UNIVERSITY OF Hull

SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
Computer Science
ASSESSMENT DESCRIPTION 2017/18
(EXAM TESTS WORTH ≤15% AND COURSEWORK)

MODULE DETAILS:

Module Number:	08348	Semester:	1
Module Title:	Languages and Compilers		
Lecturer:	Eur Ing Brian Tompsett		

COURSEWORK DETAILS:

Assessment Number:	1	of	1		
Title of Assessment:	The SPL Language Compiler				
Format:	Program	Demonstration			
Method of Working:	Individual				
Workload Guidance:	Typically, you should expect to spend between	60	and	80	hours on this assessment
Length of Submission:	This assessment should be **no** more than: *(over length submissions **will be** penalised as per University policy)*	N/A - coding exercise			

PUBLICATION:

Date of issue:	2nd October 2017

SUBMISSION:

ONE copy of this assessment should be handed in via:	Canvas		If Other (state method)	Demonstration
Time and date for submission:	**Time**	14:00	**Date**	30th November 2017
If **multiple hand–ins** please provide details:	Canvas for source code and results + Demonstrations in Lab			
Will submission be scanned via TurnitinUK?	No			

The assessment must be submitted **no later** than the time and date shown above, unless an extension has been authorised on a *Request for an Extension for an Assessment* form: search 'student forms' on https://share.hull.ac.uk.
Canvas allows multiple submissions: only the **last** assessment submitted will be marked and if submitted after the coursework deadline late penalties will be applied.

MARKING:

Marking will be by:	Student Number

ASSESSMENT:

The assessment is marked out of:	20	and is worth	50	% of the module marks
N.B If multiple hand-ins please indicate the marks and % apportioned to each stage above (i.e. Stage 1 – 50, Stage 2 – 50). It is these marks that will be presented to the exam board.				

ASSESSMENT STRATEGY AND LEARNING OUTCOMES:

The overall assessment strategy is designed to evaluate the student's achievement of the module learning outcomes, and is subdivided as follows:

LO	Learning Outcome	Method of Assessment *{e.g. report, demo}*
Intellectual Skills: 2	*Explain, with comprehension, a range of issues pertinent to compiler construction theory and techniques.*	Program, results & Demo
Intellectual Skills: 3	*Critically evaluate the relationship between computer architecture and programming language design and implementation, justifying any links between these areas*	Program, results & Demo
Practical Subject Specific Skills: 4.	*Select, use, build and critically evaluate a compiler for a simple language*	Program, results & Demo
Transferable Skills: 5	*Select, justify and use appropriate approaches, including some at the forefront of the subject / profession, to design, build, test and document programs*	Program, results & Demo

Assessment Criteria	Contributes to Learning Outcome	Mark
BNF	2,3,4	10%
Lexical Analyser	2,3,4,5	25%
Parser	2,3,4,5	25%
Synthesis	2,3,4,5	25%
Testing, Test Coverage, Presentation	2,3,4,5	15%

FEEDBACK

Feedback will be given via:	Feedback Sheet	Feedback will be given via:	Verbal (via demonstration)
Exemption (staff to explain why)			
Feedback will be provided no later than 4 'teaching weeks' after the submission date.			

This assessment is set in the context of the learning outcomes for the module and does not by itself constitute a definitive specification of the assessment. If you are in any doubt as to the relationship between what you have been asked to do and the module content you should take this matter up with the member of staff who set the assessment as soon as possible.

You are advised to read the **NOTES** regarding late penalties, over-length assignments, unfair means and quality assurance in your student handbook, which is available on Canvas - https://canvas.hull.ac.uk/courses/17835/files/folder/Student-Handbooks-and-Guides.

In particular, please be aware that:

- Up to and including 24 hours after the deadline, a penalty of 10%
- More than 24 hours and up to and including 7 days after the deadline; either a penalty of 10% or the mark awarded is reduced to the pass mark, **whichever results in the lower mark**
- More than 7 days after the deadline, a mark of zero is awarded.

- The overlength penalty applies to your written report (which includes bullet points, and lists of text. It does not include contents page, graphs, data tables and appendices). 10-20% over the word count incurs a penalty of 10%. Your mark will be awarded zero if you exceed the word count by more than 20%.

