).