Introduction to Compilers

| **Module Leader** Chris Bates | | **Level** Five |
| --- | --- | --- |
| **Module Name** Introduction to Compilers | | **Module Code** 55-508227 |
| **Title** Implementation of a basic compiler | | |
| Individual submission | **Weighting** 50% | **Magnitude** 2,000 word equivalent |
| **Submission date and time** 3 p.m. on Thursday 31st March, 2022. | **Blackboard submission** | **Format** Source code archive |
| Planned feedback date | **Mode of feedback** Verbal | **In-module retrieval available** No |
| This assignment partially covers the following learning outcomes:   * Understand the principles which lie beneath programming language design. * Design formal representations of language constructs. * Implement the basic components of a simple compiler. | | |

# Introduction

In this assignment you must implement a compiler based on a grammar that I have provided, and use it to convert simple programs into JavaScript.

This assignment is worth 50% of the module’s marks.

# The task

On Blackboard you will find the specification of a simple imperative programming language in the Assessment folder. In this task you will write a compiler for that language. You will write a few small programs using the language and use the compiler to translate those into valid JavaScript.

You may use either the visitor or listener implementations. Each will be marked in the same way and to the same standards.

Your solution must:

1. Use ANTLR and Java.
2. Build a parse tree for valid programs. You should write code which demonstrates that the tree can be walked.
3. Extend your application so that:
   * A user can search the parse tree for language elements.
   * The parse tree is displayed.
   * Subtrees can be displayed.
4. Transform simple programs into JavaScript.

A sample grammar is available If you feel that your own grammar lacks sufficient richness to be used in tackling this problem then you may use one of the provided solutions instead. You must, though, write your own sample programs.

# Walkthrough

Your work will be assessed at a walkthrough demonstration in which you:

* Show your compiler building each of your sample programs.
* Discuss the design and implementation of the compiler.
* Demonstrate each item from the marking scheme.
* Walk through some of the sections of code with which you are most pleased.

Your walkthrough will last no longer than twenty minutes. Details of times and locations will be given nearer the time.

# Submission

You must submit a single zip file containing all of your code plus instructions for using it and sample programs through the assignment handler on Blackboard.

# 

# Marking Scheme

Your work will be marked using a grade-based approach that is described in a separate document available on the module’s Blackboard site and at <http://tinyurl.com/zkej95m>.

| **Aspect** | **Marks available** | **At pass level** | **At distinctive level** |
| --- | --- | --- | --- |
| **Creation of a parse tree** | 5 | A simple application is provided which may   * make some progress towards parsing the input * work on one type of valid program * accept code from the command line only | * A range of valid programs can be parsed. * Programs are read in from files. * The parse tree can be a (valid) subset of an entire program |
| **Displaying the parse tree** | 25 | * Some attempt is made to print the tree. This may be as a simple string. * Very simple trees can be output | * Trees are displayed successfully for most valid, and parsable, programs. * The output may be a useful intermediate representation such as   + S-expressions   + JSON documents   + XML structures |
| **Searching of the parse tree** | 20 | Some attempt is made which may   * accept a search criterion * build an appropriate data structure to hold the result * start to search the tree * attempt to return a result | * Search criteria can be entered and chained * Most valid programs can be searched * Searches return results where those results are available * Results are presented clearly and usefully * No matching data is still treated as a valid result |
| **Displaying subtrees** | 20 | * A subtree may be displayed – often from the root only * There is code to find the start of a subtree | * Subtrees are found and displayed * Code for searching and displaying trees is re-used sensibly |
| **Transform a program into JavaScript** | 30 | Some attempt is made to produce output. This may include   * creation of suitable templates * mapping of templates to rules * modification of the grammar to support the necessary set of rules | * A range of valid programs can be transformed into valid outputs that can be executed. * Code is clear and appropriate * Correct structures are used * Errors or edge-conditions are handled gracefully |