CSE 401 Midterm Exam

February 13, 2009

Name _____Sample Solution_____

The exam is closed book, closed notes, closed electronics, closed neighbors, open mind, ...

Please wait to turn the page until everyone has their exam and you have been told to begin.

If you have questions during the exam, raise your hand and someone will come to you.

Legibility is a plus as is showing your work. We can't read your mind, but we'll try to make sense of what you write.

1	/ 16
2	/ 16
3	/ 16
4	/ 16
5	/ 20
6	/ 16
Total	/ 100

Question 1. (16 points) The lexical- (scanner), syntactic- (parser), and semantic-analysis phases of a compiler front-end each process parts of the source program in particular ways and also check certain rules of the language being compiled. For each of the following possible language rules, specify which phase of the compiler should verify that a program conforms to that rule and why that part of the compiler is the best place for that check. If a check could be done equally well in more than one phase of the compiler, briefly discuss the tradeoffs between the alternative implementations. Keep your answers short and to the point.

(a) A function is called with the correct number of arguments.

Semantics. This requires comparing information about the declaration and call of the function, which cannot be done in the parser or scanner.

(b) Underscore characters (_) may appear in the middle of identifiers, but not at the beginning or end (i.e., `this_identifier` is legal, but `_this_one` is not).

Scanner. In a normal compiler, only the scanner sees individual input characters. The rest of the compiler deals with tokens or intermediate representations of the program.

(c) Every variable must be declared before it is used in the program (the classic C or Pascal rule).

Most likely semantics, since this can be detected in a pass over the AST at the same time other checks are being performed. It is possible to do this in the parser if the parser records information about identifiers as it sees their declarations and then checks each use to see if the identifier was previously declared.

(d) Assignment statements must end with a semicolon (;).

Parser. The scanner can't do this since it doesn't know the context in which the tokens appear. The semicolon is not present in the AST, so the semantics phase can't check for it.

Question 2. (16 points) Give an English description of the sets of strings generated by the following regular expressions. For full credit, give a description that describes the set without just rewriting the regular expressions in English. For example, if the regular expression is (a*b*)*, a good answer would be "all strings with 0 or more a's and b's in any sequence". A not so good answer would be "zero or more repetitions of zero or more a's followed by zero or more b's".

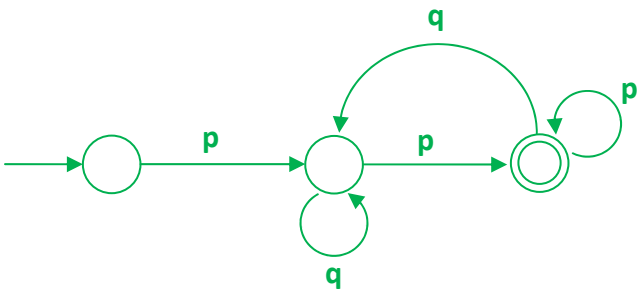
(a) (x|y)*x(x|y)

All strings of x's and y's ending with xx or xy.

(b) p(p|q)*p

All strings of p's and q's starting and ending with p.

(c) Draw a DFA that accepts the same set of strings generated by the regular expression in part (b).



Question 3. (16 points) Give regular expressions for the following sets of strings. You may only use basic regular expressions formed from characters and epsilon (ε),character classes denoting a single character ([…] and [^…]), concatenation (xy), alternation (x|y), repetition (x* and x+), and optional (x?). You may also give names to subexpressions (name=re) and use parentheses for grouping.

(a) Identifiers formed as follows: an identifier consists of one or more letters (a-z and A-Z), digits (0-9) and underscores (_). An identifier must begin with a letter and may not end with an underscore.

letter = [a-zA-Z]
digit = [0-9]

letter ((_)? (letter | digit)+)*

(b) Remote file identifiers of the form *user*@*hostname*:*filename*, constructed as follows. The parts of the identifier are made up of words, which are sequences of one or more letters and digits. The *user* part contains a single word. A *hostname* consists of one or more words separated by periods, like `www.google.com` or `attu.` A filename consists of one or more words separated by slash (/) characters with an optional leading and/or trailing slash (standard Unix conventions). The *user*@ part is optional and may be omitted. The entire *user*@*hostname*: part may be omitted, including the trailing colon. The *user*@ part may not appear unless the *hostname*: part is also included.

word = [a-zA-Z0-9]+

((word@)? word (. word)* :)? (/)? word (/ word)* (/)?

Question 4. (16 points) Consider the following grammar, which we previously saw in a homework problem:

S ::= (L) | x
L ::= L , S | S

(a) Complete the following table with the FIRST and FOLLOW information for the productions and non-terminals in this grammar.

Production	FIRST	FOLLOW
S ::= (L)	(	
S ::= x	x	
L ::= L , S	(x	
L ::= S	(x	,)

(b) Does this grammar have the LL(1) property that makes it suitable for top-down parsing? Briefly justify your answer.

