COMP SOLUTIONS

# Solutions to Compilers Comprehensive, October 2000

This is a 60 minute open book exam (but do not use computers). All answers should be written in the blue book (not on the exam). Clear, organized answers to essay questions is important.

1. (10 points) In your blue book, indicate whether each language is LL(1), LR(1), both, or neither.

   (a) $S \rightarrow aSa|aSb|c$

   (b) $S \rightarrow Sa|bS|c$

   (c) $S \rightarrow aSa|bS|c$

   (d) $S \rightarrow aSa|bSb|\epsilon$

   (e) $S \rightarrow Sa|b|c$

   *Solution*

   There were two interpretations of this problem: (1) Can the grammar be rewritten into an equivalent form that has the desired property; (2) does the grammar have the desired property as it is given. I intended the second interpretation, but graded by guessing what interpretation the student had in mind.

   Here are the answers for the first interpretation:

   (a) $S \rightarrow aSa|aSb|c$ *both*

   (b) $S \rightarrow Sa|bS|c$ *both*

   (c) $S \rightarrow aSa|bS|c$ *both*

   (d) $S \rightarrow aSa|bSb|\epsilon$ *neither*

   (e) $S \rightarrow Sa|b|c$ *both*

   Here are the answers for the second intepretation:

   (a) $S \rightarrow aSa|aSb|c$ *LR(1) (not LL(1) because not left factored)*

   (b) $S \rightarrow Sa|bS|c$ *Neither (ambiguous)*

   (c) $S \rightarrow aSa|bS|c$ *LL(1) and LR(1)*

   (d) $S \rightarrow aSa|bSb|\epsilon$ *Neither (not a deterministic CFL)*

   (e) $S \rightarrow Sa|b|c$ *LR(1) but not LL(1) (left recursion)*

2. (20 points)

The following YACC grammar parses simple Polish expressions (e.g., `*+12+34`, which is equal to 21).

Show what actions need to be added to print the equivalent reverse Polish expressions (e.g., `12+34+*`). Assume that the lexical analyzer returns the numerical value of the number described by `NUM`

```
%token NUM

%%

S       :       R
        ;

R       :       NUM  { printf("%d ", $1); }
        |       '+' R R
        |       '*' R R
        ;
```

*Solution:*

```
S       :       R
        ;

R       :       NUM { printf("%d ", $1); }
        |       '+' R R { printf("+ "); }
        |       '*' R R { printf("* "); }
        ;
```

Now add actions to print Polish expressions, given reverse Polish:

```
S       :       R
        ;

R       :       NUM
        |       R R '+'
        |       R R '*'
        ;
```

*Solution.*

If we use the same actions from the previous solution, the result prints in RPN, not PN (the operators are printed after the reductions of the R's).

Here's something else that doesn't work:

```
...
|       { print("+ "); } R R '+'
```

The problem is a shift/reduce conflict: shift *NUM* (from $R \rightarrow NUM$) or reduce the implicit production $M_1 \rightarrow \epsilon$ that was added for the new action?

It is difficult to output the result on-the-fly during parsing because the last operator in the input has to be printed first. We have to parse the whole input before we can print anything. The solution before builds up the entire output in a string, then prints it at the end of parsing. Another approach would be to build a tree and then print it in PN.

```
S       :       R { printf("%s\n", $1); }
        ;

R       :       NUM
                {
                  char *str = (char *) calloc(1,12);
                  sprintf(str, "%d", $1); $$ = str;
                }
        |       R R '+'
                {
                  char *str = (char *) calloc(1, strlen($1) + strlen($2) + 1);
                  sprintf(str, "+ %s %s", $1, $2); $$ = str;
                }
        |       R R '*'
                {
                  char *str = (char *) calloc(1, strlen($1) + strlen($2) + 1);
                  sprintf(str, "* %s %s", $1, $2); $$ = str;
                }
        ;
```

3. (20 points) Describe some well-known compiler optimizations that could be usefully applied to the following code, and why they would improve the code. Be specific about what parts of the code would be improved and why. The most basic optimizations will count most heavily in grading this question.

```
struct s {
  int f1;
  double f2;
};

struct s A[100];

extern f(struct s *);

int main(void)
{
  int i;
  for (i=1; i<100; i++) {
    A[i].f1 = A[i-1].f1 - 1;
  }

  return f(A);
}
```

*Solution*

There is significant variation in terminology, and sometimes the same effect can be achieved from different avenues. Here is a sample answer.

Unoptimized, computing the address of `A[i]` requires generating code to do something like: `globalsaddress + Aoffset + 12 * 4 * fetch(stackpointer + localsoffset + ioffset)` where "globalsadress", "Aoff", "localsoffset" and "ioffset" are all constant. *Constant-folding* could compute `globalsaddress+Aoffset`, `localsoffset+ioffset`, and `12*4` before the program is run (at compile time and load time).

`stackpointer+localsoffset+ioffset` (getting the variable $i$) will appear twice, so *Common subexpression elimination* could be used to compute it only once (and save the result for later re-use). A sophisticated compiler could attempt to do algebraic transformations to make the address of A[i].f1 into a common subexpression (A[i-1].f1 is a constant offset from this), but re-arranging expressions in the right way is not necessarily easy.

The loop optimization *Reduction in strength* could be used to avoid multiplying by 48 every time the array is indexed in the loop (instead, 48 could be added to the array address each time).

If the loop were unrolled, there would be an opportunity to eliminate a different common subexpression, since the address of A[i] on one iteration is the same as A[i-1] on the next.

4. (10 points) This question asks for practical reasons to prefer one way of writing parser over another.

(a) Give three good reasons to write a recursive descent parser by hand, even though highly efficient automatic parser generators are freely available.

*Solution:*

- The grammar is simple, and I don't want to require people building the system to have YACC.
- No good parser generator is available (e.g., in the early days of Java).
- Greater flexibility, e.g., to look ahead more than one symbol or use other information to decide on an action.
- Error recovery may be more flexible and understandable.
- etc.

(b) Give three good reasons to use an LALR parser generator such as YACC instead of writing a recursive descent parser by hand.

*Solution:*

- It's easier, because parser construction is automatic.
- LALR(1) parsing is more powerful than recursive descent, which is basically LL(1). E.g., left recursion in the grammar works.
- The context free grammar is more readable, and is easier to make consistent with the source language specification.
- The resulting parser is likely to be faster.
- etc.