Spring 2013
Programming Languages
Homework 1

- Due Tuesday, February 12, 2013 at 4:59 PM Eastern Standard Time.
- This assignment consists of `flex`/`bison` questions and also "pencil and paper" questions. Submit a separate `flex` and `bison` files for every question. Submit one file for all pencil and paper questions.
- 15 points per day (up to 2 days) will be deducted after the deadline. Any homework received after Thursday, February 14 at 5:59 PM will receive 0. No exceptions.
- No collaboration with other students permitted for answering homework questions. You may collaborate for more general purposes such as understanding grammars, learning how to use `flex` and `bison`, etc.
- There are 100 possible points. See the brackets for the number of points each question is worth.
- Each submission is subject to automatic scanning for plagiarism. **Do not** cheat.

Standard Query Language (SQL) is a formal language for managing data in a relational database system. In this assignment, you will write a scanner and parser for a subset of SQL. We will not concern ourselves with semantics—in other words, we will not be querying any actual databases. Although processing a SQL query is not exactly the same as compiling programming language source code, both situations involve scanning, parsing, and even semantic analysis (which we will not consider here).

The grammar description shown below is typical of what you might see in a standards document or syntax description. This one is a subset of the H2 database system. BNF notation is used below, but there are some differences between what you see below and the syntax we studied in class. In particular,

- `{ XXX }` means that `XXX` should appear exactly once
- `XXX [...]` means: $\texttt{XXX}^+$
- `XXX [,...]` means: $\texttt{XXX[,XXX]}^*$

SQL Scanner [15] The SQL subset for this assignment contains the following keywords/phrases: SELECT, *, DISTINCT, ALL, ANY, SOME, FROM, WHERE, GROUP, BY, ORDER, UNION, MINUS, EXCEPT, INTERSECT, LIMIT, OFFSET, LEFT, RIGHT, OUTER, INNER, CROSS, ON, IN, AS, JOIN, IS, NOT, NULL, EXISTS, DATE, TIME, TRUE, FALSE, ASC, DESC, NULLS, FIRST, LAST, HAVING, NATURAL.

The token **compare** consists of: `<>`, `<=`, `>=`, `=`, `<`, `>`, `!=`.

Additionally, **value** is a kind of token, consisting of the following:

- string : 'anything'
- dollarQuotedString: $$anything$$
- int: -2147483647 to 2147483647
- long: -9223372036854775808 to 9223372036854775807
- decimal: [+|−] number [. number]
- date: DATE 'yyyy-MM-dd'
- time : TIME 'hh:mm:ss'
- boolean: TRUE | FALSE
- array: (expression [,...])
- number : digit[...]
- digit: 0-9

For `int`, `long`, and date expressions above (e.g., 'yyyy') above, it is theoretically possible (but not feasible) to enforce the value ranges using regular expressions alone. (Can you figure out how to do this?) In practice, scanners enforce the general pattern, but not the specific range of values. Typically, enforcement of the specific value ranges shown above would be accomplished by a semantic analysis step. You can assume this to be the case for this assignment.

Using `flex`, write a scanner to accept the tokens above. Submit your solution in a file named `sql.l`.

Tip: Remember that `flex` generates C++ code, creating an identifier with the same name as the token. Because NULL has a special meaning in C++, using a token with that name may cause errors. I recommend naming that token differently.

Tip: Regular expressions are matched in the order that they appear in the `flex` input file. Ensure that the more specific expressions are placed before the more general ones. Otherwise you may see a "rule cannot be matched" error.

SQL Parser [30] The syntax of a SQL `SELECT` statement is given by the description below. Note that all-uppercase words are keywords and the rest of the words are nonterminals:

select =

```
SELECT [ DISTINCT | ALL ] selectExpression [,...]
FROM tableExpression [,...]

[ WHERE expression ]
[ GROUP BY expression [,...] ]

[ HAVING expression ]
[ { UNION [ ALL ] | MINUS | EXCEPT | INTERSECT } ( select ) ]

[ ORDER BY order [,...] ]
[ LIMIT expression [ OFFSET expression ] ]
```

The nonterminals described above are as follows:

- *name* := { { A-Z | a-z | _ } [{ A-Z | a-z | _ | 0-9 } [...]] } | " { A-Z | a-z } [...] "
- *tableExpression* := { [*name* .] *name* | (*select*) } [[AS] *name*] [[{ { LEFT | RIGHT } [OUTER] | [INNER] | CROSS | NATURAL } JOIN *tableExpression* [ON *expression*]] [...]]
- *operand* := *value* | [*name* .] *name*
- *condition* := *operand* [*conditionRightHandSide*]
- *conditionRightHandSide* := IS [NOT] NULL |
  IN ({ *select* | *expression* [,...] }) |
  compare { *operand* | { ALL | ANY | SOME } (*select*) }
- *expression* := [NOT] *condition* | EXISTS (*select*)
- *selectExpression* := * | *expression* [[AS] *name*]
- *order* := { *expression* } [ASC | DESC] [NULLS { FIRST | LAST }]

