Assignment 1

CSE 6341

Due: February 8, by 12:25 pm

This assignment contains 3 questions, for a total of 25 points.

Q1 (5 pts): Consider a slightly generalized version of the attribute grammar for simple expressions, based on the following context-free grammar.

$\langle S \rangle$ `::=` $\langle E \rangle$
$\langle E \rangle$ `::= const` | $\langle I \rangle$ | `(` $\langle E \rangle_1$ `+` $\langle E \rangle_2$ `)` | `let` $\langle I \rangle$ `=` $\langle E \rangle_1$ `in` $\langle E \rangle_2$ `end`
$\langle I \rangle$ `::= id`

Assume that terminal `const` represents integer constants and has an attribute *lexval* of type integer, representing the value of the constant. Similarly to `id`.*lexval*, the value of `const`.*lexval* is initialized by the lexical analyzer. (The lexical analyzer is also referred to as "scanner".) The evaluation rule for $\langle E \rangle$ `::= const` is as expected: $\langle E \rangle$.*val* `:=` `const`.*lexval*. The rest of the attributes and their evaluation rules are as discussed in class.

Consider the following input program:

```
let x = 3 in let x = ( x + let x = x in x end ) in ( x + x ) end end
```

Is this a valid string for the attribute grammar? If so, show the parse tree for this string and all attribute values at tree nodes. If not, explain why not.

Q2 (10 pts): Consider a simple language with Java-like labels

$\langle program \rangle$ `::=` $\langle c \rangle$
$\langle c \rangle_1$ `::=` $\langle c \rangle_2$ `;` $\langle c \rangle_3$ | `while` $\langle be \rangle$ `do` $\langle c \rangle_2$ `end` | $\langle label \rangle$ `:` `while` $\langle be \rangle$ `do` $\langle c \rangle_2$ `end`
| `break` | `break` $\langle label \rangle$
$\langle label \rangle$ `::= id`

In addition to regular `while` loops, this language also has *labeled* `while` loops. The labels are represented by nonterminal $\langle label \rangle$. Similarly, the language has `break` statements with labels. Such a `break` exits its surrounding loop that has that same label. For example, for

```
L: while b1 do while b2 do ....  break; ....  break L; ...  end end
```

the outer loop is labeled with label `L` and the inner loop is not labeled. The first `break` exits the inner loop. The second `break` exits the outer loop (i.e., the loop with label `L`).

A `break` with a label can be nested inside several loops that surround it (e.g., an innermost loop, which itself is nested in an outer loop, which itself is nested in another loop, etc.). Such a `break` only makes sense if there exists a surrounding loop with the same label. Another correctness condition is the following: for any labeled loop, there should not exist

a surrounding labeled loop with the same label. Write an attribute grammar to check these two validity conditions. Include a description of all attributes, evaluation rules for attributes, and conditions. Use *Cond* in your grammar to perform the necessary checks. Assume that ⟨*label*⟩ has a pre-defined attribute *name* giving the label's string name (e.g., "`L`" in the example); you do *not* have to define this attribute. **Any attributes you define should be inherited**.

Q3 (10 pts) Consider the following generalization of the type-checking example discussed in class:

⟨*intexp*⟩ `::=` ... | ⟨*intexp*⟩ `-` ⟨*intexp*⟩ | ⟨*intexp*⟩ `*` ⟨*intexp*⟩
⟨*boolexp*⟩ `::=` ... | ⟨*intexp*⟩ `=` ⟨*intexp*⟩
⟨*stmt*⟩ `::=` ... | `return` ⟨*intexp*⟩
⟨*formalslist*⟩ `::=` ⟨*formal*⟩ | ⟨*formal*⟩ `,` ⟨*formalslist*⟩
⟨*formal*⟩ `::=` `int id` | `bool id`
⟨*actualslist*⟩ `::=` ⟨*actual*⟩ | ⟨*actual*⟩ `,` ⟨*actualslist*⟩
⟨*actual*⟩ `::=` ⟨*intexp*⟩ | ⟨*boolexp*⟩

Here ... represents production alternatives already discussed in class. In the new grammar we add operators "subtraction", "multiplication", and "equality" for integer expressions, as well as a `return` statement to return an integer value from a function call. In addition, non-terminals ⟨*formalslist*⟩ and ⟨*actualslist*⟩—which were briefly discussed in class—are precisely defined.

Part 1 (2 pts): Write the necessary attribute grammar rules (if any) for production alternatives ⟨*stmt*⟩ `::= return` ⟨*intexp*⟩, ⟨*intexp*⟩ `::=` ⟨*intexp*⟩ `-` ⟨*intexp*⟩, ⟨*intexp*⟩ `::=` ⟨*intexp*⟩ `*` ⟨*intexp*⟩, and ⟨*boolexp*⟩ `::=` ⟨*intexp*⟩ `=` ⟨*intexp*⟩

Part 2 (3 pts): In the lecture notes, there is an informal description of an attribute *types* for ⟨*formalslist*⟩. Write the attribute grammar rules for computing this attribute.

Part 3 (5 pts): In the lecture notes, there is an informal description of an inherited attribute *expectedTypes* for ⟨*actualslist*⟩. At ⟨*intexp*⟩ `::= id (` ⟨*actualslist*⟩ `)` the value of this attribute for the ⟨*actualslist*⟩ node is obtained from information about function `id`, which itself is provided by attribute *tbl-stack*; helper function `paramtypes` looks up this information.

Describe informally how attribute *expectedTypes* should be used to perform type checking of the actual parameters at the call site. As discussed in class, the basic idea is to make sure that the expected type of each actual parameter is indeed correct (e.g., if the expected type is `INT`, the actual parameter should be an ⟨*intexp*⟩). Based on your approach, write all evaluation rules for this attribute, as well as for all other attributes of nodes of type ⟨*actual*⟩ (if necessary). **Do not introduce additional attributes for** ⟨*actualslist*⟩.