Q 1. This one asked you to think of the Core Tokenizer as a parser and asked you to write down the BNF grammar of the language that the Tokenizer was a parser for.

A surprising number of people said this could not be done because the Tokenizer didn't know what a legal Core program looked like. True, but the Tokenizer *does* know what a legal token stream looks like; e.g., the stream "proggram end +" is illegal but "program end +" is legal. So the correct answer is to write down the grammar for the langugage of legal token streams. It would look like:

```
<tokenStream> ::= <token> | <token> <tokenStream>
```

⟨*tokenStream*⟩ is the starting non-terminal; and we have to write the BNF for ¡token¿:

```
<token>    ::= <keyword> | <opSymbol> | <int> | <id>

<keyword>  ::= program | begin | end | ...

<opSymbol> ::= + | * | > | ...

<int>      ::= ... see the original Core grammar, productions 20, 21

<id>       ::= ... see the original Core grammar, productions 18, 19
```

The above doesn't handle white-space between tokens. It turns out, it is a bit involved to take care of the white-space requirement between pairs of tokens, given that if a token is a special symbol, white-space immediately before and immediately after that token are optional. The approach would be to first, change <token> to correspond only to <int>, <id> and <keyword>. Second, have two kinds of <tokenStream>, call them <opSymbolTokenStream> and <tokenStream>, respectively; the <opSymbolTokenStream> represents a token-stream that follows a <opSymbol>; <tokenStream> represents a token-stream that follows one of the other three kinds of tokens. Third, define the productions for these two non-terminals as follows

```
<tokenStream> ::= <ws> <token> | | <ws> <token> <tokenStream>
                | <ows> <spSymbol> | <ows> <opSymbol> <opSymboltokenStream>
<opSymboltokenStream> ::= <ows> <token> | <ows> <token> <tokenStream>
                | <ows> <spSymbol> | <ows> <opSymbol> <opSymboltokenStream>
```

Here, <ws> stands for one or more white-spaces; <ows> stands for *zero* or more white-spaces. That pair of productions represents all possible cases: a <tokenStream> starts with <ws> unless its first token is a <spSymbol> in which case the white-space is optional; following the first token, we may have another token-stream in which case, that stream will be either a <spSymboltokenStream> or <tokenStream> depending on whether the first token is a special symbol or not. The production for <spSymboltokenStream> is similar. With those two productions, we ensure that there is at least one white-space between each pair of tokens unless at least one of them is a special-symbol in which case the white-space is optional.

Some students seemed to think that since the Tokenizer doesn't take care of the values of ⟨*int*⟩'s or the names of the ⟨*id*⟩'s, this could not be done. This is wrong for two reasons; first, the Tokenizer *does* account for

these, that is how it is able to provide the operations idName() and intVal(); second, even if didn't, that would not be relevant because that has nothing to do with whether the grammar can be written or not.

A few students did everything right but did not include ⟨*tokenStream*⟩ as part of the grammar. This is incorrect but what exactly is wrong with it? Think about that; and we can discuss it on Piazza.

2. This one asked you to explain what each of the three types of errors, context-free (c.f.), context-sensitive (c.s.) and runtime (r.t.), were; and which of them could appear in a Java program and, in each case, either provide an example of it appearing or explain why it could not appear.

C.f. errors are ones that are violations of one or more requirements expressed in the BNF of the language. An example would be a missing semi-colon at the end of an assignment. C.s. errors are violations of syntactic requirements that, because of their nature, cannot be expressed in BNF. In terms of parse-trees, these requirements impose conditions on what may or may not appear in one part of the tree based on what does or does not appear on a separate part of the tree. R.t. errors are semantic errors that involve the *meaning* of programs, in other words, things that are determined during execution. (Some students labeled c.s. errors as semantic. That is incorrect because you could easily make up a language for which there is no semantics but has c.s. requirements. For example, consider the language with the following BNF grammar:

```
<ls> ::= <as> <bs> <cs>
<as> ::= a | a <as>
<bs> ::= b | b <bs>
<cs> ::= c | c <cs>
```

Here, a ⟨*ls*⟩ consists of a string of a's followed by a string of b's followed by a string of c's. If we add the requirement that the number of a's in ⟨*ls*⟩ must be equal to the number of b's in that ⟨*ls*⟩ which must, in turn, be equal to the number of c's in ⟨*ls*⟩, it turns out that *cannot* be captured in a BNF grammar and that would be a c.s. requirement. Obviously, there is no semantics here because this is just a string. I know that many authors who should know better call c.s. requirements "semantics" but that is incorrect because no semantics need be involved.)

A Java program may involve any of these errors. For example, an assignment statement with an expression such as "x + y" on the left of the "=", instead of an ⟨*id*⟩, is a c.f. error. Use of an undeclared variable in the expression in the assignment statement is a c.s. error. And if there are no c.f. or c.s. errors but, during execution, one of the variables in the ⟨*exp*⟩ on the right of the assignment statement was not initialized prior to the time that the ⟨*exp*⟩ is evaluated, that would be a r.t. error. (By the way, some people seemed to think that Java system means the Java runtime system, i.e., the JVM, etc. That is not true; Java includes the Java compiler which translates the Java source program into byte-code and will check for c.f. and c.s. errors; and the runtime system which will catch the r.t. errors.)

3. This asked about eIds[] and idCount which we used in the Id class to keep, in one place, a collection of pointers to existing Id objects and a count of the number of those objects. The question was whether you would need eIds[] and idCount once parsing of the Core program was complete.