The start nonterminal is: *query* := *select* ;

Example valid inputs are:

```
SELECT name FROM salesreps WHERE rep_office IN ( 22, 11, 12 );

SELECT DISTINCT name FROM employees WHERE manager IS NULL;

SELECT * FROM prices JOIN regions ON prices.regionid = regions.id
```

```
        JOIN products ON products.prodid = prices.prodid;

SELECT city, target, sales FROM offices WHERE region = 'Eastern'
      ORDER BY city;
```

Using `bison`, write a parser to accept the language described above. Submit your solution in a file named `sql.y`.

Tip: `bison` input is very simple: it does not accept any BNF except for the | symbol. You will need to rewrite the grammar above to implement BNF concepts like [ ], +, and *. Read the `bison` documentation to get some ideas.

Answer the following parsing questions:

1. Recall from our class discussion that an *infinite language* is one containing infinitely many strings (under the theoretical assumption of infinite space). SQL is an infinite language. Using the grammar above, explain why.

2. We see that *select* is enclosed in parentheses wherever it appears on the right hand side of a production. What happens when you remove parentheses from one or more productions in the grammar? Explain what happens and why.

3. Do all of Exercise 2.13 from the course textbook (Scott, 3rd ed.)

Mini-Language Parser [25] Using `flex` and `bison`, write a scanner and grammar to accept the following:

```
myProcedure ( int : n )
begin
  abs = n
  if n < 0 then abs = 0 - abs fi

  sum = 0
  read count
  while count > 0 do
     read n
     sum += n     # equiv: sum = sum + n
     count -= 1   # equiv: count = count - 1
  od
  write sum
end
```

Ensure that the grammar allows for one or more statements in the body of the `if` statement and `while` loop. All 6 standard comparison operators should also be supported with arbitrary expressions as operands.

Turn in `lang.l` and `lang.y` containing your `flex` and `bison` input, respectively.

C++ Standard [10] This question will require you to look at the most recent C++ draft Standard (C++0x) from the course web site.

1. According to the standard, how many *kinds* of tokens are there in C++? What are they?

2. How many keywords does C++ have (including alternate representations)?

3. We've previously seen that BNF may have a different syntactic appearance than what we studied in class. The C++ grammar is no exception. Consider the fragment presented in section 2.13.3 (Floating literals). Translate this fragment to the form of BNF we studied in class.

4. The Gnu Compiler Collection (GCC) is one of the only widely used mainstream compilers with a (handwritten) LL parser. Almost all other major/commercial parsers (like `bison` parsers) are table-driven LR. Take a brief look at the C++ grammar in Annex A (Grammar summary). Can this grammar be used in an LL parser "as is," or will some modification be necessary? Why?

Precedence [5] During our first class, we saw this example of an expression that copies one string to another in the language of C:

```
while (*p++ = *q++) ;
```

In this example, variables `p` and `q` are *character pointers* (i.e., memory addresses that "point" to the location in memory of a character.) One may access the character itself by *dereferencing* the pointer.

According to the grammar for C (and C++), the operators seen in this example have the following *precedence* (listed below from highest to lowest):

1. `++` Postfix increment (applies to scalar variables)
2. `*` Dereference (applies to pointer variables)
3. `=` Assignment

What is the postfix operator incrementing? What is assigned? Explain.

Scopes [5]

1. Under static scoping, what value does the program return?
2. Under dynamic scoping, what value does the program return?

```
 1  const int b=4;
 2  void f()
 3  {
 4   return b + 6;
 5  }
 6
 7  void g()
 8  {
 9   int b = 3;
10   { int b = 1; }
11   return f();
12  }
13
14  int main()
15  {
16   return g();
17  }
```

Short-Circuit Evaluation[10] Consider the following C++ code fragment below.

```
for (int x=0,y=0; x++ < 5 || y++ > 3; )
  std::cout << "Output " << x << " " << y << std::endl;
```

1. Are C++ compilers required to implement short-circuit evaluation, according to C++0x? Note that the operator for logical OR in C++ is `||`.

2. Given the above, what does the code above print? Explain why.

3. Suppose your answer to the first question above was not true. What would it print then? Explain why.

4. Are Java compilers required to implement short-circuit evaluation according to the standard? What does Section 15 of the Java Language Standard (JLS) on the course page say about it? Note that the operator for logical OR in Java is the same as C++.