Please be reminded that you are responsible for reading the University Code of Practice on Academic Misconduct through the Assessment section of the Quality Handbook (via the SharePoint site). This govern all forms of illegitimate academic conduct which may be described as cheating, including plagiarism. The term ‘academic misconduct’ is used in the regulations to indicate that a very wide range of behaviour is punishable.

In case of any subsequent dispute, query, or appeal regarding your coursework, you are reminded that it is your responsibility, not the Department’s, to produce the assignment in question.

Created by Brian Tompsett based on original material by Peter Parsons

The aim of this project is for you to construct a fully working compiler for a small simple programming language, SPL. The compiler will read in SPL source code and produce ANSI C as output.

A full description of the language is given in this document, from which, over the course of the semester, you will construct a compiler using some of the techniques learnt in the lectures and by using the tools *Flex* and *Bison*.

The project was almost a whole semester long and is split in to six stages, with milestones staged throughout the course of the semester. An automated testing script will be used to test your compiler, so be sure to name the files correctly before submission.

The suggested component stages are as follows

Step 1

A BNF description of the SPL language described by the roadmaps given in this document. This could be a word file or a text file.
This should be saved for later assessment.

Step 2

A *Lexical Analyser* file called `spl.l` which will enable Flex or Lex to generate a working lexical analyser for the language. This will be demonstrated by printing the token recognized without using a parser. The use of the C symbol PRINT indicates that tokens are to be printed. These results should be saved for later assessment, as well as the source code for this printing version.

Step 3

A grammar file called `spl.y` for the parser without any semantic actions that recognizes valid SPL programs. This will be demonstrated by using the parser debug mode which will show the correct parsing for the various test cases. The C macro YYDEBUG is used to indicate parser debugging is enabled. These results should be saved for later assessment. The main program should be called `spl.c` .

Step 4

A partially complete *Parser* file (`spl.y`) containing all of the code necessary to create a parse tree using the Yacc or Bison tools. This can be demonstrated by using a printtree capability you have written into your code which is enabled by the C macro DEBUG. These results should be saved for later assessment.

Step 5

A complete *Parser* file (`spl.y`) which also includes a code generation function so that you now have a fully working compiler using the tools Yacc or Bison. If you are able you could also add code optimization of some form for further marks. This can be demonstrated by saving the output code in a file and building and running it. You should save both the generated code and the results of running it for later assessment, as well as the parser code.

Step 6

Collect the results from the standard tests and your own tests which will effectively demonstrate that all the features of the BNF, lexer, parser, code generator and optimizer have all been implemented correctly without the need for the compiler to be run again later, and package these results suitable for submission. Create a zip file containing the source files and results together with a README file clearly identifying and explaining what is included. The single zip file should be uploaded to Canvas for submission; the code will be

run through automated testing scripts and will also be examined during the scheduled final demonstration for marking.

If you are unable to complete any part for the whole SPL language it is suggested that you should reduce the language to a subset so that you can complete the compiler stages.

In order for you to test your compiler, some example programs written in SPL are supplied together with some skeleton files that might be helpful in building your compiler. These skeleton files will be explained in the lectures at an appropriate time. The files can be downloaded from in a from Canvas as a zip archive, or found on the U: network drive.

