Compiler Project
PA1 – Syntactic Analysis
Due: 1/31/24 11:59:59pm

The programming project in this class is the construction of a compiler for miniJava, a subset of the Java programming language. Every miniJava program is a legal Java program with Java semantics. The miniJava compiler is built over the semester and is divided into 4 phases (Syntactic analysis, AST construction, Contextual analysis, 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 also be provided. A final fifth phase allows one to make corrections and add optional extensions.

Development Environment.
Although any Java IDE is fine, we suggest using Eclipse for Java Developers:
https://www.eclipse.org/downloads/packages/release/2023-12/r/eclipse-ide-java-developers
This will ensure that your submitted project runs appropriately with the autograder on Gradescope. If you are not already enrolled in Gradescope, either search for COMP 520, or use the following Entry Code: PWBK78.

Starter Files.
The starter files for PA1 will be on the course website. You do not have to use the starter files. An example of syntactic analysis of the mini-Triangle programming language is also provided, do not mistake this for the starter files. The mini-Triangle language is very simple and our compiler will be far more robust. This example may help jump-start your progression in the syntactic evaluation of miniJava.

Project Overview.
PA1 will focus on syntactic analysis. This assignment will take a filename as command line input and output whether any syntax errors were encountered. Note that a syntactically correct program does not necessarily mean a valid program. We are purely focused on syntax analysis and little else in this assignment. For example:

```
class TestA {
    public static void fun1() {
        x = 1 + 3;
        int x = 0;
    }
}
```

```
class TestB {
    public static void fun2() {
        int x = 0;
        x = 1 + 3;
    }
}
```

`TestA` follows the proper miniJava syntax, but *contextually* is incorrect as "x" has not yet been defined. In this assignment, `TestA` will **pass** as the **syntax** is correct.

Project Details.
All of the compiler will reside in the miniJava package. Ensure the Compiler class resides at the root of the miniJava package. The Compiler class must contain the main function for the autograder to properly function. Other than that requirement, you are free to organize your compiler project as you see fit.

1. Create the miniJava and miniJava.SyntacticAnalyzer package.

The starter files have a package called miniJava.SyntacticAnalyzer, and most of the PA1 code will reside here. So long as Compiler.java exists in the miniJava package, the autograder should be able to grade the submission. Lastly, the autograder supports Java 11, so please do not use any Java features from newer runtime environments.

MiniJava differs from regular Java in that the compiler will initially only support compiling one java source file. It lacks a package declaration and thus corresponds to an unnamed or anonymous Java package and has no imports. The source file consists of Java class declarations, and the classes are simple. There are no subclasses, interface classes, or nested classes. However, because of how the compiler is organized, some missing features can be easily added in PA5 for extra credit, while other features would not be so trivial.

2. Create a TokenType enumeration and a Token class.

A Token consists of some underlying text and a type classification. For example, the "Class" token would have the "Class" TokenType, and "class" as the text. Similarly, ">=" could be the "RelationalOperator" TokenType, and contain the text ">=", but should likely just be a generic "Operator" TokenType.

The first step towards interpreting code in a compiler involves setting up a Scanner. The Scanner object will take an established input stream and *tokenize* the source code. This tokenization process is known as Lexical Analysis. The Scanner is also responsible for removing whitespace such that the output from the scanner is a series of tokens. For example, `TestA` on the previous page would result in the following token types:

```
Class, Identifier, LCurly, Public, Static, Void, Identifier, LParen, RParen,
LCurly, Identifier, Equals, IntLiteral, Operator, IntLiteral, Semicolon,
Int, Identifier, Equals, IntLiteral, Semicolon, RCurly, RCurly, null.
```

You do not have to tokenize precisely as stated above, however, keep in mind that you should be able to differentiate between different token types quickly. This will ensure the next section of PA1 (the Parser) will be simpler to implement. The full list of token types is not given, and I encourage you to seek out Java grammar guidelines to better determine how you should organize your tokens. This may also become clearer when Tokens are used in the Parser, so continue with the project even if you are currently unsure of how to categorize your tokens in the TokenType enumeration.

3. Create the Scanner object.

The scanner object removes whitespace and turns the source code into a stream of tokens. Consider what should happen when a comment (//) or block comment (/* */) is encountered. Much like whitespace, any text inside a comment should not be tokenized and completely ignored in the stream of tokens.

The Scanner is responsible for providing one crucial function, `scan`. This function will find the next Token in the source code file and return it. Subsequent calls to `scan` will return subsequent Tokens until no more Tokens remain, at which point `scan` should return null.

From the Triangle compiler example, we read a single letter to determine what type of Token it may be. For example, if the current letter is numeric, then we know this is the start of an integer literal. Accept the first letter (function is called `takeIt`), and continue accepting letters until the letters are not numeric. All accumulated letters belong to one integer literal and this can be packaged into a Token object and returned.

Lexical Rules.
The lexical unit is the Token. 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. As described earlier, the number token `num` (also called Integer Literal) is a sequence of decimal digit characters.

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 that starts with /* and ends with */, while the other begins with // and extends to the end of the line.

