University of Toronto
CSC324 –Programming Languages, winter 2011

Assignment # 3B

Details
- Topics: Grammar
- Weight: 15%
- Due Date: this assignment is due on April 7th, 2011 @ 11:59pm
- This assignment **must be done in a team of two members** (if you can't find a team member, post a message to the discussion board asking for one**).**

Submission
Place all your files in a folder named after your UTORID and submit that folder zipped through the portal (if your UTORID is abcd, the file should be named abcd.zip). A link for submission of 3B is provided in the assignments folder in the portal (http://portal.utoronto.ca).

Resources
- ANTLRWorks http://www.antlr.org/download.html
  http://www.antlr.org/works/help/index.html
- Java Development Kit (SE)
  http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Blackboard Discussion board http://portal.utoronto.ca

Description
This assignment is to get you trained in writing a valid grammar for a programming language. You are given an informal specification of a language called **uot**, as well as sample programs. Your task is to write a grammar for that language using ANTLRWorks. ANTLRWorks is a visual grammar builder, verifier and also parser generator. It will allow you to automatically generate a uot parser from a valid grammar. After you generate the parser, you are required to test the parser against the given sample code. Note that being able to parse only the files we provide you will not be enough to get full marks, your parsers should be able to parse any string in the uot language.

What to submit?
- uot.g grammar file
- README (***team members names, student ids and UTORIDs)***

Language Description
- A *uot* file consists of two parts: *use statements* and *prototype declarations*
- A use statement is either of the form "***using*** *<directory$_1$>/<directory$_2$>/ … <directory$_n$>;*" or "***using*** *<directory$_1$>/<directory$_2$>/ … <directory$_n$>/*;*"
- A prototype declaration starts with a possibly empty list of *modifiers*, continues with the keyword "**prototype**", a prototype name, and ends with a "*prototype body*"

- A modifier is one of the following words: **visible**, **shielded**, **onlyone**, **constant**, **hidden**.
- A prototype body is enclosed between the keywords "**begin**" and "**end**". Between those two keywords, there are *method declaration*s and *field declaration*s.
- A method declaration starts with modifiers, and continues with a *yield type*, a list of *formals*, and finally a list of *statement*s enclosed between "**begin**" and "**end**" keywords.
- A field declaration starts with modifiers, continues with a *data type*, a field name, and an optional initial value.
- A data type is one of the following: **integer**, **bool**, **char**, **double**.
- A yield type is either "**nothing**" or a data type.
- A statement is one of the following: an *assignment*, a *yield statement*, a *variable declaration*, an *expression*, an *when statement*, an *aslong statement*.
- An assignment has a variable name at the left hand side, and an expression at the right hand side.
- A yield statement consists of the keyword "**yield**" followed by an expression
- A variable declaration is a variable name preceded by a data type, and followed by an optional initial value assignment.
- An expression can consist of variable names, *constant*s, *function call*s with *operator*s in between them.
- A constant is either a integer value, a double value, a quoted string value, keyword "**true**" or keyword "**false**".
- A function call is a function name, followed by a possibly empty list of arguments.
- An operator can be a binary operator or a unary operator.
- An if statement starts with the keyword "**when**" followed by an expression and a statement list enclosed by the keywords "**begin**" and "**end**". The if statement may also have "**orwhen**" statements and "**otherwise** statement"s before the "**end**" keyword.
- An **aslong** statement is similar to an if statement, however doesn't allow an "**else**" statement inside.
- The statements except **when** statements and **aslong** statements end with a **semicolon**.
- User defined names (prototype names, variable names etc.) may include numbers and integers. However they cannot start with a number, or cannot be any of the keywords.
- The operators have different precedence from each other. Following tables summarizes them:

Priority	Operators	Operation
1	+ -	Unary plus, unary minus
2	*	Multiplication
3	+ -	Addition, subtraction
4	lessthan lessorequal greaterorequal greaterthan	Less than, less than or equal to, greater then, greater than or equal to
5	is isnot	Equal to, not equal to
6	and	Boolean and
7	or	Boolean or

You may check the sample code for the finer details.

