# CSCI 4627
# Spring 2002
# Answers to practice questions for exam 2

1. What is the difference between a synthesized attribute and an inherited attribute? Define each of those terms.

   A synthesized attribute is defined for a node v in terms of the attributes of the children of v. Any attribute that is not synthesized is called inherited.

2. One way to convert from intermediate code (three-address codes) to machine code is to generate code for each three-address code instruction separately. Optimizing compilers take a different approach. Explain the general approach that is used to create better code.

   Macro expansion is simple but inefficient. An approach that produces better results is symbolic execution, also called static execution. By doing a symbolic execution, the compiler can see what a program is doing, and can avoid redundant operations, such as fetching a variable whose value is already in a register.

3. Sometimes it is desirable to use only synthesized attributes *in your parser*. Is it always possible to do that, even if your semantics requires inherited attributes? If so, how can you deal with complex semantics while only using synthesized attributes in the parser. If not, what prevents you from using only synthesized attributes?

   You can use only synthesized attributes *in the parser* by building syntax trees there, and passing those syntax trees to semantic processing routines. The semantic processing is free to make many passes over the syntax trees, even though the parser makes only one bottom-to-top traversal of the parse tree.

4. Exercise 6.6 of the text contains a grammar that describes binary trees where each node of the tree has an integer label. It discusses an ordering requirement that is the same one required by binary search trees. Solve that exercise.

   Attribute *good* is true if the tree is ordered, and false if not. Attribute *smallest* is the smallest value in the tree, and *largest* is the largest value in the tree. In the case of an empty tree, largest is -infinity and smallest is +infinity. All of these attributes are synthesized.

| Production | Attributes |
|---|---|
| btree -> **nil** | btree.largest = -infinity<br>btree.smallest = +infinity<br>btree.good = true |
| btree -> **(** **number** $btree_1$ $btree_2$ **)** | btree.largest = max($btree_2$.largest, number.val)<br>btree.smallest = min($btree_1$.smallest, number.val)<br>btree.good = $btree_1$.good & $btree_2$.good & number.val > $btree_1$.largest & number.val < $btree_2$.smallest |

5. You are given the following (ambiguous) grammar for expressions.

```
expr -> expr + expr
expr -> expr * expr
```

```
expr -> NUM
expr -> VAR
```

where NUM and VAR are tokens. The lexer provides an attribute NUM.val that is the (integer) value of a NUM token. It also provides an attribute VAR.name that is the name of a variable. You would like to translate these expressions into instructions for a stack machine. The stack machine has the following instructions.

| | |
|---|---|
| PUSH_INT k | Push integer k onto the stack |
| PUSH_VAR k | Push the value of the variable at offset k onto the stack |
| ADD | Pop the top two numbers from the stack and push their sum |
| MULT | Pop the top two numbers from the stack and push their product |

The PUSH_INT instruction can handle any integer that the lexer will produces as an attribute of a NUM token. You have access to three support functions: get_var_offset(v) returns the offset where the variable named v is stored; gen1(I) generates single-part instruction I, and gen2(I,k) generates two-part instruction I, with parameter k.

Write semantic actions to be performed at each production that will generate code to compute a given expression and leave its value on the top of the stack. Do not worry that the grammar is ambiguous. That is a parsing problem, not a semantic one.

Just write an action for each production.

| Production | Action |
|---|---|
| expr -> NUM | gen2(PUSH_INT, NUM.val) |
| expr -> VAR | gen2(PUSH_VAR, get_var_offset(VAR.name)) |
| expr -> expr + expr | gen1(ADD) |
| expr -> expr * expr | gen1(MULT) |

6. This is the same as the preceding exercise, but instead of performing actions to generate the code, you would like to create the code sequence as an attribute of an expression nonterminal. Suppose that, in addition to get_var_offset(v), the following functions are available. single(I) produces, as its value, a sequence that represents the single-part instruction I. doub(I,k) produces a code sequence for a two-part instruction. Operator + can be used to compute the concatenation of two code sequences.

| Production | Equation |
|---|---|
| expr -> NUM | expr.code = doub(PUSH_INT, NUM.val) |
| expr -> VAR | expr.code = doub(PUSH_VAR, get_var_offset(VAR.name)) |
| expr -> $expr_1$ + $expr_2$ | expr.code = $expr_1$.code + $expr_2$.code + single(ADD) |
| expr -> $expr_1$ * $expr_2$ | expr.code = $expr_1$.code + $expr_2$.code + single(MULT) |