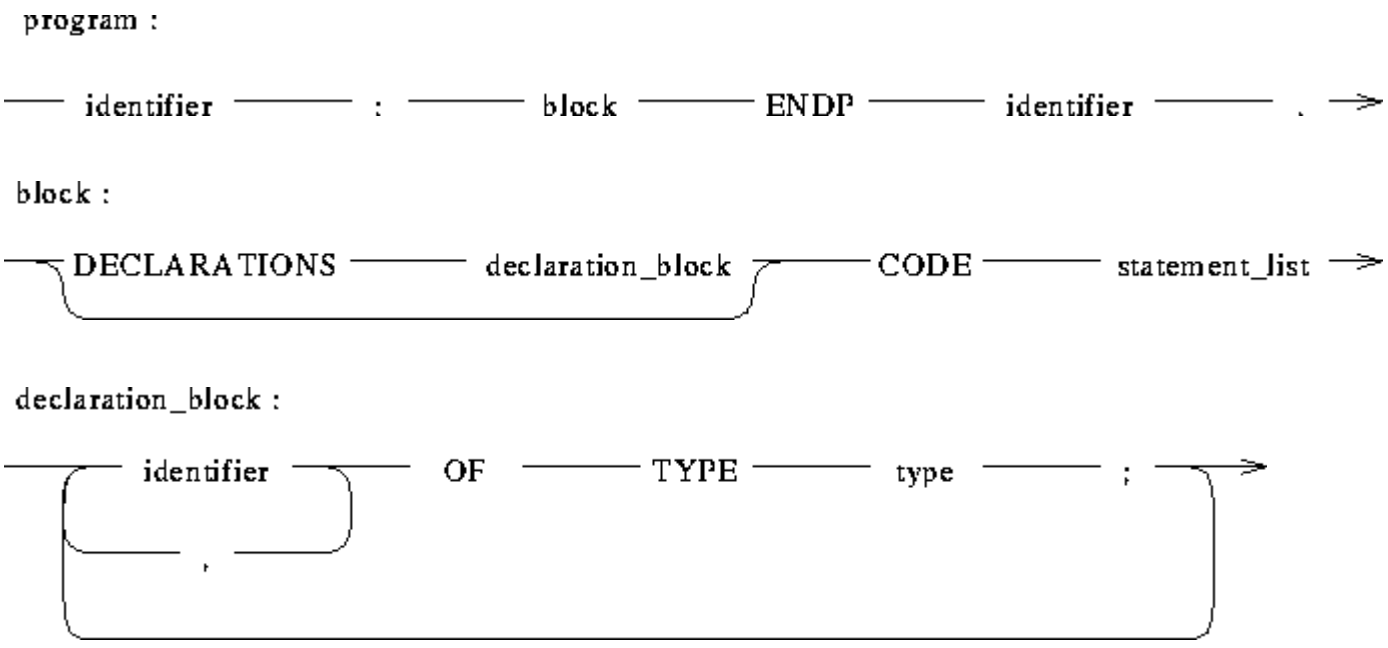
There are also video tutorials available to help you with this assignment on YouTube by following the link on Canvas.

SPL Language Description

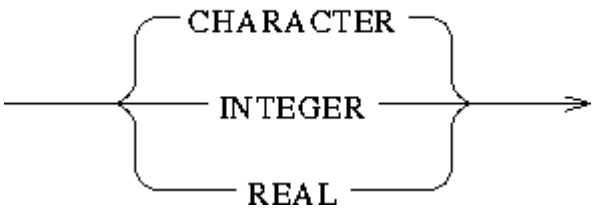
The SPL source resides in a single file. The following diagrams illustrate the complete structure of a SPL program. Identifiers follow the same rules as for PASCAL and other common languages. They must start with a letter and can only contain alphanumeric characters. Identifiers cannot be reserved words in the language.

Three data types are supported in the language, *characters*, *integers* and *reals*. Characters are always enclosed inside single quotes, 'a'. Integers and reals can both be either positive or negative. Reals contain a decimal point and must have at least one digit after the decimal point i.e. 3.0 is a valid real, 3. is not.

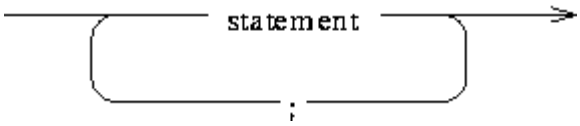
Programs are labelled with a program name, which must be a valid identifier. The identifier must be present both at the start and end of the program, and may not be used elsewhere in the program as a variable.



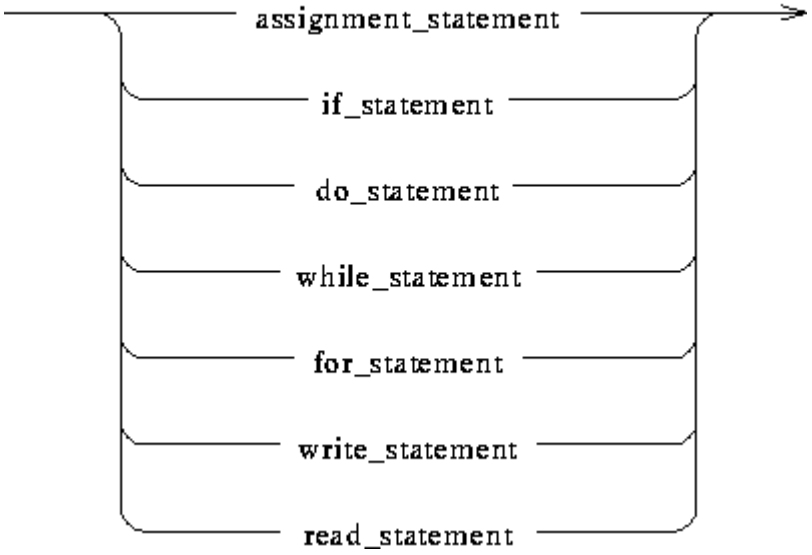
type :



statement_list :

