Assignment 2

CSE 6341

Due: March 20, by 12:25 pm

This assignment contains 4 questions, for a total of 20 points.

1. (5 points) Consider the language used to illustrate code generation via attribute grammars. Suppose we extend it with do-while loops: $\langle c \rangle_1$ `::= do` $\langle c \rangle_2$ `while` $\langle be \rangle$

   Extend the attribute grammar discussed in class to handle such loops. Show the relevant rules for computing the necessary attributes.

   Illustrate your solution using the parse tree for string `i:=0; do i:=i+1 while ...` Show the values of attributes *temp*, *labin*, and *labout* at relevant parse tree nodes. Show the value of attribute *code* **only for the root node**. In your solution, display the value of $\langle be \rangle$.*code* as `[..be..]` — that is, as some sequence of assembly instructions that is not displayed in detail in the solution.

2. (6 points) In class we briefly discussed several helper functions used in the definition of the Lisp interptreter. These were

   - $bound(x, z)$: given a literal atom x and a list z of S-expressions, does there exist an element e of list z such that $car(e)$ is the same as x? If so, the resulting value is `T`; otherwise, it is `NIL`.
   - $getval(x, z)$: given a literal atom x and a list z of S-expressions such that $bound(x, z)$ is `T`, the resulting value is $cdr(e)$ where e is the first element of z such that $car(e)$ is the same as x.
   - $addpairs(x, y, z)$: given a list x of literal atoms, a list y of arbitrary S-expressions, and an list z of pairs (literal atom, arbitrary S-expression), produce a new list where the i-th element of x is paired with the i-th element of y, and the resulting pairs are prepended to list z (see example in the lecture notes). Precondition: the length of x is the same as the length of y (do not check this condition, just assume it is true).

   Write the Lisp functions that correspond to these three mathematical functions. That is, write `(DEFUN BOUND (X Z) ...)` etc. and make sure that you code is a correct Lisp function that can be successfully executed by the interpreter discussed in class.

3. (6 points) Consider the following subset of the language from our discussion of type systems:

   $\langle E \rangle$ `::= const` | `NIL` | `( PLUS` $\langle E \rangle_1$ $\langle E \rangle_2$ `)` | `( CAR` $\langle E \rangle$ `)` | `( CDR` $\langle E \rangle$ `)` | `( CONS` $\langle E \rangle_1$ $\langle E \rangle_2$ `)`

   Here `const` denotes a numeric atom representing an integer value.

   Suppose we wanted to check statically (i.e., *without* evaluating an expression), whether a given input expression satisfies the following condition: the expression and each of its subexpressions evaluate to either a *numeric atom* or a *list of numeric atoms*, but nothing else: not something that is not a list, not a list of lists, not a list containing a mix of atoms and lists, etc. Further, the following informal guidelines apply: (1) `NIL` is an (empty) list of numeric atoms; (2) the operands in `(PLUS ...)` evaluate to numeric atoms; (3) the operands in `(CAR ...)` and `(CDR ...)` evaluate to lists of numeric atoms; (4) the first operand in `(CONS ...)` evaluates to a numeric atom, while the second one evaluates to a list of numeric atoms.

   Part 1 (3 points): Write the *inference rules* for the typing relation $E : T$ to formalize the informal guidelines from above. Show the *derivation tree* for `(CAR (CDR (CONS 1 (CONS (PLUS 2 3) NIL))))` : *Nat*

   Note: do *not* worry about the case where the operand value of `CAR` or `CDR` at run time is an empty list. Your typing rules should not try to catch and reject this case—for example, they should determine that `(CAR (CDR (CONS 8 NIL)))` is well-typed.

   Part 2 (3 points): Write a type checker based on these inference rules, using an attribute grammar. Define an attribute *type* for $\langle E \rangle$ that represents the type of the expression, and design an arrtibute grammar based on this attribute. The grammars should reject all and only input expressions that are not well-typed. **Do not add any other attributes**; $\langle Expr \rangle.type$ is enough to solve the problem.

4. (3 points) Suppose we extend the language from the previous question with $\langle E \rangle$ `::=` ... | `(RAND` $\langle E \rangle_1$ $\langle E \rangle_2$`)`. Assume the run-time semantics is the following: randomly, with equal probability, we choose the first or the second subexpression. The chosen subexpression is evaluated and the result is used as the result of the entire `RAND` expression. Generalize your inference rules and attribute grammar from Problem 3 to ensure that the two subexpressions in a `RAND` have the same type (could be "numeric atom" or "list of numertic atoms"; either case is possible and allowed) and to associate a suitable type with the entire `RAND` expression.