University of Glasgow | School of Computing Science

# Assessed Coursework

| | | | | | |
|---|---|---|---|---|---|
| **Course Name** | **Programming Languages H** | | | | |
| **Coursework Number** | **1** | | | | |
| **Deadline** | **Time:** | **17.30** | **Date:** | **08/11/2016 and 29/11/2016 (Glasgow)**<br><br>**11/11/2016 and 02/12/2016 (Singapore)** | |
| **% Contribution to final course mark** | **20%** | | **This should take this many hours:** | **15** | |
| **Solo or Group** ✓ | **Solo** | ✓ | **Group** | | |
| **Submission Instructions** | **Submit through Moodle. See detailed instructions in the assignment document.** | | | | |
| **Who Will Mark This?** ✓ | **Lecturer** ✓ | | **Tutor** | **Other** | |
| **Feedback Type?** ✓ | **Written** | | **Oral** | **Both** ✓ | |
| **Individual or Generic?** ✓ | **Generic** | | **Individual** | **Both** ✓ | |
| **Other Feedback Notes** | **Individual written feedback will be provided for both stages of the assignment. Generic oral feedback will be provided in class for the first stage of the assignment.** | | | | |
| **Discussion in Class?** ✓ | **Yes** ✓ | | **No** | | |
| **Please Note: This Coursework cannot be Re-Done** | | | | | |

## Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below. The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

**Penalty for non-adherence to Submission Instructions is 2 bands**

**You must complete an "Own Work" form via**
**https://webapps.dcs.gla.ac.uk/ETHICS for all coursework**
**UNLESS submitted via Moodle**

| Marking Criteria |
| --- |
| **The marking scheme is described in the assignment document.** |

# Programming Languages H

# Coursework Assignment (2016-17)

In this assignment you will extend the Fun compiler, using the compiler generation tool ANTLR. The Fun compiler is outlined in the course notes. You will start your assignment by familiarizing yourself with ANTLR and the Fun compiler.

The assignment itself consists of three stages: syntactic analysis, contextual analysis, and code generation. The deadlines are slightly different for Glasgow and Singapore.

Glasgow: **08/11/2016** (stage 1) and **29/11/2016** (stages 2 and 3).

Singapore: **11/11/2016** (stage 1) and **02/12/2016** (stages 2 and 3).

The assignment contributes **20%** of the assessment for the PL(H) course.

## Familiarization with ANTLR

ANTLR runs on Linux machines. To use ANTLR, ensure that your `CLASSPATH` includes "`.`" and `/usr/local/antlr/antlr-4.5.1-complete.jar`.

Go to the PL(H) Moodle page, download `Calc.zip`, and extract the files (`Calc.g`, `CalcRun.java`, `ExecVisitor.java`, and some test files) into a new directory named `Calc`.

Study `Calc.g4`. It contains the grammar of Calc, expressed in ANTLR notation.

Study `ExecVisitor.java`. The methods are used to define a depth-first, left-to-right traversal of the syntax tree of a Calc program. Each method defines what will be done at one kind of node, and returns the value of the corresponding expression (or returns 0 if the node is a command).

To make ANTLR generate a lexer and parser for Calc, enter the following Linux command:

```
…$ java org.antlr.v4.Tool –no-listener –visitor Calc.g4
```

This should generate files named `CalcLexer.java` and `CalcParser.java`. You might wish to look at them briefly (although they are not intended for human readers!). In particular, the `CalcParser` class contains methods `prog()`, `com()`, `expr()`, etc., which work together as a modified form of recursive-descent parser.

It is convenient to define a script file containing the ANTLR command. Put the following line into a file called `antlr4` and put the file in a directory that is on your `PATH`.

```
java org.antlr.v4.Tool $*
```

ANTLR also generates `CalcVisitor.java` and `CalcBaseVisitor.java`. `CalcVisitor` is an interface that specifies the necessary methods for traversing the syntax tree. `CalcBaseVisitor` is a class that implements `CalcVisitor`, but with empty methods so that the tree is not actually traversed. `ExecVisitor` extends `CalcBaseVisitor` and defines the tree traversal.

Now study `CalcRun.java`. It expects an argument that is a named Calc source file. `CalcRun` creates a lexer and uses it to translate the source code into a token stream. Then it creates a parser and calls the parser's `prog()` method, which in turn calls `com()`, `expr()`, etc. The result of `prog()` is the syntax tree of the input source code. Finally a visitor is created and its `visit` method is called on the syntax tree in order to execute the Calc program.