The answer is "no" because the purpose of eIds[] (and idCount) was to allow us to see whether a given Id had previously been declared. For example, in ParseAssign(), we need to check that the ⟨*id*⟩ on the left had, in fact, been declared; we do this by comparing the name of the current ⟨*id*⟩ (which we obtain by calling

idName() of the Tokenizer) with the names of each of the Id objects that we have seen so far. The only thing that eIds[] contains is pointers to those Id objects; the names of the individual Id objects is stored inside each Id object. And when we find a match, we add a pointer to that particular Id object in the Assign object that ParseAssign() constructs. So once the parsing is complete, the Assign object has a pointer to the relevant Id object *without* having to go through eIds[]. Therefore, there is no more need for eIds[] (or idCount either).

4. This was about OO polymorphism and exactly how it works. We discussed this in some detail in the class, slides 58, 59 and 60. Slide 60 is especially important since it precisely identifies the problem. In the code outline on that slide, ep is a pointer to an Exp object which means it could point to an object of type Expi or Exps or Expp; and, correspondingly, the call to evalExp() that is applied to the object that ep points to must result in the execution of the evalExp() defined, respectively, in the Expi or Exps or Expp class. But there is no conditional statement in the code. There is simply a call that says, `"ep -> evalExp()"`. And at the time that the code is compiled, we, i.e., the compiler, cannot really determine which particular one of these three types of objects ep will be pointing to when we compile the code.

Why *can't* the compiler determine which type of object ep will be pointing to? Because, in the code outline on slide 60, it is possible that the code that is indicated by "..." may be conditional code that, depending on some value to be input from the input stream, sets ep to point to an object of type Expi or Exps or Expp. So no matter how clever the compiler is, there is no way for it to determine which type of object ep will point to when control, at runtime, gets to the statement `"ep -> evalExp()"`. And the question asked, so how exactly does this work? Meaning, what does the compiler do and what happens at execution time?

The answer is that the code that the compiler produces corresponding to a call to "new" to construct an object that is an instance of any class, such as Expp, etc., that contains one or more virtual, i.e., abstract methods, will allocate space not only for the member variables of that class but also locations, at the start of the memory allocated for the object, in which the addresses of the various concrete methods defined in the class corresponding to each of the virtual methods defined in the base class will be stored. Thus the code the compiler produces for a call such as "new Expp(..)" will result (when this code is executed at runtime) in *three* words of memory being allocated, the last two being for the member variables (called e and f in the code on page 59) and the first is set to contain the address of the evalExp() method defined in the Expp class. If we had "new Exps()", the code produced by the compiler would be similar except that the first location will contain the address of the evalExp() method defined in the Exps class; etc. And the code the compiler produces for the call, `"ep -> evalExp()"` is *not* conditional code of any kind; instead, the code produced, when executed, will simply transfer control to the method whose address is in the first location of the object that ep points to. In other words, each Exp object, whether it is of concrete type Expi or Exps or Expp (or any additional derived classes of Exp that we may define) will contain, in its first location, the address of the *correct* evalExp() method that applies to *that* particular type of Exp object.

That is how polymorphism is implemented. The use of what are called "v-tables" optimizes the memory usage if the Exp class happened to contain, say, 20 abstract methods instead of just one as in our case. In this case, the simple approach would require each object to contain the address of each of the concrete methods corresponding to the 20 abstract methods of the base class; and depending on how many objects are constructed, this can be a large amount of memory. So what the compiler does is to construct one table corresponding to each concrete class, such as Expp, with this table containing the addresses of the 20 concrete methods defined in that class corresponding to the abstract methods of the base class. When an object of type Expp is constructed, its first location will simply contain the address of the table corresponding

to the Expp class; if an object of type Exps is constructed, its first location will simply contain the address of the table corresponding to the Exps class; etc. But in this case too, each object will carry information with it that points to the concrete methods that apply to objects of that particular concrete type. Then, when compiling calls such as, say, `"ep -> m5()"` where m5 is the name of the fifth abstract method of Exp, the compiler produces code which, when executed, will obtain the address of the correct m5() method from the fifth location in the table pointed to by the first location in the object whose address is in ep. Again there is no conditional code of any kind. (This entire approach is called "runtime dispatching".)

5 & 6. Questions 5 and 6 were Scheme questions and I was glad to see most students got them right. Q 5 asked you to define a function check[s] that, given a non-empty list of numbers s, returns `#t` if the *largest* value in s is exactly twice as large as the *smallest* value in s and `#f` otherwise. This is easily defined as follows:

```
check[s] :: =[largest[s], *[2, smallest[s]]
```

where largest[] is the same function as the maxList[] function we saw in class. And the smallest[] function can be defined in exactly the same way as maxList[].

Q 6 asked you to define a function, difference[s1, s2], that given two lists of atoms, returns a list of those atoms that are in s1 which are *not* in s2.

This may be defined as follows:

```
difference[s1, s2] ::
  [ null?[s2] --> s1; // no elements in s2, so we just return all of s1
  | null?[s1] --> s1; // no elements in s1, so again we just return s1
  | xmemb[car[s1], s2] --> difference[cdr[s1], s2];
            // here car[s1] is in s2, so we forget it and just return
            // difference[cdr[s1], s2]
  | #t --> cons[car[s1], difference[cdr[s1], s2]];
            // but if car[s1] is in not in s2, we have to include it
            // in the result; so we cons it to the rest of the result,
            // i.e., difference[cdr[s1], s2], and return that as result
  ]
```

Hope all that made sense. We will talk about this during our Monday on-line class!

–Neelam