4. Create the Parser object.

The parser object parses the token stream coming from the Scanner. Before we begin parsing tokens, we need to discuss grammar.

Grammar.
A terminal symbol is a symbol that does not contain itself or other symbols. Non-terminal symbols are symbols that are composed of other symbols (possibly including other non-terminals). On the last page, non-terminals are displayed in normal font and start with a capital letter, while terminals are tokens and are displayed in `fixed-width` font. Terminals like `id`, `num`, `unop`, and `binop`, are a set. The miniJava grammar is described using a CFG-like syntax (COMP 455).

An option is denoted: $\alpha \equiv (s|t)$ where the symbol α is composed of either s or t. An optional symbol is given the question-mark '?' symbol after it. An asterisk indicates the symbol may be repeated zero or more times. For example, there can be multiple class declarations, so the root symbol "Program" may contain multiple ClassDeclaration symbols.

The start symbol is Program, meaning your Parser should start by reading ClassDeclaration symbols until the end of the file is reached.

There are two approaches towards implementing your Parser. Some prefer a top-down approach to follow the Grammar as it becomes more specific, while others prefer a bottom-up approach where they create the methods to parse identifiers, operators, and other terminals, then work towards more difficult parse methods that utilize already complete parse methods. Either approach is acceptable, so long as errors are detected.

Error Detection.

Consider the following examples of a variable declaration statement:

`int x = 1 + 3;`	`int x is 1 + 3;`

Grammar: Type `id` = Expression ;

The two differ at one key location in the stream of tokens:

`IntToken Identifier EqualsToken IntLiteral Operator IntLiteral Semicolon.`

The second example would forego the `EqualsToken` and instead have an `Identifier` at that location. When parsing the Statement symbol, if the first token is a "Variable Type" such as an `int`, then it would be safe to assume that an identifier comes next, then an equal symbol for initial value assignment, and finally an expression with a semicolon. However, in the right-hand example, the stream of symbols does not match the expected grammar rule. As such, an error should be thrown.

Note there is no elegant way to recover from errors. The most important detail is to detect the first error when present. When the Statement from the right-hand example fails, the next Statement will start with an integer literal, which is not a statement, and thus more errors will be added to the list. Once again, you are only graded on the first error being detected, and how you choose to report subsequent errors is up to you.

5. Finish the Compiler object.

The compiler object's main function takes in an argument list. The input file location is passed via the argument list in the main function. Open a FileInputStream to that file location and detect whether the file exists or not. If it exists, create the Scanner and Parser objects, then check if the parser recorded any errors. If errors exist, first output "Error" on its own line then report errors. If no errors exist, output "Success".

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])

Relational Operators: >, <, ==, <=, >=, !=
Logical Operators: &&, ||, !
Arithmetic Operators: +, -, *, /
All operators are binary operators (`binop`) except logical negation(!), and arithmetic negation(-). The latter is both a unary and binary operator.

Addendum: Lexical Analysis Clarifications

There is no need to overcomplicate your Scanner. From the lectures, we see there is value in making Lexical Analysis as simple as possible. By studying the miniJava grammar, we can infer two major simplifications. (1) The integer literal (`num`) can be scanned by ensuring the current letter is numeric, and so long as the current letter is numeric, continue taking the input. When the letter is non-numeric, you can package the previously accepted letters into a Token and return it. There is no need to check to make sure the current letter is whitespace or in { } , ( ) [ ] ; eof * + - / == or any other way to terminate an integer literal. (2) The identifier can be treated similarly, if the FIRST letter is alphabetic (A-Za-z), then take it and continue scanning so long as the letter is alphanumeric or an underscore.

When should something be a Lexical error vs a Syntactic error? That depends on the programming language specification, but in general, so long as the error is detected, that is the important part. Consider the "&" symbol. There is no operator "&", but there is an operator "&&". If we were to have a parseOperator method and accept one "&" and then another "&", then that would imply we also accept "& &" as an operator (note the whitespace), when it is not an operator. As such, some prediction in the Lexer is unavoidable, first accept a single "&", and require that another "&" comes after, otherwise there is no way to package a single "&" as that is not an operator, and thus a lexical error. However, this prediction is not related to syntax, this is instead making sure operators are packaged into Tokens appropriately.

Also consider "54353asdf", while it might look similar to the above example, it is easier to package this as an intliteral and then an identifier. By analyzing the miniJava grammar, we know that an identifier token never comes after an int literal, and so we can shift this processing to the Parser and keep our Scanner simple without having to have any additional checks in the Parser. In other languages, this may not be possible. Consider "0ull" as a valid "unsigned long long" literal in C/C++. Because "0 ull" is not valid, the entire "0ull" sequence needs to be grabbed as a single token in the Lexer.

As such, it highly depends on the language being implemented. For miniJava, we suggest keeping it simple and using the property that the Scanner can be constructed in a simple manner. It is always good to avoid doing syntax checking in your Lexer, so in the case of symbols (>=,<=,==,!=,etc.) that require no whitespace, your Lexer will have to ensure the proper letters are received. In the case of making sure that after an int literal must come a +,-,*,/,;,etc., that is best left to the Parser as that involves syntax.