Homework 2: Overview and Project Start

See the syllabus and listen in class for the due dates.
In this homework you learn more about the fundamental ideas of program analysis described in chapter 1 of the text, and you will get a start on your semester project.
If you wish, you can work in groups, and that is the default for the semester project part of this homework. However, be sure to follow the process described in the course's grading policy if you work in groups.

Read chapter 1 of our textbook: *Principles of Program Analysis* [1].

1. Consider the following abstract syntax of expressions in set theory.

e	$\in$	**Exp**	expressions
es	$\in$	**Exp**	expression sequences
n	$\in$	**Num**	numeric literal
s	$\in$	**SetExp**	set expression
x	$\in$	**Var**	variables

$$
\begin{array}{rcl}
e & ::= & \texttt{true} \mid x \mid s \mid a \mid e_1 \in e_2 \mid e_1 \subseteq e_2 \mid e_1 = e_2 \\
& & \quad \mid \neg e \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid e_1 < e_2 \mid \lambda x.e \mid e_1(e_2) \mid \textbf{let } x = e_1 \textbf{ in } e_2 \\
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
s & ::= & \{es\} \mid \mathcal{P}(s) \mid s_1 \times s_2 \mid \{e_0 \mid es\} \mid e_1 \cup e_2 \mid e_1 \cap e_2 \\
es & ::= & e_1, \ldots, e_n \ \texttt{(where } n \geq 0\texttt{)}
\end{array}
$$

Since this is abstract syntax, we will use parentheses to disambiguate expressions written in these forms. The notation for expressions is supposed to have its standard mathematical meaning. For example, $e_1 \times e_2$ can be used to multiply numbers and also to form the cross product of two sets. Note that $\lambda x \,.\, e$ is the function with formal x and body e. Also $\mathcal{P}(e)$ is the powerset of e; i.e., it is the set of all subsets of e.

The following will begin our exploration of type checking for set theory. We say that an expression in set theory *has a type error* if it does not make sense mathematically. For example, true $+ 3$ has a type error. Note that false *does* make sense mathematically, as does $3 = 2$, so "making sense" is not the same thing as "being true". What does *not* make sense is using an operator outside its domain, as in $\neg 4$ or $3 \in 3$.

(a) (5 points) Which of the following expressions have a type error? Note all that have type errors (there may be 0, 1, or more of them). Briefly explain why.

A. $3 < 2$
B. $\{y \mid y = 105\}$
C. $4 \in \mathcal{P}(\{-1, 0, 1\})$
D. $\mathcal{P}(\{y \mid y = 105\})$
E. $\{4\} \subseteq \mathcal{P}(\{-1, 0, 1\})$

(b) (5 points) Which of the following expressions have *no* type error? Briefly explain why.

A. $3 \in 33$
B. $\{(\lambda x \,.\, x)(y) \mid \neg(y \in y)\}$
C. **let** $q = 15$ **in** $\neg((q \times q) < q)$
D. $\{5, 0\} \times (\{2, 1\} + 10)$
E. $\neg((\lambda x \,.\, x)(3))$

Read section 1.6 (especially 1.6.2) of our textbook: *Principles of Program Analysis* [1].

2. [Concepts] [Semantics] Suppose we want to write a type system for set theory that prevents type errors. We wish to use judgments of the form $\Gamma \vdash e : \tau$, where Γ is a type environment (a map from variables (x) to type expressions, and types and type environments are given by the following abstract syntax.

Γ	$\in$	**TypeEnv**	type environments
τ	$\in$	**Type**	type expressions

   $\Gamma ::= x : \tau \mid \Gamma, x : \tau$
   $\tau ::= \texttt{Int} \mid \texttt{Boolean} \mid \texttt{Set(}\tau\texttt{)} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$

   Here are a few type checking rules for set theory to give you the idea.