Compile all the Java files:

```
…$ javac *.java
```

Run `CalcRun` with a selected source file, e.g.:

```
…$ java CalcRun test1.calc
16
56
72
```

Note the outputs: these are numbers printed by the "`put`" commands in the source program.

Now try:

```
…$ java CalcRun test2.calc
line 4:15 no viable alternative at character '/'
line 4:16 extraneous input '2' expecting EOL
```

The Calc parser prints an error message because this source program uses "`/`", but Calc has no such operator. (*Note:* The generated parser's error messages are not very informative. However, they always include a line number and column number, such as "4:15", so you can locate the error exactly.)

ANTLR provides a tool for visualising the parse trees produced by a grammar from an input file.

```
…$ java org.antlr.v4.gui.TestRig Calc prog -gui < test1.calc
```

You might find it convenient to define a script file containing `java org.antlr.v4.gui.TestRig $*`

In this command, `prog` specifies the non-terminal that you want to match. You can try other possibilities from the Calc grammar, for example `expr` to test expressions. It is also possible to enter input from the keyboard instead of a file, by omitting the `< test1.calc`. Alternatives to the `-gui` option are `-tree` to display a textual representation of the syntax tree, and `-tokens` to show how the input is separated into tokens.

Experiment by modifying the Calc grammar in `Calc.g4`. Try at least one of the following:

(a) Add comments (choosing your preferred syntax, such as "`/*…*/`" or "`//…`"). To do this, add a line in `Calc.g4` after the line defining `SPACE`, which defines the syntax of comments and then specifies `-> skip` to say that they should be ignored.

(b) Add a "`/`" operator. You will also need to modify `ExecVisitor.java`. Find the place where the behaviour of the other operators is defined, and extend it.

(c) Allow variable identifiers to consist of *one or more* letters (instead of just a single letter). If you make this change in the grammar and then compile the calculator, you will find that variables are only distinguished by the first letter of their name. To make them work properly, you can go into `ExecVisitor.java` and replace `int[] store = new int[26];` by a `HashMap` that will be indexed by the full variable names. You will need to make corresponding changes in the `visitSet` and `visitId` methods, where the store is used.

Whenever you modify `Calc.g4`, run ANTLR again to regenerate the `.java` files, then recompile these Java files.

If you make any mistakes in the *grammar*, ANTLR will print error messages. (*Note:* These error messages are not very informative, so it is wise to make only one modification at a time.)

## Familiarization with the Fun compiler

Go to the PL(H) Moodle page, download `Fun.zip`, and extract all the files into a new directory named `Fun`.

Study `Fun.g4`. It contains the grammar of Fun, expressed in ANTLR notation.

To make ANTLR generate a Fun syntactic analyser (lexer and parser), enter the following Linux command:

```
…$ java org.antlr.v4.Tool -no-listener -visitor Fun.g4
```

This should generate files named `FunLexer.java`, `FunParser.java`, `FunBaseVisitor.java` and `FunVisitor.java`. The `FunParser` class contains methods `prog()`, `com()`, `expr()`, etc., which work together as a modified form of recursive-descent parser. The parser's output is a syntax tree.

As before, you can use the ANTLR visualisation tool to see the syntax trees produced by the parser from Fun source files.

Study `FunCheckerVisitor.java`. It implements a visitor that will traverse a parse tree and enforce Fun's scope rules and type rules.

Study `FunEncoderVisitor.java`. It implements another visitor that will traverse a parse tree and generate SVM object code.

Study `SVM.java`. This class defines the representation of SVM instructions. It also contains a group of methods for emitting SVM instructions, i.e., placing them one by one in the code store; these methods are called by the Fun code generator. This class also contains a method `interpret()` that interprets the program in the code store.

Study `FunRun.java`. This driver program first compiles a named Fun source file to SVM object code. To help you to see what is going on, the program prints the AST and the SVM object code. Finally (if compilation was successful) the program interprets the object code. There are two other driver programs: `FunParse.java` just does syntactic analysis (parsing), and `FunCheck.java` does syntactic analysis and contextual analysis (typechecking).

