University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Programming Languages 3**				
Coursework Number	**1**				
Deadline	**Time:**	**17.30**	**Date:**	**13/11/2014 and 04/12/2014 (Glasgow)** **14/11/2014 and 05/12/2014 (Singapore)**	
% Contribution to final course mark	**20%**		**This should take this many hours:**	**15**	
Solo or Group ✓	**Solo**	✓	**Group**		
Submission Instructions	**Submit by e-mail. See detailed instructions on pages 5 and 6 of the assignment.**				
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 3

Coursework Assignment (2014-15)

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: **13/11/2014** (stage 1) and **04/12/2014** (stages 2 and 3).

Singapore: **14/11/2014** (stage 1) and **05/12/2014** (stages 2 and 3).

The assignment contributes **20%** of the assessment for the PL3 course.

Familiarization with ANTLR

ANTLR runs on Linux machines. To use ANTLR, ensure that your `CLASSPATH` includes "`.`" and `/usr/local/antlr/antlr-3.5.2-complete.jar`.

Go to the PL3 Moodle page, download `Calc.zip`, and extract the files (`Calc.g`, `CalcRun.java`, and some test files) into a new directory named `Calc`.

Study `Calc.g`. It contains the grammar of Calc, expressed in ANTLR notation. It also contains *actions* that will execute the commands and evaluate the expressions in a Calc program. These actions use an array that has space for all the 26 variables provided by Calc. That array, `store`, is declared in the `@members` section of `Calc.g`.

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

```
…$ java org.antlr.Tool Calc.g
```

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. Notice that ANTLR has pasted the actions from `Calc.g` into these parsing methods.

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. While parsing the token stream, these methods also execute the actions that ANTLR pasted into them.

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.)

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

(a) Add comments (choosing your preferred syntax, such as "`/*…*/`" or "`//…`").

(b) Add a "`/`" operator.

(c) Allow variable identifiers to consist of *one or more* letters (instead of just a single letter).

Whenever you modify `Calc.g`, make ANTLR regenerate `CalcLexer.java` and `CalcParser.java`, 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.)

On the other hand, if you make any mistakes inside *actions*, these mistakes will not be noticed by ANTLR. Instead, ANTLR blindly pastes the actions into the generated Java files, leaving it to the Java compiler to detect these mistakes and print error messages.

Familiarization with the Fun compiler

Go to the PL3 Moodle page, download `Fun.zip`, and extract all the files into a new directory named `Fun`.

Study `Fun.g`. It contains the grammar of Fun, expressed in ANTLR notation. It also contains *tree-building operations* that will translate a Fun source program to an AST.

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

```
…$ java org.antlr.Tool Fun.g
```

This should generate files named `FunLexer.java` and `FunParser.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 an AST. The translation from token stream to AST is specified by *the tree-building operations* in `Fun.g`.

Study `FunChecker.g`. It contains a tree grammar of Fun ASTs, expressed in ANTLR notation. It also contains actions that will enforce Fun's scope rules and type rules.

To make ANTLR generate a Fun contextual analyser, enter the following Linux command:

```
…$ java org.antlr.Tool FunChecker.g
```

This should generate a file named `FunChecker.java`. The `FunChecker` class contains methods that work together to walk the AST and enforce Fun's scope rules and type rules.

Study `FunEncoder.g`. It contains the tree grammar of Fun ASTs, expressed in ANTLR notation. It also contains actions that will allocate addresses and emit SVM object code.

To make ANTLR generate a Fun → SVM code generator, enter the following Linux command:

```
…$ java org.antlr.Tool FunEncoder.g
```

This should generate a file named `FunEncoder.java`. The `FunEncoder` class contains methods that work together to walk the AST, allocate addresses and emit 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.

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`.

Assignment: extension to Fun

In this assignment you are required to extend Fun by adding two new forms of command, corresponding to new loop structures. The assignment consists of three stages (1, 2 and 3) and each stage has two parts (A and B). Part B is more difficult because you are given less information. The marking scheme is such that perfect answers to Part A will get a grade of B1. To get an A grade you will need to attempt Part B.

Part A

Adding a *for-command*. The following Fun function contains a for-command:

```
func int fac (int n):  # returns n!
   int f = 1
   for i = 2 to n:
      f = f*i .
   return f
.
```

This particular for-command makes the control variable `i` range from 2 up to the value of `n`. The for-command's body "`f = f*i`" is executed repeatedly, once for each value of the control variable. If the value of `n` happens to be less than 2, the for-command's body is not executed at all.

The Fun grammar is extended with the for-command as follows:

```
com  =  ident '=' expression                          – assignment command
     |  ident '(' actual ')'                          – procedure call
     |  'if' expr ':' seq-com
           ('.' | 'else' ':' seq-com '.')             – if-command
     |  'while' expr ':' seq-com '.'                  – while-command
     |  'for' ident '=' expr 'to' expr ':'
           seq-com '.'                                – for-command
```

The for-command '`for`' *ident* '`=`' *expr1* '`to`' *expr2* '`:`' *seq-com* '`.`' must respect the following scope/type rules:

- The control variable *ident* does not need to be declared before the for-command. It is created as a local variable of type **`int`** whose scope extends from the beginning of the for-command to the end of the current local block (procedure or function body).
- Both *expr1* and *expr2* must be of type **`int`**.

