

CS 6353 Compiler Construction, Homework #3

1. Consider the following grammar for a very small csh-like language.

```
P → S; P | S;  
S → A | R | F  
A → id := E  
E → E + id | id | O | ( M )  
R → print id  
F → foreach id in L do { P }  
L → ( M ) | id  
M → MO | ε  
O → string | integer
```

A program consists of a sequence of statements. A statement can be an assignment statement (A), a print statement (R), or a foreach loop statement (F). The foreach loop has a loop list L. L is a list of objects which can be a string, an integer, or a real number. L can be an id of type “list” or can be an actual list of objects enclosed in (). In each iteration, an object from the loop list is assigned to the loop identifier (id) and P is executed. The following is a sample program for this csh-like language.

```
total := 0;  
count := 0;  
objList1 := (11 15 302 287 19 8);  
objList2 := (15 87 66 78);  
objList := objList1 + objList2;  
foreach num in objList do  
  { totoal := total + num;  
    count := count + 1;  
  }  
print objList;  
print total;
```

This language does not require variable declaration, but it does have a few semantic rules.

- (i) In the list ((M)), all objects should be of the same type, either all strings or all integers.
 - (ii) The loop identifier should not be modified in the loop body.
 - (iii) There should be at most two levels of nested foreach loops.
- (a) Use attributed grammar to check the semantic rules given above. In other words, you should add attribute rules to the given grammar. Only inherited and synthesized attributes are allowed, no global attributes.
 - (b) If your attributed grammar can be evaluated during parsing without an additional evaluation phase, then briefly discuss how it is possible. If your grammar cannot be evaluated at the parsing phase, then use the techniques we have introduced to modify your attributed grammar or the original grammar so that the semantic analysis can be done with parsing.
2. Consider the following attribute grammar for code generation for array references (the attribute grammar is not pure, it uses global variables).

The newtemp function returns a new temporary variable name. The name consists of a leading character t and a number. For the i-th call, the number is i. In other words, the temporary variables generated by a sequence of call to newtemp function are: t1, t2, t3,

The lookup function searches the symbol table to find the corresponding id entry. The input to the function is the name of the identifier and the field of the entry to be retrieved. The field arrayInfo of an array identifier stores all the definitions of the corresponding array. The field arrayInfo.arraySize is an array storing the maximum sizes of all the dimensions of the array. The field arrayInfo.elementWidth is the size of each element in the array.

```

S → L := E      { if (L.array = true and E.array = false) then
                  emit (L.place '[' L.offset ']' := E.place); ... }
L → A           { L.array := true; L.offset := A.offset;
                  L.place := A.place; }
A → Elist]     { A.place := arrayPlace; A.offset := offset;
                  emit (offset := offset * elementWidth; }
Elist → Elist1, E { Elist.dim := Elist1.dim + 1;
                  emit (offset := offset * arraySize[Elist.dim]);
                  emit (offset := offset + E.place; }
Elist → id[E   { Elist.dim := 0;
                  arrayPlace := lookup(id.name);
                  offset := newtemp();
                  p := lookup (id.name, arrayInfo);
                  arraySize := p.arraySize;
                  elementWidth := p.elementWidth;
                  emit (offset := E.place); }
E → id         { E.place := lookup(id.name); }
E → num       { E.place := num; }

```

Consider an given input statement: “member[w, x, y, z] := 103”. Follow the attribute grammar above and generate the three address code. A statement of the three address code may look like, for example: “t1 := x + y”.

3. Assume that we are using an LL parser. Consider the following attribute grammar which evaluates the input expression.

```

L → E          print(E.val)
E → E1 + T    E.val := E1.val + T.val
E → T          E.val := T.val
T → T1 * F    T.val := T1.val * F.val
T → F          T.val := F.val
F → ( E )     F.val := E.val
F → digit     F.val := digit.lexval

```

As you can see, it is not possible for the LL parsing algorithm to evaluate the attribute grammar (evaluate the input expression). Discuss why not.

4. Dragon book, second edition, Exercise 8.4.2. Consider the following program.

```

for (i=2; i<=n; i++)
    a[i] = TRUE;

```

```
count = 0;
s = sqrt (n);
for (i=2; i<=s; i++)
    if (a[i]) {
        count++;
        for (j=2*i; j<=n; j = j+1)
            a[j] = FALSE;
    }
```

- (a) Translate the program into three address code as defined in Section 6.2, dragon book.
- (b) Identify all basic blocks in your three address code.
- (c) Build the flow graph for the three address code.
- (d) Build the dominator tree and identify the back edges in your flow graph in (c).
- (e) Find the entry node and the set of nodes in the natural loop associated with each back edge identified in (d).