Compile all the Java files:

```
…$ javac *.java
```

You will find several Fun test programs in the directory `tests`. Run the driver program with a selected source file:

```
…$ java FunRun tests/func.fun
.....
```

This particular test program repeatedly invites you to input an integer, and outputs that integer's factorial. It terminates when you input 0.

If you wish, you can make the interpreter print each instruction as it is executed. In `FunRun.java`, simply change the static variable `tracing` from `false` to `true`.

## Warm-up: extending Fun to allow multiple procedure/function parameters

In the Fun language, procedures and functions have either no parameters or one parameter. In this warm-up exercise, you will extend Fun so that procedures and functions can have any number of parameters. Formal parameters (in procedure and function definitions) and actual parameters (in procedure and function calls) will be separated by commas. The warm-up exercise is in three stages, corresponding to the three stages of the assessed exercise. Each depends on some of the lecture material. You might be able to work ahead of the lectures by studying the Fun compiler, but it's OK to take the warm-up one stage at a time.

### Warm-up stage 1 (depends on the lectures in the week beginning 24th October)

The file `Fun-multiple.g4` contains a new version of the grammar. Look at this file and compare it with `Fun.g4`. There is a new non-terminal, `formal_decl_seq`, which is defined to be a sequence of one or more `formal_decl`, separated by commas. The optional tag (`?`) has moved from the definition of `formal_decl` into the definitions of `proc_decl` and `func_decl`. This means that the case of no parameters will be handled as a special case, and the general case is a non-empty sequence of parameters. It would be nice for the general case to be a sequence, empty or non-empty, of parameters, but the problem is that the comma only appears when we have at least two parameters.

Rename `Fun-multiple.g4` to `Fun.g4`, overwriting the original `Fun.g4`. Now you can compile `FunParse`:

```
…$ javac FunParse.java
```

and you should be able to parse (syntax check) `tests/multiple.fun`:

```
…$ java FunParse tests/multiple.fun
```

### Warm-up stage 2 (depends on the lectures in the week beginning 31st October)

The next step is to extend the contextual analysis phase, which is defined in `FunCheckerVisitor.java`. The file `Type.java` already defines the class `Type.Sequence`, which represents a sequence of types; this class is not used

yet, but the idea is to use it to represent the parameter types of a procedure or function. The same file also defines `Type.EMPTY`, representing an empty sequence of types.

Make the following changes to `FunCheckerVisitor.java`.

- In the method `predefine`, which defines the types of Fun's built-in procedures and functions, change the parameter type of `read` from `Type.VOID` to `Type.EMPTY`. Change the parameter type of `write` to be a `Type.Sequence` containing just `Type.INT` (you will have to do a little programming to construct this).
- Change the definition of `MAINTYPE` so that the parameter type is `Type.EMPTY`.
- In the methods `visitProc` and `visitFunc`, in the third line, instead of calling `ctx.formal_decl()`, call `ctx.formal_decl_seq()`. This is necessary to match the new grammar. The result type of this call is `FunParser.formal_decl_seqContext`. If it is null, meaning that there are no parameters, then the variable `t` should be set to `Type.EMPTY` instead of `Type.VOID`.
- Because we have added `formal_decl_seq` to the grammar, with the label `formalseq`, we need to add a method `visitFormalseq`. If you look in the file `FunBaseVisitor.java` you can see what the method header should be. The method needs to visit every item in `ctx.formal_decl()`, which has type `List<FunParser.Formal_declContext>`. Visiting an item returns a `Type`. These values need to be collected into an `ArrayList` and used to construct a `Type.Sequence`, which is returned.
- The method `visitFormal` can be simplified because the result of `ctx.type()` is never `null`. This is because the optional clause in the grammar is now `formal_decl_seq`, and if we have one, then it must be a non-empty sequence of declarations.
- `visitProccall` and `visitFunccall` need to be modified because `ctx.actual_seq()` might return `null`. In this case we construct an empty sequence of types; otherwise we visit the result of `ctx.actual_seq()` to get the sequence of types.
- Replace `visitActual` by `visitActualseq`, which needs to visit every item in `ctx.expr()` and construct a `Type.Sequence` of their types.

Now you can compile `FunCheck.java`, and you should be able to typecheck `tests/multiple.fun`.

```
…$ javac FunCheck.java

…$ java FunCheck tests/multiple.fun
```

**Warm-up stage 3 (depends on the lectures in the week beginning 7th November)**

Finally, a few changes are necessary in `FunEncoderVisitor.java`.

