Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Compiler Engineering (2018)

Assignment 1 — Parsing and Interpreting

Due: midnight, 12 August

Overview

This assignment will extend the parser and interpreter for the WHILE language which we have been studying in lectures.

Some of the features you are asked to add are described in the While Language Specification document (WLS), but not all of them. You are provided with around 100 `JUnit` tests cases to illustrate what is expected and help you determine whether the compiler is working correctly or not. If you are not sure what is expected, please ask for clarification.

The While Language Compiler

You can download an initial version of the compiler and the WHILE Language Specification from the SWEN430 homepage here:

`http://ecs.victoria.ac.nz/Courses/SWEN430_2018T2/Assignments/`

To help get you started with the system, let's consider compiling the following file:

```
void main() {
  print "Hello World!";
}
```

If this file is stored in, say `test.while`, then you should be able to execute it using the built-in interpreter as follows:

```
% java -jar lib/whilelang.jar test.while
Hello World!
```

As you can see, the output from the program is printed directly to the console. Furthermore, a `build.xml` file is provided to rebuild the `whilelang.jar` file after you have modified its source.

Passing the command-line switch `-verbose` will produce more verbose output which is useful for debugging. For example: `java -jar lib/whilelang.jar -verbose test.while`.

You can run the `JUnit` tests from within Eclipse by selecting "Run as JUnit Test" on the class `whilelang/testing/tests/RuntimeValidTests`.

NOTE: The given unit tests do not cover all aspects of the language, and you should supplement these with your own tests.

NOTE: There may be some mistakes in the test themselves — if you find any please let me know.

The changes you afre required to make are described in the following sections. The changes are roughly in order of increasing complexity, but you don't need to do them in the order listed.

1 Comments (10%)

Currently, the WHILE compiler does not support comments in the source code (see WLS, 2.2). You should find that at least two test cases (`SingleLineComment_Valid_1` and `MultiLineComment_Valid_1`) fail because of this. The objective here is to modify the compiler so that comments work as expected.

You might like to consider what the impact would be on the implementation if block comments (`/* ... */`) were allowed to be nested.

2 Constant Declarations (10%)

The WHILE language is intended to support *constant declarations*, as follows:

```
const PI is 3.1415
int main() { print PI; }
```

When executed, this program should print "`3.1415`", however at the moment you should find it does not, and that tests `Const_Valid_1` ... `Const_Valid_4` fail. The objective here is to extend the `Interpreter` such that `const` declarations are appropriately implemented.

HINT: The simplest way to do this is to build a `Map<String,Expr>` from constant names to their bodies. Then, when an unknown `Expr.Variable` is encountered, you can check to see whether it's actually referring to a `const` declaration (or not) — but you need to think about where that test is done.

An interesting extension here would be to allow the right hand side to be an expression containing only constants (including named constants, e.g. `2*PI`), rather than just constant values.

3 Switch Statements (20%)

Currently, `switch` statements are not implemented in the compiler (see WSL, 5.3.3), and you should find that the tests `Switch_Valid_1` ... `Switch_Valid_8` fail. The objective here is to implement switch statements appropriately. This will require modifying both the `Parser` and `Interpreter` classes. The syntax for `switch` statements is identical to that found in Java.

HINT: The `default` case can appear at any position in the `switch` statement, but is only taken if none of the other cases match. Furthermore, there may be at most one `default` case.

HINT: Java `switch` statements permit *unevaluated constant expressions* (see JLS§15.28). For example, one can write `case 1+1:` instead of `case 2:` and the Java compiler will evaluate this at compile time. **For simplicity, you may assume case conditions are either constants or named constants (i.e. from `const` declarations)**.

4 String Assignments (20%)

Currently, the WHILE compiler does not support assignment to string elements. For example, the following code fails with a run time error:

```
void main() {
  string s = "Hello";
  s[0] = 'h';
  print s;
}
```

Similarly, the test `Char_Valid_3` fails because it assigns to a string element. The problem manifests itself in `Interpreter.execute(Stmt.Assign, HashMap<String,Object>)`, and is surprisingly tricky to resolve. Superficially, it appears as though the method simply doesn't consider the case of assignment to a string element (which it doesn't). Unfortunately, the problem is more involved because `String`s in Java, unlike e.g. `ArrayList`s, are *immutable*. In contrast, `string`s in WHILE are *mutable*.

There are many ways you can go about resolving this issue. Perhaps the simplest is to move away from representing WHILE `string`s as Java `String`s, but instead as e.g. `char[]` or similar. No matter how you address it, this will likely require widespread changes throughout the `Interpreter` to implement. Care should be taken to ensure that existing test cases still pass after you have fixed this problem.

5 Type Tests (20%)

Run time type tests are used to determine the type of a value at run time. The following illustrates a simple example:ax

```
bool isInt(int|null x) {
  if(x is int) { return true; }
  else { return false; }
}
```

Here, the `is` operator can be thought of as roughly equivalent to Java's `instanceof` operator. The WHILE language compiler does not currently implement this operator, and you should find that a number of test cases (`TypeEquals_Valid_1` ... `TypeEquals_Valid_16`) fail because of this. The objective is to implement the type test operator such that these tests will pass.

6 Explicit Casts (20%)

The WHILE language is intended to support *explicit coercions* (a.k.a *casts*), as follows:

```
int main() {
  int x = 1;
  real y = (real) x;
  print y;
}
```

Executing this method should print "`1.0`"; however, currently, you should find that it prints "`1`", which indicates that the coercion from an `int` to a `real` did not take place. Currently, the tests `Cast_Valid_1` ... `Cast_Valid_4` will fail.

The objective here is to implement **primitive casts**. Primitive casts can be used to convert an `int` into a `real`, as illustrated above, and can also be applied to structured types with `int`s as components. For example, we can cast from `[int]` to `[real]` by performing the primitive cast on all elements. Likewise, `{int x}` can be cast to `{real x}`. Note that such casts require an explicit coercion at run time.

Submission

Your program code 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 (see the export-to-jar tutorial linked from the course homepage).
2. The names of all classes, methods and packages remain unchanged.
3. You have removed any debugging code that produces output, or otherwise affects the computation.

Note: Failure to meet these requirements could result in you getting zero marks for the assignment.