Compiler Project
PA1 – Syntactic Analysis

Assigned: Tue Jan 17
Due: Mon Feb 6 (11:59 PM)

The programming project in this class is the construction of a compiler for the miniJava language. The first assignment is to build a scanner and parser for miniJava to recognize syntactically correct programs. Section 1 of this assignment describes the miniJava language syntax and section 2 details the assignment.

1. miniJava

The miniJava language is a subset of Java. Every miniJava program is a legal Java program with Java semantics. Following is an informal summary of the syntactic restrictions of Java that define miniJava. Later assignments will modify restrictions.

A miniJava program is a single file without a package declaration (hence corresponds to an unnamed or anonymous package), and has no imports. It consists of Java classes. Classes are simple; there are no interface classes, subclasses, or nested classes.

The *members* of a class are fields and methods. Member declarations can specify **`public`** or **`private`** access, and can specify **`static`** instantiation. Fields do not have an initializing expression in their declaration. Methods have a parameter list and a body. There are no constructor methods.

The *types* of miniJava are primitive types, class types, and array types. The primitive types are limited to **`int`** and **`Boolean`**. A class type is the name of a declared class. The array types are the integer array **`int`** `[]` and the *class*`[]` array where *class* is a class type.

The *statements* of miniJava are limited to the statement block, declaration statement, assignment statement, method invocation, conditional statement (**`if`**), and the repetitive statement (**`while`**). A declaration of a local variable can only appear as a statement within a statement block and must include an initial value assignment. The **`return`** statement can appear anywhere a statement can appear and has an optional expression for the result to be returned.

The *expressions* of miniJava consist of operations applied to literals and references (including indexed and qualified references), method invocation, and **`new`** arrays and objects. Expressions may be parenthesized to specify evaluation order. The operators in miniJava are limited to

relational operators:	`>`	`<`	`==`	`<=`	`>=`	`!=`
logical operators:	`&&`	`||`	`!`			
arithmetic operators:	`+`	`-`	`*`	`/`		

All operators are infix binary operators *`binop`* with the exception of the unary prefix operators *unop* that consists of logical negation (`!`), and arithmetic negation (`-`). The latter is both a unary and binary operator.

1.1. Lexical rules

The terminals in the miniJava grammar are the tokens produced by the scanner. The token *id* stands for any identifier formed from a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase letters. The token *num* stands for any integer literal that is a sequence of decimal digits. Each operator shown above should have a corresponding token. Classes *binop* and *unop* stand for sets of operator tokens. Token *eot* stands for the end of the input text. The remaining tokens stand for themselves (i.e. for the sequence of characters that are used to spell them). Keywords of the language are shown in bold for readability only; they are written in regular lowercase text.

Whitespace and comments may appear before or after any token. Whitespace is limited to spaces, tabs (`\t`), newlines (`\n`) and carriage returns (`\r`). There are two forms of comments. One starts with `/*` and ends with `*/`, while the other begins with `//` and extends to the end of the line.

The text of miniJava programs is written in ASCII. Any characters other than those that are part of a token, whitespace, or a comment are erroneous.

1.2. Grammar

The miniJava grammar is shown on the next page. Nonterminals are displayed in the normal font and start with a capital letter, while terminals are displayed in `fixed width font`. Terminals *id*, *num*, *unop*, and *binop* and represent a set of possible terminals. The remaining symbols are part of the BNF extensions for grouping, choice, and repetition. Besides these extensions the *option* construct is also used and is defined as follows: $(\alpha)? = (\alpha \mid \varepsilon)$. To help distinguish the parentheses used in grouping from the left and right parentheses used as terminals, the latter are shown in bold. The start symbol of the grammar is "Program".

2. Syntactic analysis assignment

The first task in the compiler project is to create a scanner and parser for `miniJava` starting from the lexical rules and the grammar given in this document. Create a `miniJava` package that includes the `Compiler.java` mainclass and a `SyntacticAnalyzer` subpackage.

Populate the `miniJava.SyntacticAnalyzer` package with the Scanner, Parser, and Token classes. A simple parser and scanner design will be shown in class that can be used as the basis for your miniJava syntactic analyzer. You can also study the Triangle compiler, although many details are different between miniJava and Triangle. You will want to include other classes for reading the sourcefile and keeping track of a token's position in the sourcefile (although not needed at this checkpoint), as well as handling parsing errors. You can study the classes defined in the syntactic analyzer in the Triangle distribution (e.g. `SourceFile`, `SourcePosition`, `SyntaxError`). You will not be building an AST in this assignment, so you need not import `AbstractSyntaxTree` classes in the parser.

The `Compiler.java` class should contain a main method that parses the sourcefile named as the first argument on the command line (the extension may be `.java` or `.mjava`). Execution

must terminate using the method `System.exit(`rc`)` where $rc = 0$ if the input file was successfully parsed, and $rc = 4$ otherwise. A diagnostic message is helpful in case the parse fails.

miniJava Grammar

Program ::= (ClassDeclaration)* *eot*

ClassDeclaration ::= **class** *id* **{** (FieldDeclaration | MethodDeclaration)* **}**

FieldDeclaration ::= Visibility Access Type *id* **;**

MethodDeclaration ::= Visibility Access (Type | **void**) *id* **(** ParameterList? **) {**Statement***}**

Visibility ::= (**public** | **private**)?

Access ::= **static** ?

Type ::= **int** | **boolean** | *id* | (**int** | *id*) **[]**

ParameterList ::= Type *id* (**,** Type *id*)*

ArgumentList ::= Expression (**,** Expression)*

Reference ::= *id* (**[** Expression **]**)? | **this** | Reference **.** *id* (**[** Expression **]**)?

Statement ::=
{ Statement* **}**
| Type *id* **=** Expression **;**
| Reference **=** Expression **;**
| Reference **(** ArgumentList? **) ;**
| **return** Expression? **;**
| **if (** Expression **)** Statement (**else** Statement)?
| **while (** Expression **)** Statement

Expression ::=
Reference
| Reference **(** ArgumentList? **)**
| *unop* Expression
| Expression *binop* Expression
| **(** Expression **)**
| *num* | **true** | **false**
| **new** (*id* **()** | **int [** Expression **]** | *id* **[** Expression **]**)