How to compile/run the Parser?

1. Install Java Development Kit (see resources section above)
2. Open console window (in Windows; Start -> Run)
3. Go to the directory (folder) where the generated parser source file exists
4. Copy the <grammar-name>Lexer.java and <grammar-name>Parser.java files to a directory with the same name as the package name your declared in your ANTLR grammar file (i.e. your `@header {package <package-name>;} @lexer::header {package <package-name>;}` in the beginning of your grammar file)
5. Place the MyParser.java file we provide to you one level above the directory you have just created.
6. To compile the parser (assuming parser file is called uot.java)

```
javac MyParser.java
```

7. To run the parser (assuming parser file is called uot.java, compilation will result in uot.class generated – do not include the .class when running the program)

```
java MyParser sample.uot
```

MyParser.java

```
import java.io.*;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import <package-name>.*;

public class MyParser
{

  public static void main(String[] args) throws IOException
  {

   <grammar-name>Lexer    lexer    =    new    <grammar-name>Lexer(new
ANTLRFileStream(args[0]));

    CommonTokenStream tokens = new CommonTokenStream(lexer);

    <grammar-name>Parser parser = new <grammar-name>Parser(tokens);

    try {
      parser.prog();
    } catch(Exception e) { System.out.println("Error!"); }

    if (parser.getNumberOfSyntaxErrors() == 0)
    {
      System.out.println("Parsing successful");
    }
    else
    {
      System.out.println("Parsing unsuccessful");
    }

  }

}
```

Note: Be sure to replace <package-name> and <grammar-name> with your own package and grammar names.

Sample Code

Valid File #1

```
using x/y/*;

visible prototype foobar
begin

   visible onlyone nothing start()
   begin
      x <- 3;

      aslong(true)
      begin
           y <- +5;

           aslong(x and y or func() greaterthan 5)
           begin
                z <- 6 * 3.E-0;
           end
      end
   end

   shielded onlyone constant integer y <- 4;

end
```

Valid File #2

```
using uot/utilities/*;
using uot;

visible prototype Prototype1
begin
   visible onlyone nothing start()
   begin
      x <- x - 5;
   end
end


hidden prototype Prototype2
begin
   hidden onlyone hidden constant integer func()
   begin
      someValue <- -5;

      x1y1 <- someFunction();
   end


   visible double func2()
   begin
      yield 7 greaterorequal 5;

      double x <- 3.0;

      x <- x + 7.56;
   end
end
```

Valid File #3

```
using just/header;
using but/no/other/definitions/*;
```

Valid File #4

```
shielded constant hidden visible prototype something
begin

     constant shielded visible char something <- 3.33E-33;

     nothing something(integer x, integer x)
     begin
          yield other3things;
     end

     visible onlyone nothing something(integer x, double y, char z,
bool t)
     begin
          x <- 3;

          when ((x is 3) and (y isnot 4))
          begin
               y <- 5;

          orwhen ((x is 3) and (y isnot 4))

               z <- 6;

               bool y <- 3;

          orwhen (((x is (3 and y)) isnot 4))

               yield (((x is 3) and y) isnot 4);

          otherwise

               when (9.99E+99 lessorequal false)
               begin
                    (x + y) * (z + t) and (-6.66E66);
               end
          end

     yield x;
     end

   constant onlyone constant integer y <- 4;
end
```

Valid File #5

```
using use/statement/with/a/directory;
using use/statement/with/everything/inside;


constant onlyone prototype MyFirstPrototype
begin

   onlyone bool true1OrFalse0 <- 0.5;

   constant bool areTheseTrue(bool x, bool y)
   begin
      yield askOtherFunctionWhetherTheseAreTrue(y, x);
   end

   constant nothing duplicateVariables()
   begin
      when (x is x)
      begin
           integer are;

           integer are;

           double allowed;

           double allowed;

           char noone;

           char noone;

           bool cares;

           bool cares;

      end
   end
end
```