CSE 401 Midterm Exam 2/19/10

Name _________________________________

There are 7 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed books, closed notes, closed calculators, etc., except that you may have one sheet of paper with handwritten notes on it.

Please wait to turn the page until everyone is told to begin.

Score _________________

1 _______ / 18

2 _______ / 24

3 _______ / 16

4 _______ / 10

5 _______ / 8

6 _______ / 8

7 _______ / 16

Question 1. (18 points) In FORTRAN, a floating-point numeric constant must contain a decimal point or an exponent or both to distinguish it from an integer. The exponent uses the letter `E` or `e` to indicate a single-precision constant and the letter `D` or `d` to indicate double-precision. A floating-point constant without an exponent is taken to be single precision. Some **examples** of floating-point constants: `3.14` `1.` `1.0` `01e2` `6.023D23` `3.5E-12` `1.0d+6` `.1e-001` `000.000d00` Some examples that are **not** floating-point constants: `17` (no decimal point or exponent), `1,000,000.00` (contains commas), `-17.0` (unary minus operator followed by a constant), `.e02` (no digits before the exponent), `e02` (an identifier, not a number).

(a) (9 points) Give a regular expression for FORTRAN floating-point constants as described above. You may only use basic regular expression operators (concatenation `rs`, choice `r|s`, Kleene star `r*`) and the additional operators `r+` and `r?`. You may also specify character sets using the notation `[abw-z]`, and sets excluding specified characters `[^aeiou]`. Finally, you can name parts of the regular expression, like `vowel=[aeiou]`.

(continued on next page)

Quesiton 1. (cont.) (b) (9 points) Draw a DFA (Deterministic Finite Automata) that recognizes FORTRAN floating-point constants as described in the problem and generated by the regular expression in your answer to part (a).

Question 2. (24 points) The you're-probably-not-surprised-to-see-it LR-parsing question. Here is a tiny grammar that gives the essence of expressions with optional parentheses and optional casts preceding the expression.

0. *exp′* ::= *exp* $
1. *exp* ::= `id`
2. *exp* ::= `(` *exp* `)`
3. *exp* ::= `( id )` *exp*

(a) (18 points) Draw the LR(0) state machine for this grammar. You do not need to write out the parser tables or first/follow/nullable sets, although you can do that if it helps you to answer the remaining parts of the question on the next page.

(continued on next page)

Question 2. (cont.) Grammar repeated from previous page for reference.

0. *exp′* ::= *exp* $
1. *exp* ::= `id`
2. *exp* ::= `(` *exp* `)`
3. *exp* ::= `( id )` *exp*

(b) (3 points) Is this grammar LR(0)? Why or why not?

(c) (3 points) Is this grammar SLR? Why or why not?

Question 3. (16 points) Parsing tools. In CUP and similar tools we can use precedence declarations to resolve ambiguities and precedence issues in grammars. Here is a CUP specification for integer expressions with `PLUS` (`+`), `TIMES` (`*`), and `POWER` (`^`), where `PLUS` and `TIMES` associate to the left (i.e., `a+b+c` means `(a+b)+c`) and `POWER` (exponentiation) associates to the right (i.e., `a^b^c` means `a^(b^c)`). As usual, `TIMES` has higher precedence than `PLUS`, and `POWER` has the highest precedence. `INT` is the terminal symbol for integers.

```
precedence left PLUS;
precedence left TIMES;
precedence right POWER;

expr ::= expr PLUS expr | expr TIMES expr
      |  expr POWER expr | INT ;
```

For this problem, give an unambiguous context free grammar without precedence declarations that generates expressions with the same precedence and associativity as in the CUP specification above. You only need to give ordinary grammar rules – they do not need to be a properly formatted CUP specification.

(Hint: you may find it useful to introduce additional non-terminals into your grammar.)

Some short questions.

Question 4. (10 points) Compilers almost always do parsing and semantics/type checking in separate phases of the compiler. Give **two distinct** reasons why this separation into phases is done. Your reasons could be technical examples of things that can only be done in one phase but not the other, or engineering reasons why this is a good design, or similar persuasive arguments.

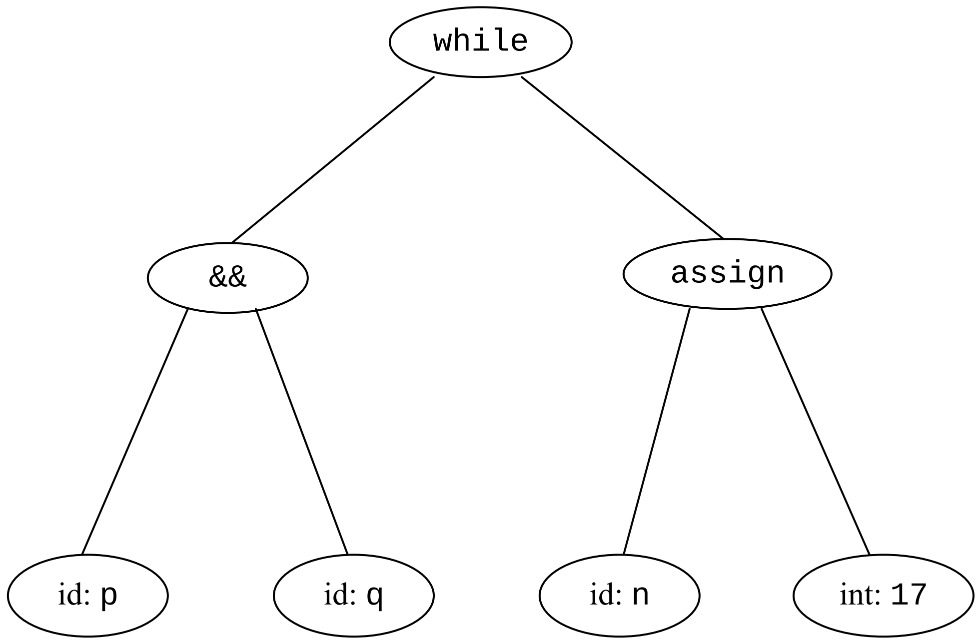
i)

ii)

Question 5. (8 points) We suggested using at least two separate passes over the AST to check semantics and types in a MiniJava program. Why not do it in one pass? Is there some technical reason why two passes are needed or at least very helpful?

Question 6. (8 points) In full Java, the declaration of a field in a class can be preceded various modifiers. The possibilities are: `public, protected, private, static, final, transient,` and `volatile`. These modifiers may appear in any order. A compiler is required to check that in any single declaration no modifier appears more than once, and that at most one of the access modifiers `public`, `protected`, or `private` is used. Would it be best to put this check in the scanner, the parser, or in the static semantics part of the compiler? Give a technical justification for your answer.

Question 7. (16 points) Abstract syntax and semantics. Here is the abstract syntax for a fragment of a MiniJava program.



(a) (4 points) What is the Java source code that corresponds to this abstract syntax? i.e., what is the original concrete syntax fragment that the scanner and parser read to produce the above tree, including all necessary punctuation to make it legal Java code?

(b) (12 points) What checks need to be performed in the static semantics/typechecking phase of the compiler to verify that this abstract syntax is, in fact, legal and contains no type errors or other static semantics problems? A good way to show your answer is to annotate the above diagram to show the checks that need to be performed at each location in the AST. Or you can write your answer below.