Midterm practice

Exam Facts

Monday October 30 12:30pm – 2pm Terman Auditorium
(note the early start time and different location than our usual lecture room)

There is no alternate exam, everyone is expected to take the exam at the scheduled time. Local SITN students will come to campus for the exam, since that ensures you will be able to ask questions and receive any clarifications that might come up during the exam. SITN folks outside the Bay Area will take the exam at their local site. There is a link to a campus map up on the web page and some information about parking options for those of you coming from off-campus.

Format

The midterm will be an in-class 90-minute written exam. The exam is closed-book/closed-note, but you can bring one 81/2" x 11" sheet of paper filled on both sides with whatever facts you think will be helpful**.** This compromise was my solution to the open/closed book dilemma. Your sheet is supposed to lessen the anxiety of the closed-book exam (i.e. don't have to memorize all details, do have something to refer for the complicated topics) without requiring that the questions be as involved or difficult as an open-book setting usually warrants. Writing up your sheet is a good means of studying for the exam, too. You are encouraged to prepare your sheets in groups, share ideas about what to include, and so on.

Material

The midterm will cover material from all lectures prior to the midterm. This primarily means the first two compilation phases: lexical and syntax analysis. Material similar to the homework will be emphasized, although you are responsible for all topics presented in lecture and in the handouts. The sorts of things that you should be prepared to demonstrate knowledge of:

- Overview of a compiler— terminology, general task breakdown
- Regular expressions— writing and understanding regular expressions, equivalence between regex/NFA,/DFA, conversion among forms (Thompson's algorithm, subset construction)
- Lexical analysis— loop and switch implementations, scanner generators, using lex
- Grammars— Chomsky's hierarchy, parse trees, derivations, ambiguity
- Top-down parsing— LL(1) grammars, grammar conditioning (removing ambiguity, left-recursion, left-factoring), first and follow set computation, top-down recursive descent implementation, table-driven predictive parsing
- Shift-reduce parsing— building LR(1) configurating sets, constructing parser tables, tracing parser operation, shift/reduce and reduce/reduce conflicts, differences between LR, SLR, and LALR
- Comparisons between parsing techniques— advantages and disadvantages: grammar restrictions, parser code maintenance and future flexibility, runtime efficiency, error-handling, etc.

The rest of this handout is a collection of various problems culled from past CS143 exams and problem sets. I tried to provide a good variety so you could see a range of question formats. Solutions will be distributed in Wednesday's lecture.

Sample true/false questions

1) Indicate true or false for the following statements:

_____ The language {(), (()), ((()))} is not parsable by any LL(1) parser.

_____ Given an SLR(1) parser and an LALR(1) parser for the same grammar, both have the same number of states.

_____ A grammar is ambiguous if there are several derivations for the same string.

_____ NFAs can recognize more languages than DFA due to the added expressive power of non-determinism and ε-transitions.

_____ A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

_____ Empty entries in the Goto table of an LR(1) parser are not useful for detecting parse errors.

_____ An SLR(1) parser operates in O(n) time, where n is the length of the input string.

_____ Every LL(1) grammar is SLR(1).

_____ Even if the language of a CFG is regular, the grammar may not be LL(1).

_____ No right-recursive grammar can be LR(1).

_____ An SLR(1) parser will never shift an erroneous symbol onto the parse stack, i.e. only symbols that can possibly be reduced will ever be shifted.

_____ A table-driven top-down predictive parser is capable of parsing more grammars than a recursive-descent non-backtracking parser.

_____ Consider a grammar with no e-productions and no single productions (i.e. no productions with a single symbol on the right-hand side). For an input of n tokens, a bottom-up parser can make a maximum of 2*n reductions.

Sample short answer questions

2) Answer the following short answer questions. A few sentences is all it takes.

a) When working on pp1, many students are perplexed that the input `-123` is scanned as `T_MINUS` followed by `T_INTCONSTANT (value 123).` They believe a better way is to produce a single token `T_INTCONSTANT (value -123).` Why, generally speaking, is this a bad idea? Is there a way for the lexical analyzer to solve the problem you describe?

b) In some languages (Modual-2 and Ada come to mind), a procedure declaration is terminated by syntax that includes the procedure name. For example, a Modula-2procedure is declared:

```
PROCEDURE BINKY;
BEGIN
...
END BINKY;
```

Note the use of the procedure name `BINKY` after the closing `END`. Can such a requirement be checked by a parser? Briefly explain.

