Victoria University of Wellington
School of Engineering and Computer Science

# SWEN430: Compiler Engineering

# Assignment 1 — Parsing and Interpreting

**Due: Tuesday 22nd March @ 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.

This project is primarily concerned with the *parsing* and *interpreting* components of the WHILE compiler. Specifically, you are tasked with extending the WHILE compiler in a number of ways. 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 e.g. `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.

**NOTE:** 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
```

Over one hundred `JUnit` tests are also provided to help you determine whether the compiler is working correctly or not. You can run these tests from within Eclipse by selecting "Run as JUnit Test" on the classes `RuntimeValidTests`, `TypeCheckingTests`, `DefiniteAssignmentTests` and `UnreachableCodeTests`.

## Part 1 — Do-While Loops (40%)

The first task is to extend the WHILE language with support for do-while loops, as this is currently lacking. As in languages like Java or C, a do-while statement executes a statement block at least once, and then repeatedly whilst a given boolean expression (the condition) evaluates to `true`. The syntax for do-while loops is given as follows:

```
DoWhileStmt ::= do { Stmt* } while ( Expr ) ;
```

**Example.** The following illustrates a do-while statement:

```
1  bool allPositive(int[] items) {
2    int i = 0;
3
4    do {
5      if(items[i] < 0) {
6        return false;
7      }
8      i = i + 1;
9    } while(i < |items|);
10
11   return true;
12 }
```

Here, we see a simple do-while statement which checks whether the elements of a given array are all positive or not. Note this program assumes the array has at least one element on entry!

**What to do.** In this part of the assignment, you will need to modify at least the following classes in the While compiler:

```
whilelang.compiler.Lexer
whilelang.compiler.Parser
whilelang.compiler.TypeChecker
whilelang.compiler.DefiniteAssignment
whilelang.compiler.UnreachableCode
whilelang.util.Interpreter
```

You should use the current implementation given for while loops as the basis by which to extend the compiler with do-while loops. In many cases, the code is very similar or identical as for while loops. For example, the condition for a do-while loop should produce a value of **`bool`** type. However, unlike for While loops, variables definitely assigned in the body of a do-while loop can be considered definitely assigned after the loop (i.e. because the first iteration always executes).

The supplementary code provided with this assignment includes a number of test cases which should pass once this extension to the WHILE compiler is working correctly. *However, these tests should not be considered comprehensive and, to do well in this assignment, you will need to write your own test cases.*

## Part 2 — Macros (60%)

Many programming languages (e.g. C, Lisp and Rust) employ *macros* as a simple mechanism for reusing code. A macro is a simple mechanism for implementing textual search-and-replace within a program. That is, the macro term is searched for within the program source and replaced with its body through a process known as *macro expansion*. The following example illustrates the proposed syntax for WHILE:

```
1 macro add(x) is x+1
2
3 int inc(int y) {
4    return add(y);
5 }
```

Here, a simple macro is defined which accepts one argument. This macro is used within the body of the `inc()` method. After macro expansion, the above program would look like this:

```
1 int f(int y) {
2    return y+1;
3 }
```

Here, we can see that the use of `add()` within `inc()` has been replaced with the macro's body, with the argument substituted appropriately. You can find more information about macros here:

- `https://en.wikipedia.org/wiki/Macro_%28computer_science%29`
- `http://en.wikipedia.org/wiki/Hygienic_macro`

**Syntactic Macros.** The goal is to implement a simple macro system for the WHILE language. Our macros will be *syntactic*, meaning that they work at the level of the abstract syntax tree, rather than on the token stream produced by the lexer or directly on the source text itself. This means they always produce code which corresponds to a valid abstract syntax tree (though not always code which will e.g. type check). For example, consider this macro:

```
1 macro add(x,y) is x+y
```

The question is, how should the expression `add(1,2)*3` be expanded? A system based purely on textual substitution will give `1+2*3` which evaluates to `7`, since `2*3` binds more tightly. In contrast, a syntactic system such as that considered here will (effectively) give `(1+2)*3` which evaluates to `9`. The syntax for our macros is given as follows:

```
      MacroDecl ::= macro Ident ( MacroParameters ) is Expr

MacroParameters ::= [ Ident ( , Ident )* ]
```

**What to do.** This part of the assignment is more challenging than the first, and will require a more systematic approach. Therefore, we suggest you complete the assignment in the following steps:

1. **Modify Abstract Syntax Tree (AST)**. To do this, you will need to extend `while.ast.WhileFile` with an appropriate class for describing a macro.

2. **Modify Parser**. To do this, you will need to extend `while.ast.Parser` appropriately to recognise macro declarations and instantiate appropriate AST nodes. At this point, your compiler should be able to correctly parse macros though it will not yet expand them. This means that programs using macros will still fail in some subsequent stage (e.g. type checking).

3. **Add Macro Expansion**. You will need an appropriate compiler stage (e.g. `MacroExpansion`) which is responsible for performing macro expansion. This should be enabled in `WhileCompiler` and should run before type checking occurs.

The supplementary code provided with this assignment includes a number of test cases which should pass once this extension to the WHILE compiler is working correctly. *However, these tests should not be considered comprehensive and, to do well in this assignment, you will need to write your own test cases.*

## Marking

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.

## 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.