Compiler Project
PA1 – Syntactic Analysis

Assigned: Tue Jan 13
Due: Mon Feb 2

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`**, and the array types are limited to the integer array **`int`** `[]` and the *class*`[]` array where *class* is any 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, if present at all, can only appear as the last statement in a method and yields a result.

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 operations:	>	<	==	<=	>=	!=
logical operations:	&&	||	!			
arithmetic operations:	+	-	*	/		

All operators are infix binary operators (*`binop`*) with the exception of the unary prefix operators (*unop*) 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. Tokens *`binop`* and *`unop`* stand for the operators listed above, and the 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. 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 `this 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 in this document. Create a `miniJava` directory that holds a `Compiler.java` and a `SyntacticAnalyzer` subdirectory. A simple parser and scanner will be shown in class that can be used as the basis for your minJava syntactic analyzer. You can also study the Triangle compiler, although many details are different between minJava and Triangle.

Populate the `SyntacticAnalyzer` subdirectory with implementations for the Scanner, Parser, and Token classes. 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). Llook at the classes defined in the syntactic analyzer in the Triangle distribution (e.g. `SourceFile`, `SourcePosition`, `SyntaxError`). You will not be building an AST yet, so you need not import `AbstractSyntaxTree` classes in the parser.

The `Compiler.java` in the miniJava directory 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. No diagnostic message is needed in case the parse fails at this point, but it will be needed at later checkpoints. Instructions for submission of PA1 will follow closer to the due date.

miniJava Grammar

Program ::= (ClassDeclaration)* *eot*

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

FieldDeclaration ::= Declarators *id***;**

MethodDeclaration ::=
    Declarators *id* **(**ParameterList? **)** **{** Statement* (**return** Expression **;**)? **}**

Declarators ::= (**public** | **private**)? **static**? (Type | **void**)

Type ::= PrimType | *id* | ArrType

PrimType ::= **int** | **boolean**

ArrType ::= (**int** | *id*) **[]**

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

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

Reference ::= Reference **.** *id* | (**this** | *id*)

IxReference ::= Reference (**[** Expression**]**)?

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

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