Computer Science 330 Language Implementation

Lecture Times Tue 8, Wed 8, Fri 9, PLT2.

Tutorial Time Wed 9, GTL (Ground Floor Tutorial Laboratory).

Tutorials start in week 2.

Lecturer

Bruce Hutton Room 587 Ph 3737599 Ext 88299

I am responsible for teaching the whole course.

There will be three lectures and one tutorial per week. The tutorial is intended for going over examples, assignments, etc. Tutorials are run in an interactive fashion.

Prerequisites

CompSci 210 and CompSci 230 are the required prerequisites. A good knowledge of data structures and recursion is essential. Some knowledge of UNIX, such as that obtained from Compsci 215 is also desirable.

You will need to be very comfortable with object oriented programming, and the notion of extending classes and overriding method declarations. You will also need to be very comfortable with data structures, such as trees, linked lists, hashing, etc. Familiarity with recursion is essential.

You need to be comfortable with "programming in the large" – dealing with large programs written by others, where you make small changes, without necessarily understanding all the source code provided.

You need some familiarity with a command language, such as the shell language provided by UNIX, LINUX, or Cygwin.

You will need some familiarity with regular expressions, used for pattern matching in grep and most program editors.

In the latter part of the course, you need to be familiar with computer architecture, and assembly language programming. You will have to generate Alpha assembly language, suitable for my Alpha simulator. If you have not taken CompSci 210 recently, obtain a copy of the lecture notes from the CompSci 210 second semester web page, and read them.

Reasons for taking Computer Science 330

Computer languages provide the main way in which computer programmers tell the computer what tasks to perform. Having an understanding how computer languages are specified, what they mean, and how they are implemented, is fundamental to computer science.

As a programmer, you spend a lot of time interacting with compilers. Understanding how a compiler works means you have a much better understanding of what the compiler error messages mean. You can debug your programs more rapidly.

Having an accurate model for the compile time and run time data structures gives you a much better understanding of the computer language you use to write your programs. Your understanding of such topics as scope, mapping of identifiers to declarations, recursion, extending classes, overriding and overloading of methods, etc, becomes much clearer, with a corresponding increase in productivity.

Many programs involve the processing of input, and building of data structures to represent the input. For example, a spreadsheet program has to lexically analyse and parse the input in each cell. Simple programs are often generated by software, rather than people. The output from one program might need to be processed by another program. For example, a user might respond to an HTML web page, and generate a request that gets converted into an SQL query, and the reply from this might need to be converted back into HTML. Similarly, many programs involve the reformatting of text. The tools used by compiler writers to implement lexical analysis and parsing can also be used in the development of such programs. Knowing how to use these tools can have an enormous effect on your productivity - perhaps the lexical analysis/parsing portion of a program can be developed in 10% of the time it would take without the use of these tools. I often use JFlex and CUP to extract or reformat data from text files, that have nothing to do with computer languages. The generation of programs by treewalking data structures also has a lot in common with the generation of code by a compiler.

If you intend to be a programmer, you find most of your time is spent making small changes to large programs written by others. You have to make the changes in a manner that is consistent with the existing program. You have to design and build sophisticated data structures, and perform recursive treewalks of the data structures, to analyse and collect information. You have to create symbol tables, and search for information based on various keys and environments. All of these tasks are performed when writing compilers. The assignments in this course give you good practice in these tasks. The material in this course thus has direct practical application.

Student Representative

A student representative needs to be selected by the class.

Lecture Content

The material covered includes:

- **Lexical Analysis, using JFlex.** It is possible to specify the lexical analyser by writing regular expressions corresponding to the different kinds of tokens that occur in the language, and actions to perform when the regular expression is matched. The regular expressions corresponding to reserved words and special symbols are trivial to write because they are nothing but the text for the symbol. Only the regular expressions for identifiers and constants have any complexity. For example, an identifier could be described by the pattern "[A-Za-z][A-Za-z0-9]*".
- **Context free grammars, and LALR(1) parsing, using Java CUP.** It is possible to specify the grammar of a computer language using a context free grammar. Essentially we write grammar rules corresponding to each construct that can occur in the language, and actions to perform when the rules are matched. For example, a while statement could be defined by the rule "Stmt → WHILE LEFT Expr RIGHT Stmt". The grammar rules can be used to generate tables that drive the parser. The parser invokes the lexical analyser whenever it needs a token, and generates an abstract syntax tree, that represents the structure of the program being compiled. The ability to understand a grammar description of a computer language is extremely useful, as is the ability to design your own grammar for a simple language.
- **Building of the abstract syntax tree, and tree-walking.** The front portion of the compiler (the lexical analyser and parser) generates an abstract syntax tree. The back portion of the compiler performs multiple recursive treewalks of the abstract syntax tree, collecting information about the program, storing it in various tables, and annotating the tree with information, such as the environment in which to determine the meaning of identifiers in a construct, the declaration corresponding to each identifier application, the type of an expression, etc. These treewalks also perform type checking, generate code, etc.

