Compiler Project
PA1 – Syntactic Analysis

Assigned: Thu Jan 13
Due: Mon Jan 31 (Midnight)

The programming project in this class is the construction of a compiler for *miniJava*, a subset of Java. Every miniJava program is a legal Java program with Java semantics. The miniJava compiler project is built over the course of the semester and is divided into 4 phases (syntactic analysis, abstract syntax tree construction, type checking, and code generation). Each phase will be tested separately. You will receive a report shortly after the due date of each phase, and the tests will be provided as well. A final fifth phase provides an opportunity to make corrections and add optional extensions.

The first assignment, syntactic analysis, is to build a scanner and parser to recognize syntactically correct miniJava programs, and reject syntactically incorrect inputs. The next section describes the miniJava language syntax and section 2 details the assignment.

1. miniJava syntax

Following is an informal summary of the syntactic restrictions to Java that define miniJava. Later assignments will modify some of these restrictions.

A miniJava program is a single ASCII text file. It lacks a package declaration, hence corresponds to an unnamed or anonymous Java package, and has no imports. It consists of Java class declarations. Classes are simple: there are no subclasses, interface classes, 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 a result to be returned.

The *expressions* of miniJava consist of operations applied to literals (e.g numbers) and references (including simple identifiers, qualified references, and indexed references), method invocation, and **`new`** array or object creation. Expressions may be parenthesized to specify evaluation order.

Operators in miniJava are limited to

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

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

hence the (-) operator seems like a place for bugs

1.1. Lexical rules

The text of a miniJava program is partitioned into lexical units (known as tokens). An *identifier* token (*id*) is formed from a sequence of letters, digits, and the underscore character, and must start with a letter. Uppercase letters are distinguished from lowercase letters. The miniJava keywords (**if**, **boolean**, etc.) each have their own eponymous token. Keyword names cannot be used as identifiers. A number token (*num*) is a sequence of decimal digit characters. The operators shown above will each have a corresponding unique token.

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.

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.

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 context-free grammar is shown on the last page. Nonterminals are displayed in the normal font and start with a capital letter, while terminals are tokens and 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 used and is defined as follows: $(\alpha)? \equiv (\alpha \mid \varepsilon)$. 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. PA1 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. For this assignment you will create a `miniJava` package that includes a `Compiler` mainclass and a `SyntacticAnalyzer` subpackage.

Populate the `miniJava.SyntacticAnalyzer` package with Scanner, Parser, and Token classes. A simple scanner and top-down parser 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 details differ between miniJava and Triangle. You will want to include auxiliary classes for reading the sourcefile and keeping track of a token's position in the sourcefile (although not required at the PA1 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 Abstract Syntax Tree (AST) in this assignment, so you need not import `AbstractSyntaxTree` classes in the parser.

The `Compiler.java` mainclass 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 of your compiler 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.

An example scanner and parser for arithmetic expressions using the structure developed in our text are provided on the class web page and is named `simpleScannerParser`.

The miniJava grammar is shown on the next page.

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 | this | Reference . id

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

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