The for-command '`for`' *ident* '`=`' *expr1* '`to`' *expr2* '`:`' *seq-com* '`.`' has the following semantics:

1. Create a new local control variable *ident.*
2. Assign the value of *expr1* to the control variable *ident*.
3. If the value of the control variable is greater than the value of *expr2*, terminate the for-command.
4. Execute the body *seq-com*.
5. Increment the control variable.
6. Continue the for-command at step 3.

Part B

Adding either a *repeat-until* loop or a *do-while* loop. You can choose which kind of loop to add. To understand how these loops are supposed to behave, you will need to do some research on the web. The *repeat-until* loop is found in Pascal and related languages. The *do-while* loop is found in Java.

Assignment stage 1: syntactic analysis

Part A

Extend the Fun syntactic analyser as follows.

Decide how you will represent a for-command by an AST. The AST will have to include the relevant parts of the for-command: its control variable, the two expressions, and its body.

Add the for-command to `Fun.g`, using ANTLR notation. Remember to extend the lexicon as necessary.

Add a tree-building operation to translate the for-command to the corresponding AST.

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

Use ANTLR to regenerate `FunLexer.java` and `FunParser.java`, then recompile them:

```
…$ java org.antlr.Tool Fun.g
…$ javac FunLexer.java FunParser.java
```

Write one or more test Fun programs containing for-commands. Test your extended syntactic analyser by running the simplified driver program `FunSA` with each of these test programs, and see whether it builds correct ASTs.

Part B

Decide which kind of loop you are going to implement. Define the grammar, scope rules and semantics in the same way as the definitions of the *for-command* above. Then build the extended lexer and parser in the same way as for Part A, write test programs, and check the ASTs.

Submission (stage 1)

The deadline for stage 1 is Thursday **13/11/2014** (Glasgow) or Friday **14/11/2014** (Singapore) at **17:30**. Submit by e-mail to Simon.Gay@glasgow.ac.uk (Glasgow) or Malcolm.Low@SingaporeTech.edu.sg (Singapore). Attach a copy of your extended `Fun.g` and your test programs. For Part B, also attach a file describing the scope rules and semantics for your chosen form of loop. The body of your e-mail should contain a brief (but honest!) status report.

Assignment stage 2: contextual analysis

Part A

Extend the Fun contextual analyser as follows.

Add an AST pattern for the for-command to `FunChecker.g`. Then add actions to perform the necessary scope/type checks.

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

Use ANTLR to regenerate `FunChecker.java`, then recompile it:

```
…$ java org.antlr.Tool FunChecker.g
…$ javac FunChecker.java
```

Test your extended contextual analyser by running `FunRun` with each of your test programs, and see whether it performs proper scope/type checks. Your test programs should include one that violates all the for-command's scope/type rules. (*Note:* At this stage `FunRun` will fail if it attempts code generation, since you have not yet added the for-command to the code generator. You will fix that problem in stage 3.)

Part B

Similarly extend and test the Fun contextual analyser to include your chosen form of loop.

Assignment stage 3: code generation

Part A

Extend the Fun code generator as follows.

Start by devising a code template for a for-command. This should combine code to evaluate the for-command's two expressions, code to execute the for-command's body, conditional and/or unconditional jumps, and instructions to initialize, test, and increment the control variable.

Add an AST pattern for the for-command to `FunEncoder.g`. Then add actions to generate the code as specified by your code template.

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

Note: Include your code template as a comment in `FunEncoder.g`. You will receive marks for a reasonable code template even if your code generator does not work as intended.

Use ANTLR to regenerate `FunEncoder.java`, then recompile it:

```
…$ java org.antlr.Tool FunEncoder.g
…$ javac FunEncoder.java
```

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.

Part B

Similarly extend and test the Fun code generator to include your chosen form of loop.

Submission (stages 2 and 3)

The deadline for stages 2 and 3 is Thursday **04/12/2014** (Glasgow) or Friday **05/12/2014** (Singapore) at **17:30**. Submit by e-mail to Simon.Gay@glasgow.ac.uk (Glasgow) or Malcolm.Low@SingaporeTech.edu.sg (Singapore). Attach a copy of your extended `Fun.g`, `FunChecker.g`, and `FunEncoder.g` and your test programs. The body of your e-mail should contain a brief (but honest!) status report.

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.g` in the light of your feedback, but your stage 1 work will not be re-assessed.

Schedule

You can work at your own pace, but here is a suggested schedule (add one day for Singapore):

Familiarization with ANTLR	by 30/10/2014
Familiarization with Fun compiler	by 06/11/2014
Assignment stage 1	by 13/11/2014
Assignment stage 2	by 20/11/2014
Assignment stage 3	by 04/12/2014

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 grade on the 22-point scale.

The assessment scheme will be:

Stage 1 (syntactic analysis)	A: 10 marks, B: 3 marks
Stage 2 (contextual analysis)	A: 10 marks, B: 3 marks
Stage 3 (code generation)	A: 14 marks, B: 4 marks
Total	*44 marks*