CS 1622 – Project 3b: MiniJava Errors & IR
Assignment Due: Sunday, April 8, 2018, by 11:59pm

Description
In this part of the project, we will add error handling to the parser and create a semantic analyzer. At the end of this phase, we will generate quadruples of three-address code as our intermediate representation.

Though it is described last, the IR generation is the most important part of this project as it will allow you to continue with your code base in part 3c. Make sure that you generate at least some IR.

Error Handling in the Parser
JavaCUP supports a special "error" token that a production can go to. For example, we may have an error in a statement and thus we would add a production such as:

```
statement ::=   … /* Other productions */
            |
            error SEMICOLON
            ;
```

The semicolon (and additional tokens afterward) is a point of synchronization that gets the parser back on track by discarding erroneous input off of the stack.

Parser Error Handling Requirements
For your project, handle the following four error situations:

1. Statements synchronized by a semicolon
2. Variable declarations synchronized by a semicolon
3. Class and method bodies synchronized by a }
4. Formal and actual parameter lists synchronized by a)

For each parser error (regardless of using the error symbol), report the line and character position that the error occurred on:

```
Parse error at line %d, column %d
```

Name Analysis
With the AST that the first part of the compiler generated, we can now do the name and type analysis for MiniJava.

Name analysis discovers identifier names that are improperly defined.

We will handle two types of errors in this step:

1. Redefinition of an identifier in a given scope
2. Use of an undefined identifier

To do name analysis and later parts of the project, you will need to construct a symbol table. You should use either the tree of hashtables or the stack of hashtables to represent your nested scopes.

As you build the symbol table, or in a second pass, you may encounter the errors listed above.

Name Analysis Requirements

If you discover more than one declaration of an identifier in a scope, output the message:

```
Multiply defined identifier %s at line %d, character %d
```

If you discover that an undefined identifier is used, output the message:

```
Use of undefined identifier %s at line %d, character %d
```

Type Checking

With AST and Symbol Table in hand, we next need to do type checking.

We will identify the following type errors:

1. Assignment to a class or method name
2. Assignment from a class or method name
3. Attempting to call a class or method name
4. Calling a method with the wrong number and/or types of arguments
5. Applying an operator to the wrong type
6. Using a non-boolean expression in an if or while condition
7. Type mismatch during assignment

Type Checking Requirements

If there is an attempt to assign to a class or method name, or the keyword `this`, output the message:

```
Invalid l-value, %s is a %s, at line %d, character %d
```

If there is an attempt to assign from a class or method name, output the message:

```
Invalid r-value: %s is a %s, at line %d, character %d
```

If an operator is applied to a class or method name, output the message:

```
Invalid operands for %s operator, at line %d, character %d
```

If an attempt to call something that isn't a method (class, variable, etc.) is made, output:

```
Attempt to call a non-method at line %d, character %d
```

If a method is called with the wrong number of arguments:

```
Call of method %s does not match its declared number of arguments at line
%d, character %d
```

If a method is called with the wrong type of argument:

```
Call of method %s does not match its declared signature at line %d,
character %d
```

If you attempt to use an arithmetic or comparison operator (<, +, -, *) with non-integer operands, output:

```
Non-integer operand for operator %c at line %d, character %d
```

If you attempt to use a Boolean operator with non-boolean types (&&), output:

```
Attempt to use boolean operator %s on non-boolean operands at line %d,
character %d
```

If you attempt to use .length on any type other than int [], output:

```
Length property only applies to arrays, line %d, character %d
```

If the condition of an if or while statement does not evaluate to a boolean, output:

```
Non-boolean expression used as the condition of %s statement at line %d,
character %d
```

If the right hand side of an assignment does not match the type of the left hand side, output:

```
Type mismatch during assignment at line %d, character %d
```

If you try to use `this` in a static method (main), output:

```
Illegal use of keyword 'this' in static method at line %d, character %d
```

Note:

- For each error, the character output should be the index of the first letter of the offending code
- System.out.println is declared as taking a single int parameter

IR Generation

The last part of this phase will be to generate IR code. We will generate quadruples of three-address code.

These are the IR statements to use:

IR Operation	Three-address code
Assignment	`x := y op z`
Unary Assignment	`x := op y`
Copy	`x := y`
Unconditional Jump	`goto LABEL`
Conditional Jump	`iffalse x goto LABEL`
Parameter	`param x`
Call	`x := call f, NUMPARAMS`
Return	`return y`
Indexed assignment	`x := y[i]`
	`y[i] := x`
New	`x := new TYPE`
New Array	`x := new TYPE, SIZE`
Length	`x := length y`

You will likely want the operands as pointers to entries in the symbol table. The keyword “this” is a local variable implicitly defined as the first parameter in each non-static method. When calling a non-static method, pass the left hand side of the dot operator as the value of parameter this.

For example:

```
c.f(a)
```

should be translated into:

```
param c
param a
call f, 2
```

Treat true and false as constants of your choosing.

IR Requirements

Generate IR on a method-by-method basis. At the end of this phase, print out your IR with a toString() method using the above style, one IR statement per line. Make sure that the main() method is the first one you output.

Submission

By the deadline, you need to submit:

1. Your JFlex file containing your lexer
2. Your JavaCUP file containing your parser
3. Your java files containing main() and any auxiliary Java files you have used
4. A Makefile to build it all
5. A README text file describing how to run it
6. Any examples that you have tested your program on

Create a zip file of the above files. For teams, submit one file named with both of your usernames. Use SFTP to copy your zipfile to unixs.cis.pitt.edu and then copy your file to:

~jrmst106/submit/1622