- **Symbol table management.** One of the main data structures created when analysing a program is the symbol table (although I call this the compile-time environment). This contains information about each declaration that occurs in the program (e.g., type, environment, run time offset, etc). Because modern computer languages have complicated scope rules, symbol tables have a complex structure, and complex search algorithms. For example, in Java, an identifier can refer to a declaration in the local block, an enclosing block, or any class that extends the current class. Moreover, local declarations take precedence over more global ones. Thus we first search the local block, then the enclosing and extended environments. Languages such as Java also have to cope with overloading of identifiers - where there are multiple declarations in the same block with the same identifier.
- **Interpretation and code generation.** After lexical analysis, parsing, and the analysis phases of the compilation, we perform a final recursive treewalk, and either interpret the tree, or generate code that we then assemble and run separately. While not many programmers write compilers that generate code, it is fairly common to write simple interpreters. For example, if the user can type an expression into a text field, then the program may have to parse the expression, and evaluate it by a recursive treewalk. Recursion is a very powerful tool when writing compilers and interpreters.
- **Run time representation of data, for object oriented languages.** To implement a computer language, we need a well-defined model for how the data is stored at run time. For example, an object is typically composed of a pointer to an instance field table, and the first entry of this table is a pointer to an instance method table for the object. The fields and methods for superclasses occur in the tables before the fields and methods for subclasses, so by ignoring some information, an object belonging to a subclass appears to have the same structure as an object belonging to its superclass. We can implement overriding of methods by accessing methods indirectly through the method table, and making the method table depend on the actual type of the value, rather than the declared type of the variable. We can implement recursion by using a stack to allocate space for each invocation that has been invoked but not returned from.

Generally I will take a practical approach, rather than dealing much with the theory of lexical analysis and parsing. For example, I will not explain how regular expressions are used to build a finite state automaton, nor how the LALR(1) states are derived. In previous years, I have included material on top down parsing. I have deleted this material, because top down parsing normally requires the grammar to be modified, which introduces students to “bad habits” that cause problems when they attempt to design grammars. I have put more material in on grammar design (which of course makes it even more important to do the assignments).

The last part of the course will deal with the implementation of object-oriented languages. It will give you a clear model of how objects are represented at run time, how classes can be extended, and how the overriding of methods is implemented.

Note that I have changed my representation of objects from two year’s ago. Before, I represented an object by the address of a pair of (field table, method table) pointers. I now represent an object by the address of the field table. A pointer to the method table occurs at the beginning of field table. I have also changed the implementation of the getClass() method.

Hardware and Software

Apart from drawing of diagrams using Visio and generating tables using a spreadsheet, for assignment 1 it should be possible to do the assignments on most platforms: Windows, UNIX/LINUX, Mac (OS X).

You will need a modern version of Java, JFlex, CUP, and the Alpha simulator. Java should be at least jdk1.5. JFlex and CUP should be the ones provided from the computer science 330 web page

(I have altered them to fix some bugs and provide better debugging facilities). The Alpha simulator should be at least version 10.007. It is a good idea to get something that corresponds to UNIX/LINUX/Cygwin, with the bash command interpreter.

Apparently "Glimmer", available from "http://glimmer.sourceforge.net/" is a good programming editor for LINUX.

Setting up Cygwin or LINUX on your home machine

See the notes "GNU for Windows (Cygwin)" available from "References" on the department web page, and "Cygwin Installation Guide" available from "Support" on the department web page, to see how to install Cygwin.

Setting up Java, JFlex, and CUP on your Home Machine

A description of how to obtain and set up Java on your home machine is available from "How to install JDK on a PC" within the "technical support" web page of the Computer Science Department.