- In `visitProc` and `visitFunc`, on the 8th line, replace `FunParser.formal_declContext` by `FunParser.formal_decl_seqContext`, and replace `ctx.formal_decl()` by `ctx.formal_decl_seq()`.
- Define the method `visitFormalSeq`; it just has to visit everything in `ctx.formal_decl()`.
- `visitFormal` can be simplified in the same way as in `FunCheckerVisitor`.
- In `visitProccall` and `visitFunccall`, use `ctx.actual_seq()` instead of `ctx.actual()`, but it might return `null`, so test for this. If it is `null` then there is no need to call `visit(ctx.actual_seq())`.
- Similarly to `FunCheckerVisitor`, replace `visitActual` by `visitActualseq`, which needs to visit every item in `ctx.expr()`.

Now you can compile `FunRun.java`, and you should be able to compile and run `tests/multiple.fun`.

```
…$ javac FunRun.java

…$ java FunRun tests/multiple.fun
```

# Assessed Exercise: extending Fun with a switch statement

In this assignment you are required to extend Fun by adding a switch command similar to the one in Java and C. The assignment consists of three stages: syntactic analysis, contextual analysis, and code generation.

You can decide whether to start from your version of Fun that has been extended with multiple parameters (the warm-up exercise), or from a fresh copy of the original Fun.

## Description of the switch command, and examples

Adding a *switch command*. The following Fun function contains a switch command with an integer expression:

```
func int test (int n):
    int r = 0
    int s = 0
    switch n:
        case 1:
            r = 1
            s = 2
        case 2..4:
            s = 3
        default:
            r = 4
    .
    write(s)
    return r
.
```

The following Fun function contains a switch command with a boolean expression:

```
func bool invert (bool b):
   bool x = false
    switch b:
        case true:
            x = false
        default:
            x = true
    .
   return x
.
```

Note the following points about the syntax, typing rules and semantics of the switch command.

- The expression being tested (i.e. the expression that appears after the **`switch`** keyword) can be any integer or boolean expression.
- Each guard (i.e. the value or values that appear after a **`case`** keyword) has one of three possible forms. (1) An integer literal. (2) A boolean literal. (3) An integer range, such as `2..4` in the first example; this case is selected when the value of `n` is either 2, 3 or 4. Note that only literal values, not arbitrary expressions, are allowed in guards.
- All of the guards must have the same type as the expression being tested.
- All of the guards must be non-overlapping. This means that guards of forms (1) and (2) must all have different values, and the ranges in guards of form (3) must not overlap with other ranges or with guards of form (1).
- There can be any number of cases.
- There must be exactly one **`default`** case.
- The code for each case can be any sequence of commands.
- There is no fall-through, therefore no need for a **`break`** keyword. At the end of the sequence of commands associated with a particular case, execution jumps to the end of the switch command. This is true even if the sequence of commands is empty.

## Assignment stage 1: syntactic analysis

Add the switch-command to `Fun.g4`. Remember to extend the lexicon as necessary. Make sure your grammar corresponds to the examples given above; the grammar needs to be general enough, but not so general that it allows incorrect syntax.

Add your own name and the date to the header comment in `Fun.g4`. *Clearly highlight* all your modifications, using comments like "`// EXTENSION`".

Use ANTLR to regenerate `FunLexer.java` and `FunParser.java` (and the visitor classes, but they are not relevant at this stage) then recompile them.

Write one or more test Fun programs containing switch-commands. Test your extended syntactic analyser by running the simplified driver program `FunParse` with each of these test programs, and see whether it accepts them or produces appropriate syntax error messages. You can also use the ANTLR visualisation tool to check the syntax trees.

## Submission (stage 1)

The deadline for stage 1 is Tuesday 0**8/11/2016** (Glasgow) or Friday **11/11/2016** (Singapore) at **17:30**. Submit through Moodle. What you should submit is a zipped directory containing all the files for your modified Fun parser, including the files that you have not changed. We want to be able to compile your parser without needing to change anything in your directory. Add your test programs to the `tests` directory. Include a brief (but honest!) status report in a file called `StatusReport.txt`.

