Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Compiler Engineering

Assignment 1 — Parsing and Interpreting

Due: Tuesday 20th July @ 23:59

The WHILE Language Compiler

This assignment will extend a compiler for the WHILE language which we have been studying in lectures. You can download the latest version of the compiler from the SWEN430 homepage, along with the supplementary code associated with this assignment.

The WHILE language compiler includes (amongst other things) a *parser*, *type checker* and an *interpreter*. This project is primarily concerned with the *parsing* and *interpreting* components. Specifically, you are tasked with extending the WHILE compiler in a number of ways. Here's an example program in the WHILE language:

```
1  int[] reverse(int[] ls) {
2      int i = |ls|;
3      int j = 0;
4      int[] rs = ls;
5
6      while i > 0 {
7          i = i - 1;
8          rs[i] = ls[j];
9          j = j + 1;
10     }
11
12     return rs;
13 }
```

The WHILE Language Specification provides a complete reference for the syntax of the language. In addition, over one hundred `JUnit` tests are also provided to help you determine whether the compiler is working correctly or not.

NOTE: In completing this assignment you will only need to update three parts of the compiler: the *lexer*, *parser*, and *interpreter*. That is, you **do not** need to update other components such as the *type checker*, etc.

Part 1 — Operator Precedence (20%)

An unusual feature of WHILE is the relatively limited support for *operator precedence*. For example, an expression such as `2 * x + y` is not permitted and, instead, braces must be provided to disambiguate operators (e.g. `(2 * x) + y`). In this part of the assignment, you will update the WHILE compiler to follow other languages (e.g. Java) in accepting expressions such as `2 * x + y`.

Example. The following table illustrates a range of expressions which should be accepted by the compiler, and what they are equivalent to.

Expression	Equivalent
`x + y - z`	`(x + y) - z`
`x - y + z`	`(x - y) + z`
`x + y * 2`	`x + (y * 2)`
`x + y / 2`	`x + (y / 2)`
`x * y / z`	`(x * y) / z)`
`x % y / z`	`(x % y) / z)`
`x && y || z`	`(x && y) || z)`
`x || y && z`	`(x || y) && z)`

HINT: To get complete this part of the assignment, you will need to modify the file `Parser.java`. In particular, those components responsible for parsing logical connectives, comparators and arithmetic operators.

Part 2 — Real Values (30%)

The second task is to extend the WHILE language with the `real` primitive type using 64bit floating point values according to IEEE 754. In particular, they correspond to the `double` type in Java. The syntactic extensions for this primitive type are given as follows:

```
RealLiteral ::= IntLiteral . IntLiteral
```

```
RealType ::= real
```

Real values are expressed as a sequence of two or more numerical digits separated with a point (e.g. `123.456`, etc). Values of `real` type support equality comparators, relational comparators, additive, multiplicative and negation operations.

Example. The following illustrates the `real` datatype:

```
real max(real x, real y) {
  if(x >= y) {
    return x;
  } else {
    return y;
  }
}

void main() {
  assert max(1.234,2.22) == 2.22;
}
```

Notes. Like Java, the WHILE language supports *implicit coercion* from `int` values to `real` values and implementing this correctly requires some care. For example, one can assign an `int` value to a variable of `real` type. Likewise, one can compare an `int` value with a `real` value for equality, etc.

Part 3 — ForEach Loops (20%)

The third task is to extend the WHILE language with support for for-each loops, roughly similar to hose found in Java. As in Java, a for-each loop allows one to easily iterate over the contents of an array. The syntax for the for-each loop is given as follows:

```
ForEachStmt ::= for ( Type Ident : Expr ) { Stmt* }
```

Example. The following illustrates a do-while statement:

```
int sum(int[] items) {
    int sum = 0;
    for(int item : items) {
        sum = sum + item;
    }
    return sum;
}
```

Here, we see a simple for-each loop which iterates over the elements of an array summing them.

Notes. The for-each loop supports `continue` and `break` statements in the expected manner.

Part 4 — Try-Catch Blocks (30%)

The final, and perhaps most challenging, component of this assignment is to extend the WHILE language with try-catch blocks, roughly similar to those found in Java. By now, you should be quite familiar with the internals of the WHILE compiler and prepared for this challenging task. The syntax for try-catch blocks is given as follows:

```
TryCatchStmt ::= try { Stmt* } ( catch ( Type Ident ) { Stmt* } )+

   ThrowStmt ::= throw Expr ;
```

A key observation is that, unlike Java, we do not support `finally` blocks. Likewise, the result of an arbitrary expression can be thrown as there is no specific type of exception, as found in Java.

Example. The following illustrates a try-catch block:

```
1  void main() {
2    try {
3      throw 1;
4    } catch(int x) {
5      assert x == 1;
6    }
7  }
```

Here, we see the value `1` is throw within the `try` block and then immediately caught by the enclosing `catch` handler.

Notes. The handling of definite assignment and unreachable code is important in the context of both `throw` and try-catch statements. Furthermore, one can leverage Java's existing mechanism for exceptions to implement try-catch blocks within the interpreter. Finally, you will observe from the test cases given, that exceptions are automatically thrown when a fault occurs in WHILE. For example, attempting to divide-by-zero generates an exception which can, as in Java, be caught.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable*. See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code*.

3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code*.

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information*.

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be awarded based on the number of passing tests, as determined by the *automated marking system*. **NOTE:** the test cases used for marking will differ from those in the supplementary code provided for this assignment. In particular, they may constitute a more comprehensive set of test cases than given in the supplementary code.