c) In Java, the list of type qualifiers includes additions such as `public static final synchronized` and `transient`. Any or all of those qualifiers can appear in a declaration, in any order, but each cannot be repeated. Consider an unambiguous grammar to generate all subsets and permutations of the Java type qualifiers. What is the relationship between the number of productions in your grammar and the number of options? If Java required that the options appear in a particular order would this simplify or complicate the grammar? Explain.

d) An LL(k+1) can recognize more grammars than an LL(k) parser. Construct a simple grammar that an LL(2) parser can parse that cannot be parsed by an LL(1) parser. Briefly explain why the grammar can be parsed by one and not the other.

e) Explain why you would not want the following three productions in your grammar, no matter what parsing technique you were using:

```
Decl    ←id | *Decl | Decl[int]
```

Sample questions on lexical analysis, regular expressions, scanning

3) A *flerp* is a sequence of lowercase letters in which any vowels appear in alphabetical order. Vowels may be omitted or repeated. Therefore, `acrimony` is a flerp, so are `zelenski` and `gtdwprttzdf` but `compiler` is not (i after o, e after i).

a) Assuming the input will only contain lower-case alphabetic letters, write a regular expression to recognize flerps. You may use any of the regular expression constructs supported by lex/flex.

b) Draw lines between the flerps that lex would find on the following input:

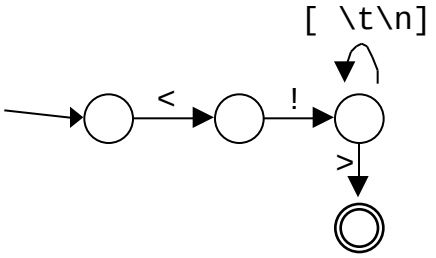
```
xdanjagbeontiutpuaceneau
```

4) Comments are a poorly understood and rarely correctly implemented portion of the HTML specification. The official rules are that a *comment declaration* starts with **<!** and ends with **>** and in between can have zero or more comments, separated by zero or more whitespace characters. A *comment* starts and ends with **- -** , and can contain anything except an occurrence of **- -** . Here are some examples:

Valid	**Invalid**
`<!-- blah blah -->`	`<! blah >`
`<!-- blah --  -- blah -->`	`<!--->`
`<!>`	`<!-- blah -- blah -- blah -->`
`<!--blah-blah-- ---->`	
`<!-- blah--->`	

Because HTML comments don't nest, they form a regular language.

a) Below, we have started building an NFA to recognize valid HTML comment declarations. Complete the machine by adding the missing portion to recognize comments within the comment declaration. This is an NFA, so you can use non-determinism and ε-transitions if desired. All transitions not explicitly shown are assumed to lead to the sinkhole.



b) Write a regular expression recognizes valid HTML comment declarations. You may use any of the regular expression constructs supported by lex/flex. Because the dash character has special meaning in certain contexts, you would need to escape it. Instead of worrying about this, just use the letter `d` for dash to avoid confusion in your expression.

Sample questions on LL(1) parsing

5) Consider this context-free grammar which recognizes valid function calls. A function call is an identifier followed by parenthesis enclosing a list of comma-separated parameters (the list can be empty). Each parameter is an integer, an identifier, or another function call.

```
F  ← id(L) | id()
L  ← L, P | P
P  ← id | int | F
```

a) Re-write the grammar as necessary to make it LL(1).

b) Compute first and follow sets for all non-terminals in your grammar.

c) Create an LL(1) parse table for your grammar.

d) Trace a parser of `Binky(4, Winky())` showing the stack, input, and parser action taken at each step.

6) There are one or more conflicts in the LL(1) table for the grammar as given below. For each conflict, identify the row, column, and conflicting entries. (Hint: you will need to compute the first and follow sets, but not the entire table).

```
S  ← aSb | A
A  ← bAb | cSc | ∂
```

Sample questions on LR(1), SLR(1), LALR(1) parsing

7) Consider the following grammar:

```
S  ← AB | Aa | CbCb | CbBa | Bc
A  ← d
B  ← d
C  ← a
```

a) Build the goto-graph of LR(1) configurating sets for this grammar.

b) Fill in the LR(1) parse table from your configurating sets.

c) Is this grammar LR(1)? Why or why not?

d) Is this grammar SLR(1)? Why or why not?

e) Is this grammar LALR(1)? Why or why not?

8) Consider the grammar below.

```
S  ←aSb
S  ←∂
```

a) Which is the smallest class of grammars this grammar fits into: SLR(1) LR(1) LALR(1)

b) Are the LR(1) and LALR(1) parse tables the same for this grammar?

b) Are the LALR(1) and SLR(1) parse tables the same for this grammar?

c)

9) What is the only US state capital that isn't graced with the presence of a McDonald's?