Compilers Comprehensive Exam Solutions (Fall 2001)

This is a 30 minute, closed book exam. The questions (except for the last) are based on the attached context-free grammar for a toy programming language. The grammar is intended for an LALR parser generator, such as YACC or Bison. It has some "semantic actions" but many have been omitted.

1. When this grammar is run through an LALR parser generator (e.g., YACC) 18 shift-reduce conflicts are reported.

   (a) Most of the conflicts involved the operators +, -, *, and /. What is the problem, and how can it be fixed easily without changing the grammar rules?
   This is a simple problem. The grammar is ambiguous because the relative precedence of the arithmetic operators is not specified. The grammar could be re-written in this case, but the problem says not to change it. So, the solution is to use the feature of YACC to specify precedence using declarations such as `%left`.

   (b) There are two conflicts that seem qualitatively different from the previous ones. One is a shift/reduce conflict when the next input is the token `ID`, and the two items in the offending state are:

   ```
   block  ->  BEGIN typedecls . constdecls vardecls stmts END   (rule 2)
   typedecls  ->  typedecls . typedecl   (rule 3)
   ```

   There is a similar problem with `constdecls` and `vardecls`. Explain what is going on here and suggest a way to fix it. Be specific.
   The problem has to do with lookahead. The parser needs to know whether it is at the end of the list of typedecls to determine whether it should reduce the empty production for constdecls or not. But the lookahead symbol in either case is same: "ID." If the next construct is a constdecl or vardecl, it should reduce the `constdecl -> /* empty */` *production. Otherwise, it should shift the ID (part of the right hand side of the typedecl production).*
   This cannot be solved by tweaking precedence, since the parser genuinely needs to shift some times and reduce others, based on insufficient information. My inclination would be to change the language to allow declarations to occur in arbitrary order. This eliminates the need to decide when the typedecls end, simplifies the grammar, and might make life easier for the programmer. There are many other language changes that could address the problem as well.
   However, we don't always have the freedom to change the language. We could generalize the language anyway, build a syntax tree, and write code to check that everything was in the right order in the tree.

2. Suppose you wanted to write a straightforward recursive descent parser for the same language (assume the parser does not use backtracking, and does not look ahead more than one token). What problems would the context-free grammar pose in its current form? First, would the conflicts in the previous question cause problems? Second, would there be additional problems? Explain your answers.

   Recursive descent parsing is based on LL(1) parsing. All of the above would still be problems, because LALR(1) parsing almost deals with a superset of the grammars LL(1) parsing can deal with. Ambiguity is definitely a problem.

An additional major problem is left recursion in the grammar. The grammar would have to be re-written to eliminate it.

Finally, the various declarations all begin with the same tokens, which will prevent it from being LL(1). This can be solved by adding a new non-terminal to represent the comment prefixes of the right-hand sides of these productions.

3. Describe what the `arithtype(t1, t2)` function should do to implement C-like type semantics for arithmetic operators, including error detection.

   `t1` *and* `t2` *should be pointers to records representing data types. Both types should be INTEGER or FLOAT, otherwise, it should report an error. The return type should be INTEGER if both are integers or FLOAT if one or both are FLOATS.*

4. Almost all machines have different instructions for integer and floating-point operations, and instructions to convert between them. Suppose you were implementing a simple code generator for this language. When would an instruction to convert an integer value to floating point value be emitted?

   If one argument to, say, + is INTEGER and the other is FLOAT, an instruction should be generated to convert the integer argument to a float before using a floating point add instruction to compute the value of the expression.

5. Describe what `checkassign` function should do, addressing the following questions:

   (a) What information should the arguments contain?

   (b) What errors should be reported? (It is important to list only those errors that would be detected here, as opposed to other productions in the grammar.)

   The arguments should be data structures describing variable declarations (whatever is built in the declarevar function call). We need to check that the lhs is declared to be a variable, not a types or constants (it's illegal to assign to constants). Then we need to check that the type of the assigned expression is compatible with the type that was declared for the lhs variable.

6. Are there circumstances under which performing the optimization of "common subexpression elimination" would slow down the compiled program? Explain your answer.

   Common subexpression elimination saves computation by storing and reusing subexpressions that appear more than once in the program. It is undesirable if the cost of storing the value is greater than the cost of computing it. This might be true because the cost of computing it is incredibly cheap (e.g., incrementing a variable), or because fast storage locations (registers) are heavily used for other purposes, such as loop index variables, at a particular part of the program.

```
%token BEGIN END TYPE CONST INTEGER FLOAT RECORD VAR ASSIGN ID INT_CONST FLOAT_CONST

%start program

%%

program         :       block

block           :       BEGIN typedecls constdecls vardecls stmts END
                ;

typedecls       :       typedecls typedecl
                |       /* empty */
                ;

typedecl        :       ID ':' TYPE type ';' { declaretype($1, $4); }
                ;

type            :       INTEGER { $$ = inttype(); }
                |       FLOAT { $$ = floattype(); }
                ;

vardecls        :       vardecls vardecl
                |       /* empty */
                ;

vardecl         :       ID ':' VAR type ';' { declarevar($1, $4); }
                ;

constdecls      :       constdecls constdecl
                |       /* empty */
                ;

constdecl       :       ID ':' CONST type const ';' { declarevar($1, $4); }
                ;

const           :       INT_CONST
                |       FLOAT_CONST

stmts           :       stmts ';' stmt
                |       /* empty */
                ;

stmt            :       block
                |       lhs ASSIGN expr ';' { checkassign($1, $3); }
                ;
```

```
lhs          :     ID { $$ = checkvardecl(lookupdecl($1)); }
             ;

expr         :     lhs { $$ = vartype($1); }
             |     const { $$ = consttype($1); }
             |     expr '+' expr { $$ = arithtype($1, $3); }
             |     expr '-' expr { $$ = arithtype($1, $3); }
             |     expr '*' expr { $$ = arithtype($1, $3); }
             |     expr '/' expr { $$ = arithtype($1, $3); }
             |    '(' expr ')' { $$ = $2; }
             ;

%%
```