MANCHESTER METROPOLITAN UNIVERSITY
Department of Computing and Mathematics

COURSEWORK COVER SHEET

Course unit:	6G6Z1110 Programming Languages: Principles and Design
Lecturers:	Ryan Cunningham
Assignment number:	1EXAM50
Assignment type:	Individual
Proportion of marks available:	50%
Hand in format and mechanism:	Moodle
Deadline	**7th May 2020 – 9:00 PM**

Learning Outcomes Assessed

- Assess and critically evaluate the benefits and disadvantages of optimisation methods for different processor architectures.
- Describe and explain the principles governing each phase of the compilation process and the role of each of the basic components of a standard compiler.
- Apply principles of compiler design to develop a functional compiler.

It is your responsibility to ensure that your work is complete and available for reference and submission during your summer exam for this unit. You should make at least one full backup copy of your work.

Exceptional Factors affecting your performance: see Regulations for Undergraduate Programmes of Study: http://www.mmu.ac.uk/academic/casqe/regulations/assessment/docs/ug-regs.pdf

Plagiarism: Plagiarism is the unacknowledged representation of another person's work, or use of their ideas, as one's own. MMU takes care to detect plagiarism, employs plagiarism detection software, and imposes severe penalties, as outlined in the Student Handbook (http://www.mmu.ac.uk/academic/casqe/regulations/docs/policies_regulations.pdf and Regulations for Undergraduate Programmes (http://www.mmu.ac.uk/academic/casqe/regulations/assessment.php). Bad referencing or submitting the wrong assignment may still be treated as plagiarism. If in doubt, seek advice from your tutor.

Table of Contents

Dr Ryan Cunningham, E142, ryan.cunningham@mmu.ac.uk

6G6Z1110 Programming Languages: Principles and Design

Coursework Overview (Compilers)

Main Task

For this coursework, you will use a variety of tools to apply what you learn in the lectures and labs to develop a fully functional compiler. Your compiler will compile high-level code written in a language called *Decaf* (see language specification on Moodle), to unoptimized assembly code (GNU assembler) for an x64 Linux machine. Your code (**all .g4 and .py files**) should be thoroughly commented and any shortcomings should be noted

Assessment

The coursework is 50% of the assessment for this unit. The **deadline for submission is 7 May 2020, 9:00 PM**. The coursework will be marked as standard in accordance with the Rubrik on the last page of this document.

The assessment is marked out of a **total** of 100; you must complete 4 tasks (see Submission below and pages 4-5 for description), each worth 25% of the **total** assessment: ***Lexer*** (25 marks), ***Parser*** (25 marks), ***Semantic Checker*** (25 marks), and ***Code Generator*** (25 marks).

Submission

(1) You must **submit a .zip file to the 1EXAM50 area** on the Moodle page for this unit.

(2) The .zip file must contain the following files (note: **filenames are strict**):

Decaf.g4	The **lexer** and **parser** definition for the Decaf language, written in ANTLR4 according the language specification.
decaf-lexer.py	This Python3 file will take a Decaf source file as input and print the token stream representation of the source file according to your Decaf.g4 lexer rules.
decaf-parser.py	This Python3 file will take a Decaf source file as input and print a formatted string representation of the parse tree of the source file according to your Decaf.g4 lexer rules.
decaf-codegen.py	This Python3 file will take a Decaf source file and: (1) for semantically valid Decaf code it will output an equivalent x64 assembly program (a string or a text file is fine) (2) for semantically invalid Decaf source code it will output an informative error which would allow a Decaf programmer to fix/debug the error.
SymbolTable.py	This Python3 file is the Symbol Table implementation you will use to do semantic checking and code generation in the decaf-codegen.py file. You may or may not need to modify the SymbolTable.py given on Moodle, but wither way you need to submit the version you used for your compiler.

Your .g4 and .py files must be well-commented, including recognition of any rules/code that do not work according to specification. Comments are ANTLR: // or /* */ multiline, Python3: #

Testing

Test files are provided to help you test your code, but it is not possible to provide test files that test all cases. It is ultimately *your responsibility to test your compiler by creating customised tests*.

Software

You may use any operating system that is capable of running Python 3 and installing the ANTLR4 python3 runtime binding – that includes Windows, MacOS and Linux.

You will need to install the following software:

(1) Python 3

It is recommended to install Python3 via Anaconda (https://www.anaconda.com/distribution/), or you can install Python3 directly (https://www.python.org/downloads/).

(2) Antlr4

Once you have installed Python3, open a command prompt/terminal, or Anaconda prompt, and run one of the following commands to install Antlr4 bindings for python:

pip install antlr4-python3-runtime

pip3 install antlr4-python3-runtime

(Task 1) Lexical Analyser [25 marks]

The first phase of a compiler is lexical analysis (analysis of words and symbols). Your compiler will make use of **regular grammar** (Antlr syntax) to detect lexical patterns in some source text and output a stream of tokens which define valid Decaf symbols.

For example:

The source file below would produce the corresponding tokens stream representing the contents of that file.

Source File

```
class Program {
  int i;
}
```

Token Stream

```
CLASS
ID
LCURLY
INT
ID
SEMICOLON
RCURLY
```

******Your Task******

You are required to write **lexer rules** a grammar file, ***Decaf.g4***, according to the Decaf language spec, and a python file called ***decaf-lexer.py***, which prints the list of tokens detected by your compiler for a given input text file (see *Week 2 Exercises*).