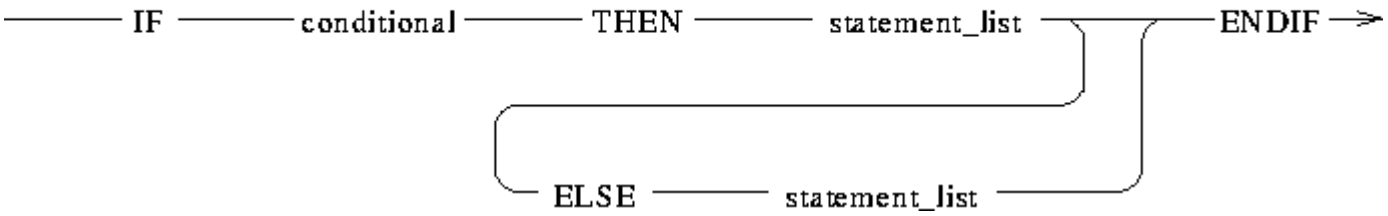


statement :



assignment_statement :

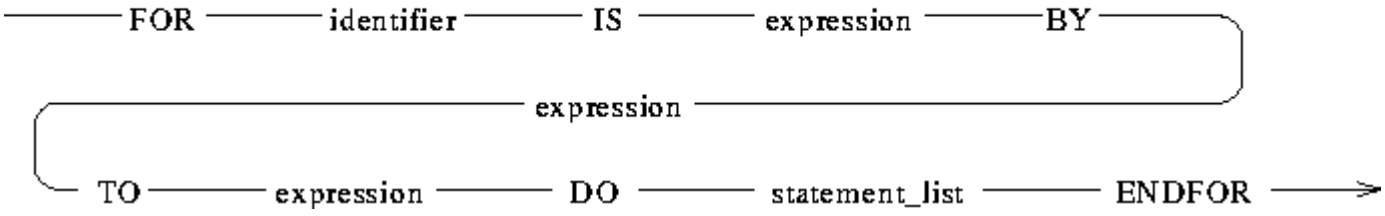
if_statement :



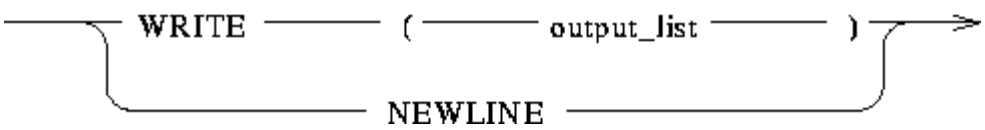
do_statement :

while_statement :

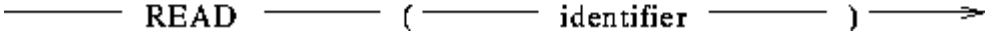
for_statement :



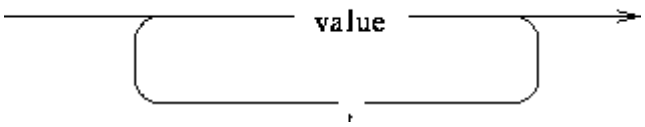
write_statement :



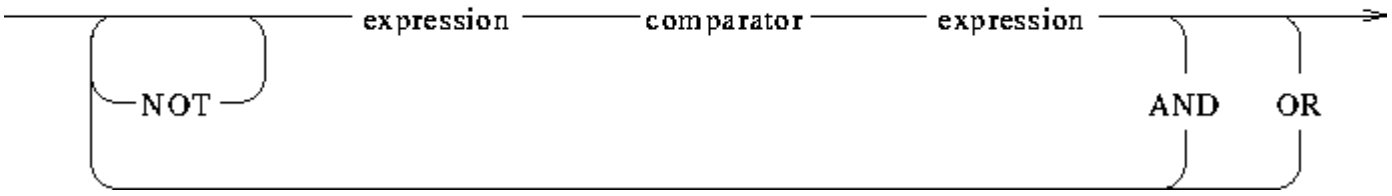
read_statement :