## Assignment stage 2: contextual analysis

Now that you have extended the Fun grammar, the `FunCheckerVisitor` class does not implement all of the required methods. You need to add implementations of the missing methods, so that all the necessary type checking is done. Also you need to implement a check that there are no repeated or overlapping guards.

As before, add your own name and the date to the header comment in `FunCheckerVisitor.java`. *Clearly highlight* all your modifications, using comments like "`// EXTENSION`".

Recompile the `FunCheck` driver program.

Test your extended contextual analyser by running `FunCheck` with each of your test programs, and see whether it correctly performs type checks and other analysis (no repeated or overlapping guards). Your test programs should include one that violates all the switch command's rules.

## Assignment stage 3: code generation

Extend the Fun code generator as follows.

Start by devising a code template for a switch command. This should combine code to evaluate the switch command's expressions, code to compare the value of the expression with each guard in turn, and the necessary conditional and unconditional jumps to ensure that only the appropriate case (i.e. sequence of commands) is executed.

When you understand the code template, define the missing methods in the `FunEncoderVisitor` class. You will almost certainly find that you need to add a couple of instructions to the SVM in order to be able to repeatedly compare the value of the expression with the guards. Therefore you will need to modify `SVM.java`.

As before, add your own name and the date to the header comment in `FunEncoderVisitor.java` and `SVM.java`. *Clearly highlight* all your modifications, using comments like "`// EXTENSION`".

*Note:* Include your code template as a comment in `FunEncoderVisitor.java`. This should be in a separate part of the file, not interspersed between other code. You will receive marks for a reasonable code template even if your code generator does not work as intended. Note that the code template does not mean the Java code that you have put into a `visit` method. It means a schematic outline of the code that will be generated, showing the structure of tests and jumps. The lecture slides show code templates for the if statement (slide 8-16) and the while statement (slide 8-31).

Recompile the `FunRun` driver program.

Test your extended contextual analyser and code generator by running `FunRun` with each of your test programs, and see whether it performs proper scope/type checks and generates correct object code.

There are two ways to verify whether the compiler generates correct object code – use both!

1. Visually inspect the object code.
2. See what happens when the object code is interpreted. If the object code's behaviour is unexpected, your compiler must be generating incorrect object code.

## Submission (stages 2 and 3)

The deadline for stages 2 and 3 is Tuesday **29/11/2016** (Glasgow) or Friday **02/12/2016** (Singapore) at **17:30**. Submit through Moodle. What you should submit is a zipped directory containing all the files for your modified Fun compiler, including the files that you have not changed. We want to be able to compile your compiler without needing to change anything in your directory. Add your test programs to the `tests` directory. Include a brief (but honest!) status report in a file called `StatusReport.txt`.

## Help and support

Your lecturer and a demonstrator will be in the lab to help you if needed.

You may collaborate with other students to familiarize yourself with ANTLR and the Fun compiler. However, *assignment stages 1–3 must be your own unaided work*.

Your stage 1 work will be marked and returned to you promptly. You are then free to modify your `Fun.g4` in the light of your feedback, but your stage 1 work will not be re-assessed. If you want to, you can use the model answer `Fun.g4` as the basis for stages 2 and 3.

## Schedule

You can work at your own pace, but here is a suggested schedule:

| | Glasgow | Singapore |
|---|---|---|
| Familiarization with ANTLR | by 25/10/2016 | by 28/10/2016 |
| Familiarization with Fun compiler; warm-up stage 1 | by 01/11/2016 | by 04/11/2016 |
| Assignment stage 1 deadline | by 08/11/2016 | by 11/11/2016 |
| Warm-up stage 2 & assignment stage 2 | by 15/11/2016 | by 18/11/2016 |
| Warm-up stage 3 | by 22/11/2016 | by 25/11/2016 |
| Assignment stages 2 & 3 deadline | by 29/11/2016 | by 02/12/2016 |

## Assessment

Your work will be marked primarily for correctness. However, marks may be deducted for code that is clumsy, hard to read, or very inefficient. Marks will also be deducted for a missing or misleading status report. Your total mark will be converted to a percentage and then to a band on the 22-point scale.

The assessment scheme will be:

| | |
|---|---|
| Stage 1 (syntactic analysis) | 5 marks |
| Stage 2 (contextual analysis) | 10 marks |
| Stage 3 (code generation) | 10 marks |
| *Total* | *25 marks* |