No. The productions for L have problems. The first one is left-recursive and can't be used directly. Also, the FIRST sets of the two productions for L are the same.

(c) Describe how to fix this grammar, if necessary, to make it suitable for LL(1) parsing. Be specific.

The standard fix would be to rewrite the productions for L as follows:

L ::= S Ltail
Ltail ::= , S Ltail | ε

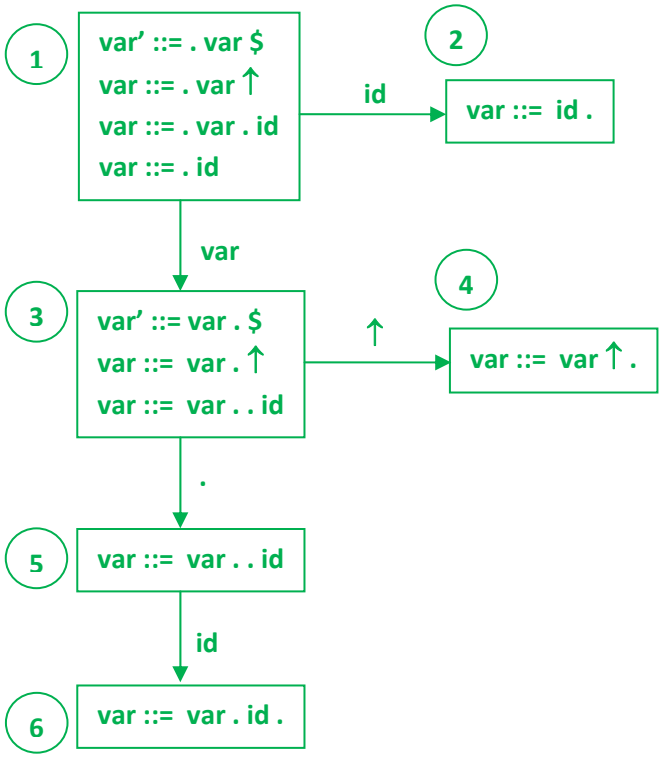
Question 5. (20 points) The Pascal programming language included pointers and records (structs), just like C. As in C, `x.y` indicated field selection. But Pascal had a much saner grammar for pointer variables. A Pascal pointer dereference was indicated by an arrow to the right of the variable. The Pascal expression `p↑` meant the same as `*p` in C, and the C expression `p->f` (which is equivalent to `(*p).f`) would be written as `p↑.f` in Pascal.

Here is a grammar for Pascal variables involving field selection and pointer dereferencing. The symbol `id` is a terminal representing a simple identifier. *var*' is the additional non-terminal inserted to handle the end-of-file marker $.

0. *var*' ::= *var* $
1. *var* ::= *var* ↑
2. *var* ::= *var* `.id`
3. *var* ::= `id`

On the remainder of this page and (if needed) the next page, do the following:

a) Construct the LR(0) state diagram for this grammar,
b) Construct the LR(0) parse table corresponding to your diagram from part (a), and
c) Indicate whether this grammar is LR(0). Justify your answer. ⇦ DON'T forget to do this part!



Question 5. (cont.) Additional space for your answer if needed. DON'T FORGET to explain whether the grammar is or is not LR(0), and why. Grammar repeated for reference:

0. *var*' ::= *var* $
1. *var* ::= *var* ↑
2. *var* ::= *var* `.id`
3. *var* ::= `id`

	id	↑	.	$	var
1	s2				g3
2	r3	r3	r3	r3	
3		s4	s5	acc	
4	r1	r1	r1	r1	
5	s6				
6	r2	r2	r2	r2	

Yes, the grammar is LR(0) because there are no conflicts in any of the table entries.

Question 6. (16 points) Java includes a conditional expression just like the one in C, C++, and other languages. A conditional expression has the form *exp0* ? *exp1* : *exp2*. The value of a conditional expression is either the value of *exp1* or *exp2* depending on the value of *exp0*. If *exp0* is true, then the value of the conditional expression is the value of *exp1*, and *exp2* is not evaluated. If *exp0* is false, then the value of the conditional expression is the value of *exp2*, and *exp1* is not evaluated.

Below is an abstract syntax tree for the assignment statement `max = x>y ? x : y;`. Describe the type checking needed in the semantics phase of the compiler to verify that there are no type errors in this assignment statement. You can assume that the identifiers are declared elsewhere and that their types can be retrieved from an appropriate symbol table entry.

A good way to show your answer would be to write beside each node what type checking needs to be done at that node (if any), and indicate the type of the resulting (sub-)expression represented by that node (if any).

The diagram below shows only the minimal checks needed and doesn't include details like the result types of the operand nodes which are, of course, actually present. One thing to note: all of the checks have to be local. In particular, checking of the ?: node is independent of its use as the right hand side of the assignment.