Assignment 3: Parsing and Dataflow

Seth and co. 🪶

Due Wednesday, February 25th, 2026 (11:59PM)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own. Hand in your solutions on Gradescope. Please read the late policy for written assignments on the course web page.

Problem 1: Parsing (30 points)

The famed 15-150 mascot Polly has invaded the compilers course-staff and is in a glorious language battle with Opal the Otter!

Of course, Polly is not a fan of C0 instead preferring SML. But rather than changing the entire class to compile to SML, he decides to instead write a new language. Using context-free grammars Polly comes up with the language, C_0^{λ}, which combines the usability of lambda calculus with the safety of C. He specifies it with the following grammar (noting that **x** is an identifier token and that γ_3 denotes function application[1]).

$$\begin{array}{lcll} \gamma_1 & : & \langle \text{E} \rangle & \rightarrow \quad \textbf{x} \\ \gamma_2 & : & \langle \text{E} \rangle & \rightarrow \quad \lambda\, \textbf{x}\, \textbf{.}\, \langle \text{E} \rangle \\ \gamma_3 & : & \langle \text{E} \rangle & \rightarrow \quad \langle \text{E} \rangle\, \langle \text{E} \rangle \\ \gamma_4 & : & \langle \text{E} \rangle & \rightarrow \quad \textbf{(}\, \langle \text{E} \rangle\, \textbf{)} \end{array}$$

(a) Polly gets Opal to review his grammar, and Opal uncovers a number of problems. Show two ambiguities in the above grammar by providing for each ambiguity two possible parse trees for the same string.

(b) Clarabelle the 15-210 mascot is seeking to use Polly's revolutionary language, having found much success with SML. However, Clarabelle requires an unambiguous grammar. As compiler writers, help Polly fix his language and rewrite the grammar so it is unambiguous. **You must accomplish this by modifying currently existing rules, and/or adding up to one more rule for a total of 5 rules.**[2].

For each ambiguity you found in (a), identify which of the two parse trees will be accepted by your new grammar. The grammar should describe the same set of strings as the original grammar.

[1] For example, **f x** is the function **f** applied to the argument **x**.

[2] Hint: your new grammar may or may not have the precedence you'd expect from lambda calculus.

Problem 2: Dataflow (20 points)

Being a very functional parrot, Polly wants to come up with an algorithm to determine what variables in a program aren't needed, so she can eliminate them later. So, she went to Opal, who introduced her to the following inference rules.

In order to understand when variables are needed, we begin by isolating effectful instructions. We say a variable is *necessary* at a line if it is used in that line's effectful instruction, denoted by $\mathrm{nec}(l, x)$. Famililarly, $\oslash$ indicates a possible side-effecting binary operator.

$$\frac{l : x \leftarrow y \oslash z}{\mathrm{nec}(l, y) \quad \mathrm{nec}(l, z)}\ \textsc{Binop} \qquad \frac{l : x \leftarrow f(x_1, \ldots, x_n)}{\forall i, \mathrm{nec}(l, x_i)}\ \textsc{Func} \qquad \frac{l : \textbf{return}\ x}{\mathrm{nec}(l, x)}\ \textsc{Ret}$$

$$\frac{l : \textbf{if}\ (x\ ?\ c)\ \textbf{then}\ l_t\ \textbf{else}\ l_f}{\mathrm{nec}(l, x)}\ \textsc{Cond}$$

Figure 1: definition of the "nec" judgement

Now that it's apparent when a variable is *immediately* necessary at a line, we introduce the following judgements to propagate this information through a program.

$$\frac{\mathrm{nec}(l, x)}{\mathrm{needed}(l, x)}\ \mathrm{N}_1 \qquad \frac{\mathrm{needed}(l', x) \quad \mathrm{succ}(l, l') \quad \neg\mathrm{def}(l, x)}{\mathrm{needed}(l, x)}\ \mathrm{N}_2$$

$$\frac{\mathrm{needed}(l', x) \quad \mathrm{succ}(l, l') \quad \mathrm{def}(l, x) \quad \mathrm{use}(l, y)}{\mathrm{needed}(l, y)}\ \mathrm{N}_3$$

Figure 2: Definition of the "needed" judgement

The rule N_2 propagates neededness much like liveness, and, in essence, the rule N_3 states that if a variable is needed later on, then any variables used in its definition are also needed. Recall that $\mathrm{succ}(l, l')$ denotes that l' is an immediate succesor of l in the program, and $\mathrm{succ}(l)$ is the set of all successors of l.

Polly also recalled seeing a generic worklist algorithm for *backwards-may* dataflow over all lines of a program, and needs your help adapting it such that for every l, $\mathrm{In}[l] = \{x \mid \mathrm{needed}(l, x)\}$ after the algorithm terminates.

```
In[l] = ⊥ for all l
W = {all lines l}
while |W| ≠ 0:
  remove l from W
  Out[l] = ⨆_{l'∈succ(l)} In[l']
  Temp = Gen(l) ⊔ (Out[l] − Kill(l))
  if Temp ⋢ In[l]:
      In[l] = Temp
      W = W ∪ pred(l)
```

Figure 3: Generic backwards-may worklist algorithm

(a) For any l, $\mathrm{In}(l)$ is an element of a lattice. What is the underlying set X for this lattice, and what are $\bot$, $\sqcup$, and $\sqsubseteq$?

(b) For a given line l, define $\mathrm{Gen}(l)$ and $\mathrm{Kill}(l)$ in terms of $\mathrm{Out}[l]$, $\mathrm{use}(l)$, $\mathrm{def}(l)$, and $\mathrm{nec}(l)$ using set notation. Hint: This should follow from the definition of the needed judgement.

(c) Argue that each line can only be inserted into W $O(n)$ times, where n is the number of variables in the program.