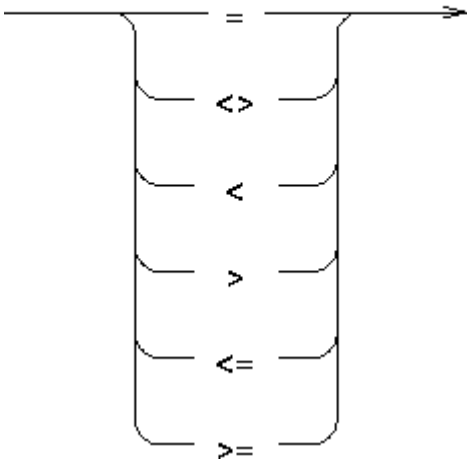
output_list :



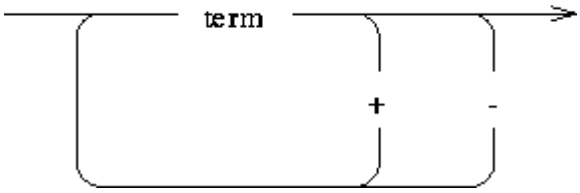
conditional :



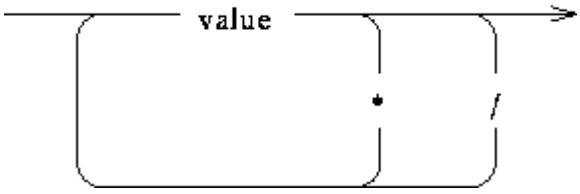
comparator :



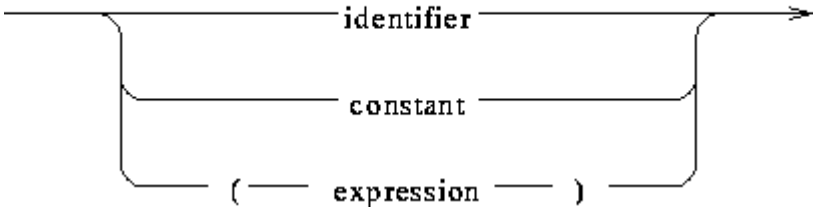
expression :



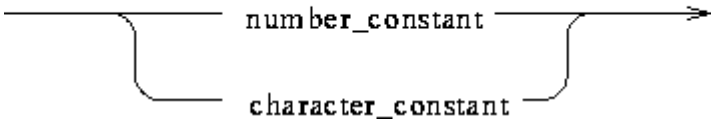
term :



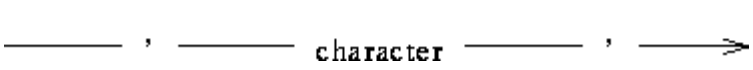
value :



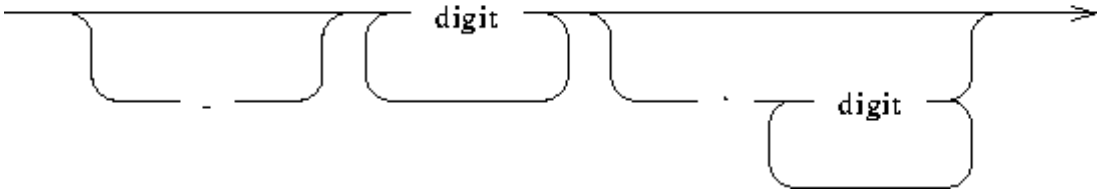
constant :



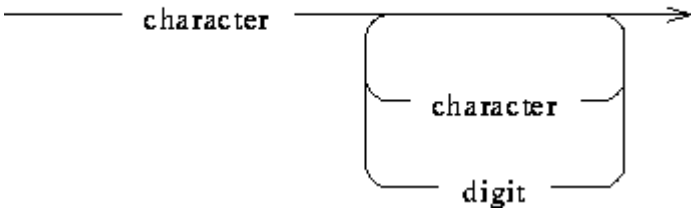
character_constant :



number_constant :



identifier :



Characters can be any upper or lower case letter.

Digits are any from 0 to 9.