Computer Science Department
Stanford University
Comprehensive Examination in Compilers
1998

October 29, 1998

READ THIS FIRST!

1. You should write your answers for this part of the Comprehensive Examination in a BLUE BOOK. Be sure to write your MAGIC NUMBER on the cover of every blue book that you use.

2. Be sure you have all the pages of this exam. There are 4 pages.

   The exam takes 1 hour.

3. This exam is CLOSED BOOK.

4. Show your work, since PARTIAL CREDIT will be given for incomplete answers. For example, you can get credit for making a reasonable start on a problem even if the idea doesn't work out; you can also get credit for realizing that certain approaches are incorrect. On a true/false question, you might get partial credit for explaining why you think something is true when it is actually false. But no partial credit can be given if you write nothing.

Compilers

This exam is *closed book.*

1. (20 points) For each of the statements below, circle "true" or "false", as appropriate. Unless there is an explicit statement to the contrary, assume that all context-free grammars have *no* useless symbols. Some of these questions are intentionally tricky!

   true **false** Every LR(2) grammar is also LR(1).

   true **false** If a CFG with no useless symbols has a single nonterminal with both left-recursive and right-recursive productions, the CFG is ambiguous.

   true **false** If a CFG is ambiguous, its grammar cannot be LR(1). and $\alpha \neq \beta$

   true **false** If a CFG has productions $A \to \alpha$ and $A \to \beta$ and both α and β are nullable (i.e., derive the empty string), the CFG is ambiguous.

   true **false** Some unambiguous context-free grammars can have more than one derivation for the same input string.

   true **false** The language of a CFG is empty if and only if the sentence symbol is useless.

   true **false** For every CFG with useless symbols, there is an equivalent CFG with no useless symbols, if the language of the original CFG is non-empty.

   true **false** An LR(1) and SLR(1) parser for the same CFG will detect errors at exactly the same point in the parse for every input string.

   true **false** An SLR(1) parser never does more reduce actions than shift actions during a parse.

   true **false** Every context-free language has at least one CFG with no left recursion.

2. (7 points) Suppose that Lex or Flex were given regular expressions for different lexemes in the following order: a, ab^*a, a^+c, ca, and ba. Draw vertical lines to show the sequence of lexemes that would be recognized in the following input string using the longest lexeme rule as used in Lex or Flex (this rule says that, if several prefixes of the input string match patterns, the longest such prefix should be returned by the lexer).

   $a\ b\ a \mid a\ c \mid a\ b\ a \mid a\ a \mid b\ a \mid a$

3. (15 points) The following CFG is ambiguous:

$$
\begin{array}{ll}
0 & S \rightarrow E \\
1 & E \rightarrow E\%E \\
2 & E \rightarrow E\#E \\
3 & E \rightarrow E@ \\
4 & E \rightarrow id
\end{array}
$$

(a) The SLR(1) parse table below shows the multiple entries that would result. Write down the entries (row, column, and entry) that should be *deleted* to obtain a parser which gives @ the highest precedence, gives # the next highest precedence and makes it *left-associative* and gives % the lowest precedence and makes it right-associative.

	ACTION					GOTO
state	$	id	%	#	@	E
0		s6				1
1	acc		s2	s4	s7	
2		s6				3
3	r1		s2, r1	s4, r1	s7, r1	
4		s6				5
5	r2		s2, r2	s4, r2	s7, r2	
6	r4		r4	r4	r4	
7	r3		r3	r3	r3	

(b) Show the sequence of stacks and inputs that occurs when parsing the string "id#id%id@" using the resulting parse table. Your answer should be in the format shown below: each line should have the stack of states written horizontally with the top to the right, and the remaining input to be parsed.

Stack (top at right)	Input
0	id # id % id

4. (8 points) This question and the next are based on the stack machine instruction set summarized on the last page of the exam.

The following is a fragment of a context-free grammar for expressions in a programming language. Add actions in C for each production in the grammar to generate code that computes the value of the expression (just print the instructions to the standard output using `printf`). When this code is executed, it should have the effect of pushing the expression value on top of the stack without disturbing any of the old stack entries.

Code is generated by other parts of the grammar that aren't shown. In particular, the nonterminals `varname` and `literal` have productions, that, when reduced, generate code that leaves the value of the variable on the top of the stack. You may assume that all variables and values are one word long.

```
expr    :       expr '+' expr    { printf("add"); }
        :       expr '-' expr    { printf("sub"); }
        :       expr '*' expr    { printf("mul"); }
        :       varname
        :       '(' expr ')'
        :       literal
```

5. (10 points) The following is a fragment of a grammar for "while" loops. A while loop first tests whether its expression is true; if so, the loop executes its statement and repeats; otherwise, the loop exits. Rearrange the grammar and add reduce actions to generate code for the loop, using the stack machine instruction set described below.

You should assume that the grammar is being parsed using a LALR parser such as YACC. However, YACC has a useful feature where you can insert actions between symbols in the right-hand side of a production. *Do not do this feature.* We want you to add actions only at the ends of the productions, to be executed when the production is reduced by the parser. This requires *rewriting* the production into an equivalent form, so you can perform each action at the right time during the parse.

Assume that expressions are compiled as in the previous question. Assume you have a function `newlabel` which returns a unique integer. For example, to generate a new label and define it, you might say: `printf("L%d: ", newlabel())`.

Hints: The answer requires a small amount of code. You will need to create labels. Note that labels can appear as a target of a jump before the label is defined (the assembler for the stack machine deals with keeping track of the values of labels). Be careful about nested loops!

```
whileloop       :  WHILE expr DO stmt ;
```

Stack Machine Summary

Here is a summary of the relevant part of a stack machine instruction set, for use in the previous problems. All stack entries and addresses are one word long. This is assembly language, which is translated into binary machine code by an assembler.

Labels

Labels are of the form "L123" (the letter L followed by a number). A label is defined when it appears followed by a colon before another label or instruction, in which case it is bound to the address of the next instruction. A label may be used in any context where a number can appear, and it is interpreted as the address to which it is bound. A label can be used in the text before it is defined (this is useful for forward jumps).

Instructions

`add` pop two entries, push their sum

`sub` subtract the top element from the second-from-top element.

`mul` pop two entries, push their product

`jump <label>` unconditional jump to constant address

`branch_true <op>` Pop top of stack; if it is non-zero, jump to the location described by the operand.

`branch_false <op>` Same, but jump if stack value is 0.