You can obtain files to set up your home directory from the 330 resources web page. One of the setup files is a directory LIB330, containing the files JFlex.jar, java_cup.jar and java_cup_runtime.jar, These files represent the source code for JFlex and CUP. The provided .bash_profile file defines a shell variable $LIB330 that refers to this directory, and is used by shell scripts for running JFlex and CUP.

Internet Access to our UNIX Machine

See the user note "Connecting from home via SSH", available from "Support" on the department web page, to see how to connect to our UNIX machines.

You might find that to copy whole directories between machines, it is best to tar and gzip them, copy them, then gunzip and untar them. To tar a directory, type

```
tar –c -z -f dirName.tar.gz dirName
```

This will create a tar/gzip file called dirName.tar.gz.

To look at the contents of a tar file, type

```
tar -t –z -f dirName.tar.gz
```

To gunzip/untar files, type

```
tar -x –z -f dirName.tar.gz
```

This will recreate the directory previously tarred.

Note that some versions of tar do not support the –z option, and silently ignore it. If this is the case, explicitly perform the gzipping yourself.

Gaining a Pass

Actually, almost everyone who persists with the course, and makes a real attempt at the assignments passes. But if you don't really do the assignments, do not expect to pass.

This course is officially recognized as a *practical* course by the Science Faculty. This means that to pass the course you have to pass two sections separately as well as getting an overall pass mark. You must get a pass in the *practical* component, namely the assignments. You must also get a pass in the *theoretical* component, namely the combined mark coming from the test and final examination.

The mark required to pass the practical and theoretical component individually are usually about 5 marks below the requirement to pass overall (e.g., 40% for a practical and theory pass, and 50% for a pass overall).

There are two reasons why we have this requirement to pass both components. Both are to do with motivating students to actually do the assignments.

The first reason is to motivate lazy but bright students. Some bright students believe that they can have an easy time throughout the semester, and then study hard in the last few weeks. They will find that this does not work in computer programming. If you have not done the work, you will fail the final examination.

The second reason is to motivate dishonest students to be honest. Some students think that if they can cheat in assignments, they can gain enough marks in assignments to compensate for a poor examination mark. In fact students who cheat in assignments rarely gain more than about 30% in the examination, and tend to fail not only the examination, but also overall.

In reality, almost all students who pass the theory component also pass the practical. Students who fail the practical do not know enough to pass the theory. At the lower levels we get lots of students who pass the practical but fail the theory. However, at the stage 3 level, most students who stay around until the end of the course pass.

The requirement for students to pass both parts of the course is for the student's benefit. We actually increase our pass rate by requiring students to pass both components of the course, because this motivates students to make a real attempt at the assignments.

Assignments

All programming assignments will involve the use of Java. I will use JFlex, a special purpose language for writing lexical analysers, and CUP, a special purpose language for writing parsers (Java equivalents of lex/flex, and yacc/bison). Both of these languages involve writing "rules" indicating the input to match, with "actions" indicating the action to perform when the match occurs. The actions and support code are essentially written in Java, and the JFlex and CUP compilers compile JFlex and CUP programs to generate Java programs, that then have to be compiled using the Java compiler. Familiarity with regular expressions for pattern matching (used in most text editors and the UNIX grep command) will help you with JFlex.

All source code for JFlex, CUP, programming assignments, and sample programs, is available over the web. I have modified CUP a bit to fix a couple of bugs related to error recovery, and greatly improve the diagnostic messages, when performing a debugging parse, outputting tables, etc. Thus it is important that you use the source code provided in the 330 web pages, and not obtain the source code from the web site for the recommended reading book.

You need some familiarity with UNIX, although the level of expertise required is not very great. The assignment template is set up in its own directory, and all you have to do is copy the whole directory to your own directory. There are shell scripts with names such as run.bash, createlexer.bash, createparser.bash, createclass.bash, etc, for running Java, JFlex, CUP, and javac so there is very little you need to know to run everything. Please make sure you keep your files with a sensible directory structure, and please back up files. Students who fail to manage this, usually waste days of time, and lose files and sometimes marks.

JFlex and CUP tend to prefer text files to be in UNIX format, with linefeeds, rather than CR or CR/LF.

There are two assignments.

- Assignment 1 will be half "theoretical" - performing a bottom up parse, building a tree, etc. This is very easy. The other half will be "practical" - implementing the parser and simple code generator for a language.
- Assignment 2 will be "practical" - implementing the parser and interpreter for a language.