(Task 2) Syntax Analyser [25 marks]

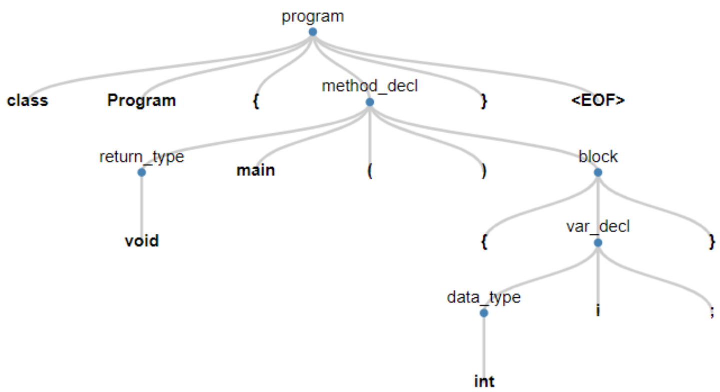
The second phase of a compiler is syntax analysis (analysis of sentences/token streams). In this phase, your compiler will use **context free grammar** (Antlr syntax) to define how tokens fit together to form valid Decaf syntax. Your compiler will take some source text, produce a token stream (task 1), then perform syntax analysis on that token stream. The Antlr tool will throw errors at this stage if invalid syntax is detected – you can use these errors to help you debug your grammar.

When your compiler successfully parses a token stream (i.e. it matches all tokens with the defined grammar), you will automatically produce a parse tree. For this, you have to interact with tree nodes and output text which is compatible with JavaScripts D3 tree format (more details on that in the labs).

Source File

```
class Program {
  void main() {
    int i;
  }
}
```

Parse Tree



******Your Task******

You are required to write **parser rules** in a grammar file, ***Decaf.g4***, according to the Decaf language spec, and a python file called ***decaf-parser.py***, which will output a formatted string representation of a parse tree for a given input text file (see *Week 4 Exercises*). That string should draw a parse tree using the provided JavaScript D3 html file.

(Task 3) Semantic Analyser [25 marks]

The third phase of your compiler will be a semantic analyser (analysis of valid code in the context of the program). In this phase you will use the Python classes produced by Antlr based on your grammar, to code high-level semantic rules for valid Decaf programs. You will write Python code which will detect, for example, if variables have been declared before use, etc. This task also requires development of a Python class which implements something called a Symbol Table (more details in labs). The symbol table keeps track of variables in scope and other metadata like line numbers so that errors can be reported to the programmer if found during compilation, therefore helping the programmer debug and remove errors.

Source File

```
1  class Program {
2    void main() {
3      int x;
4      boolean x;  // identifier declared twice
5    }
6  }
```

Python Console

```
error on line 4: identifier x redeclared in same scope
```

Your Task

You are required to produce a python file called ***decaf-codegen.py***, which will perform semantic checking on arbitrary, syntactically correct, Decaf source code, according to the Decaf language spec. Semantically valid Decaf programs should not produce any output, while semantically invalid Decaf programs should produce a detailed error message sufficient to guide a Decaf programmer to locate and fix the error.

(Task 4) Code Generation [25 marks]

The final phase in your compiler (but not the final stage in full optimizing compilers) is generating x64 assembly code from the code representation (in our case, the parse tree). You will be using the most direct and straightforward approach to generating machine code from your compiler: **pattern matching**. This means that for any given set of connected nodes in the parse tree, you will output the same template text (with dynamic modifications like stack pointer and register references).

Source Code

```
class Program {
    void main() {
        int a, b, c, d, e;
        a = 10;
        b = 20;
        c = 30;
        d = (a + b);
        e = (c * 3);
        e = d * e - 100;
        callout("printf", "%d %d\n", d, e);
    }
}
```

Generated x64 Assembly Code

```
.data
printf204_210: .asciz "%d %d\n"
.global main
main:
  movq %rsp, %rbp
  movq $10, %rax
  movq %rax, %r11
  movq %r11, -8(%rsp)
  movq $20, %rax
  movq %rax, %r11
  movq %r11, -16(%rsp)
  movq $30, %rax
  movq %rax, %r11
  movq %r11, -24(%rsp)
  movq -8(%rsp), %rax
  movq %rax, %r10
  movq %r10, -48(%rsp)
```

Your Task

You are required to produce a python file called ***decaf-codegen.py***, which will generate code for arbitrary, syntactically correct, Decaf source code, according to the Decaf language spec. For valid Decaf programs, your compiler will produce x64 Linux GNU assembly code, which can be assembled and linked with the C library. If you do not have access to a Linux system, you may use and online assembler and linker to test your compiler output (https://www.onlinegdb.com/online_gcc_assembler).

Marking

Rubric used to assess each of the 4 phases of your compiler

Criteria	**Bad Fail (0-29)**	**Fail (30-39)**	**3rd (40-49)**	**2:ii (50-59)**	**2:I (60-69)**	**1st (70-79)**	**Exceptional 1st (80+)**
Demonstrate a high degree of professionalism: correctness of solution; completeness of solution; demonstration of specific compiler concepts; clarity of code presentation (including comments).	Code is missing, or so limited as to provide no clear path to a solution in the compiler phase being assessed; there is no, or almost no evidence of demonstration of specific compiler concepts; code presentation is inadequate.	Code has limited correctness; Demonstration of specific compiler concepts is very limited; code presentation is poor.	Code performs incorrectly in a significant number of tests; Demonstration of specific compiler concepts is limited; code presentation is adequate.	Code performs correctly against a majority of testing, but some significant shortcomings are not recognised; some specific compiler concepts are demonstrated; code presentation is reasonable.	Code performs correctly against a majority of testing, but there are some significant shortcomings that have been recognised, or there some minor shortcomings that are not recognised; most specific compiler concepts are demonstrated; code presentation is good.	Code performs correctly against significant majority of testing, and any failures are noted as known shortcomings; all specific compiler concepts are demonstrated; code presentation is excellent.	Code performs correctly against all testing; all specific compiler concepts are demonstrated; code presentation is exceptional.