   $$\text{(VAR)}\ \frac{}{(\Gamma, x : \tau) \vdash x : \tau} \qquad \text{(DROP)}\ \frac{\Gamma \vdash e : \tau'}{(\Gamma, x : \tau) \vdash e : \tau'} \qquad \text{(PLUS)}\ \frac{\Gamma \vdash a_1 : \texttt{Int}, \qquad \Gamma \vdash a_2 : \texttt{Int}}{\Gamma \vdash a_1 + a_2 : \texttt{Int}}$$

   $$\text{(APP)}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau, \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1(e_2) : \tau} \qquad \text{(LAM)}\ \frac{(\Gamma, x : \tau') \vdash e : \tau}{\Gamma \vdash \lambda x\,.\,e : \tau' \to \tau}$$

   $$\text{(POW)}\ \frac{\Gamma \vdash s : \texttt{Set(}\tau\texttt{)}}{\Gamma \vdash \mathcal{P}(s) : \texttt{Set(Set(}\tau\texttt{))}} \qquad \text{(SCOMP)}\ \frac{(\Gamma, x_1 : \tau_1) \vdash e_0 : \tau_0, \qquad (\Gamma, x_1 : \tau_1) \vdash e_1 : \texttt{Boolean}, \quad (\forall 2 \le i \le n\,.\,(\Gamma, x_1 : \tau_1) \vdash e_i : \texttt{Boolean})}{\Gamma \vdash \{e_0 \mid e_1, \ldots, e_n\} : \texttt{Set(}\tau_0\texttt{)}}\ \textbf{WHERE}\ n > 0$$

   (a) (5 points) Write a type checking rule in this style for expressions of the form $e_1 \subseteq e_2$. (This means, that the conclusion of the rule has in it an expression of this form.)

   (b) (10 points) Write a type checking rule for expressions of the form **let** $x = e_1$ **in** e_2.

3. (30 points) [Concepts] [BuildTools] For purposes of this problem, a *program analysis question* is a careful specification of what an analysis will determine at each program point. For example, section 2.1.1 of our textbook [1] says that "the available expressions analysis will determine:"

   > "For each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point."

   Other examples appear at the beginning of sections 2.1.2, 2.1.3, and 2.1.4. Note that these statements often rely on auxiliary definitions.

   For your own semester project, list the most important program analysis question(s) that the project will have to answer. You can list up to 3 of these for your project. If you have others, you might want to write them down for yourself, but turn hand in the most important three questions for your project. (What "most important" means is up to you, but you might decide that these are the questions that are necessary prerequisites for answering any other questions that your project needs answered.)

4. (100 points) [BuildTools] Write and test a parser and the construction of abstract syntax trees for the programming language (fragment) that you will be using in your semester project.

   We recommend that you use JastAdd (see `http://jastadd.org`). For working with JastAdd, you should read the on-line documentation, especially the reference manual. Also, it's useful to start with an existing JastAdd sample project (see `http://jastadd.org/projects`), such as PicoJava, and modify it to suit your language.

   You can use our WHILE language as a sample project also. The Linux/Unix command for anonymous checkout is:

```
svn checkout https://refine.eecs.ucf.edu/svn/proganalysis/WHILE/trunk WHILE
```

and the URL `https://refine.eecs.ucf.edu/svn/proganalysis` can be used with tools like Eclipse, but check out the `WHILE/trunk` if you are working with Eclipse. I have used the Subversive plugin in Eclipse; if it asks for a login and password, use the login "anonymous" and an empty password. On a PC you can use Tortise SVN or install cygwin and its svn support and then it will be like the *nix example above. On a Mac, I have used svn from a macports installation, see `http://www.macports.org`.

We recommend that you use a scanner generator (such as "lex", "flex", or "jflex") and a parser generator (such as "yacc" or "beaver" or ANTLR), since this will allow you to change your language more easily as your project progresses. To avoid parsing troubles, we recommend that you use reserved words to uniquely identify each statement (or expression, etc.). Note that the syntax of C and C++ have notorious parsing difficulties (to some extent inherited by Java), but many of these can be avoided by adding some additional keywords. (It is okay to do this for your project, even if it means that your project's language is slightly different than what you originally proposed.) If you have trouble with the grammar of your language, or with ambiguity in your grammar, see the instructor or send an email.

For this project, write both a parser and a JastAdd "aspect" that does unparsing. Also you will need tests to demonstrate that your parser and unparser work properly on several test cases.

Turn in the source files for the lexer, parser, and the JastAdd files including the abstract syntax (.ast) and the unparser (.jrag), as well as test cases and their output that show how the unparser works.

If you are using another compiler system (such as LLVM), turn in sources for a lexer and parser that parse just the subset of the language you are working with, and also code that shows how the ASTs are defined and that does unparsing for your subset.

References

[1] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.