Assignment Submission

All assignment submission is electronic. Submission should be via the assignment drop box. To find out how to make a submission, go to the Computer Science Department home page at http://www.cs.auckland.ac.nz, select "Personal Portal", and look for the "Web Drop Box" link.

Remember that no computer or network is entirely reliable, and people are even less reliable. To ensure that you are able to successfully submit your assignment:

- Make sure you get an electronic receipt when making an electronic submission. This is a file with a special code number that we are able to decrypt, to verify your submission. Failure to get this means that you were not successful in your submission. Moreover, make sure that it has your name on it, not somebody else's. Make sure **all** the files you intended to submit are listed. In previous years, some students got a receipt that listed no files as submitted (because one of the programming staff set up the assignment drop box to allow 0 files to be submitted).
- Make sure you have connected via NetAccount, as yourself. Otherwise you will submit your assignment as another person. Make sure you submission contains your full name, login name, student ID, etc. Just in case you happened to be connected via NetAccount as the wrong person.
- Make sure your submission compiles/assembles and runs just before you submit. You will get no marks for a submission that fails to compile/assemble. Don't make a few cosmetic changes (such as adding comments) at the end, without checking that everything still compiles/assembles and runs. Don't transfer files as in-line text through email. Lines get wrapped around, and the program no longer compiles.
- Make sure you really are trying to submit the correct files. Look at the files before you submit. Don't assume that just because it has the right name, that it is the right file. If the file submitted is a gzipped tar file, copy it to another directory, and untar/gzip it, to make sure that you really did tar/gzip it correctly. If a shell script is provided to tar/gzip your assignment into a single gzipped tar file, use it. You are less likely to make a mistake. It is not unusual for some students to create a gzipped tar file with nothing in it.
- Don't do silly things like renaming a Microsoft Word file as Assign1.txt, or a zip file as Assign2.tar.gz, so that you have a file name that is accepted by the assignment drop box. If I asked for a gzipped tar file, I want a gzipped tar file! Don't expect me to spend an hour trying various applications to unpack your assignment submission.
- If you fail to submit on the first attempt, wait a while and try submitting again. The system gets overloaded when assignments are due, and a simple retry is often sufficient. When the system really does go down, it is usually only for a short time.
- You might find that there are problems submitting assignments from home, due to their size. One student had problems in a previous year because their text files were padded with megabytes of nulls. Another student had multiple copies of their assignment nested inside the other, because they didn't know how to drag a file from one window to another. Many students submitted gzipped tar files that included all the class files as well. Assignments that should have been less than 60KBytes were puffed up to 10MBytes for various reasons. If you have problems, submit the assignment from within the laboratory. Don't keep overloading the system by trying to submit 10Mbyte files from home. You will stop other students submitting their assignment.
- You might find it is not possible to submit assignments from home if you connect via ADSL. Apparently NetAccount times out after half a minute, if you use ADSL. I wish they would fix this!

- It is possible to submit your electronic copy more than once - the last version will be the one marked. Make a submission each time your assignment gets to a reasonable stage. That way, it will not matter if your last submission is unsuccessful, or you lose all you files. You are strongly advised to do this. It also leaves an audit trail, in the event of someone copying your assignment.
- Contact the Computer Science Department programming staff, in the event of a system failure. In most cases, academic staff are not the most appropriate people to fix the problem. You should nevertheless make sure that we are aware of any problems that are occurring.
- Do not send email copies of assignments to staff, or expect us to mark hard copy. Email submissions are frequently lost or garbled. We have no easy way of making a submission for you, and hence no way of reliably getting your assignment to the markers. All students submit files with the same name, and we have no way of telling whose assignment is whose. I now automatically delete all email classified as junk mail. So, potentially email from obscure mail servers will never be received by me. Also, I cannot verify who the sender is.
- Email submissions will not be marked. The whole point of keeping the assignment drop box open after the due date is to cope with the system going down. You should continue trying to submit, by the standard means. I am not willing to spend hours disentangling assignments emailed to me by students who have made no real attempt to submit the assignment properly. An adjustment will be made to the penalties of all students, in accordance with the down time.

You should assume that the system will go down for short periods of time. No adjustment to your marks or due date will be made for down times of a short duration or down times that are not very close to the due date. Only in the event of major down times close to the due date, will an adjustment be made.

Medicals, etc

In the event of illness, you need to provide a medical certificate from student health, with sufficient information for the supervisor of the course to make a judgement on the severity of the illness. The original certificate must be provided (photocopies are not accepted), and it must be signed by a doctor, not a nurse.

