Final Exam for Compilers

January 8, 2020, close book

Be careful: there are seven questions in this exam.

1. (a) (10 points) Consider the following yacc file. Note that `P`, `Q`, `R`, `S` are nonterminals. `charA`, `charB`, `charC` are terminals representing the characters `a`, `b`, `c`, respectively. Assume we have a suitable lexical analyzer for this yacc file. What message(s) is/are printed assuming the input is "aaabbbbbbbcc"?
(b) (5 points) What might be wrong with this attribute grammar?

```
P : Q R S        { $2 = 1; printf("Rule 1: %d.\n", $2); }  ;
R : charB R      { $2 = $$ + 1;  printf("Rule 2: $$ = %d; $2 = %d.\n", $$, $2); }
R : charB        { $$ = 1; printf("Rule 3: %d.\n", $$); };
Q : charA Q      { printf("charA; Rule 4.\n"); };
Q : charA        { printf("charA; Rule 5.\n"); };
S : charC S;
S : charC ;
```

2. (15 points) What are the assembly code generated for for the following program? You can use RISC-V, x86, Java bytecode or some pseudo assembly languages (that you invent yourself) to answer this question. We will tolorate inprecise assembly code. You need provide enough explanation of each assembly instruction you use.

```
program fibonacci(output);
VAR a, b:integer;

FUNCTION fa (a:integer) :integer;
begin
         if a = 1 then
         fa := 1
```

```
          else  if a = 0 then
          fa := 0
    else fa:=fa(a-1)+fa(a-2)
end;

begin
    a:=5;
    b:=fa(a)
end.
```

3. (20 points) Please add attributes and attribute equations to the following grammar. Assume we have a suitable lexical analyzer for this grammar file. The resulting attribute grammar should build the symbol table of the input program. The symbol table is an array of id-structures. Assume the array is long enough. Also assume the input program is correct. You do not need to do any semantic checking, such as duplicate declarations of the same variable, etc. An id-structure is similar to the following C structure:

```
struct {
    int idtype;
    /* idtype = 1 means int; idtype = 2 means float. */
    string name;
}
```

The grammar is as follows:

```
P := DclList
DclList := Dcl
DclList := Dcl DclList
Dcl := Type VarList semicolon
Type := int
Type := float
```

```
VarList := Var
VarList := Var comma VarList
Var  := id
```

4. (15 points) What are frames (or activation records)? How are they used in a compiler? What are the contents in the frames?

5. (15 points) What are static chains and dynamic chains? Why do we need them in a compiler?

6. (10 points) Consider the following Java program and the attribute grammar. You need to change the `MULT` operator to the `EXP` operator (which means exponentiation). However there is one significant difference: `MULT` is left associative but `EXP` is right-associative. Furthermore, `EXP` has a higher precedence than `ADD`. Please write your modified attribute grammar and use an example to test your attribute grammar.

```
abstract class Expr { }

class Add extends Expr {
  Expr left, right;
  Add(Expr L, Expr R) { left = L; right = R; }
}

class Mul extends Expr {
  Expr left, right;
  Mul(Expr L, Expr R) { left = L; right = R; }
}

class Num extends Expr {
  int value;
  Num(int v) { value = v; }
}
```

The attribute of every terminal and nonterminal is an `Expr` object. Use yacc for building AST as follows (LR):

```
expr   ::= expr PLUS term    { $$ = new Add($1, $3); }
expr   ::= term              { $$ = $1; }
term   ::= term MULT factor  { $$ = new Mult($1, $3); }
term   ::= factor            { $$ = $1; }
factor ::= num               { $$ = new Num($1.val); }
factor ::= LPAR expr RPAR    { $$ = $2; }
```

7. (10 points) Please draw the abstract syntax tree for the following expression: (slide 106)

$$*(f(a*b)) = *(g(d-e)) + (*h)(n+m)$$