- Note that not all medicals are accepted. For example, I refused two medicals in compsci 210 a couple of year's ago.
- No extensions will be given for assignments beyond the penalty date, since solutions will be provided on the web.
- If you do not submit an assignment, due to illness, your mark will usually be estimated from later assignments of at least the same level of difficulty. If you attempt insufficient assignments for your mark to be estimated, you will fail. (Besides, students who miss several assignments also fail the theory section.)
- If you do not sit a test or the final examination, your mark will be estimated from your rank in the other theory portion of the assessment. If you sit neither, you will fail.
- Marks obtained by requesting a remark of a test will be ignored in the case where the test is used to estimate the mark for the final examination.
- Sitting the test, then gaining special consideration due to illness is unlikely to have any real effect. Experience shows that minor illnesses such as colds have little effect on examination marks. Even a difference in 10% in your test mark would scale down to 2%, when combined with your other results, which is unlikely to have much effect on your final grade.

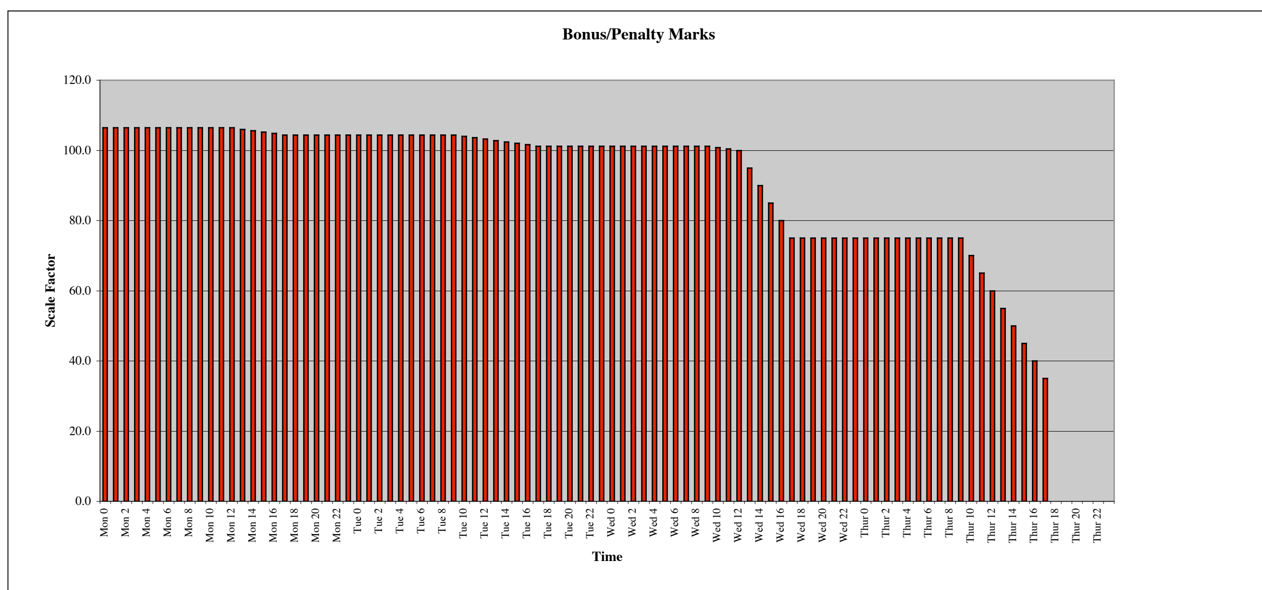
- If you failed the test, or barely passed the test, there is no point in applying for an aegrotat for the final examination. Your estimated grade will be based on your test mark, and you have to get a clear pass in the test (around a B-) before you will be considered for an aegrotat.
- Requests for a recount for your final examination only involve adding up the marks again, and checking that the grade has been computed correctly. They do not involve remarking. You can perform your own check for free by obtaining a photocopy of your examination.
- **If you are unable to do an assignment due to illness, or are granted an extension, still make sure that you make a dummy submission, before the assignment drop box closes.** Most assignments allow for some kind of documentation files. Put some text in the documentation file to indicate the situation. This makes administration much easier. A mark sheet, and submission log entry, and directory for placing your submission are automatically created for you, and I can edit this at a later date.

Bonus and Penalty Marks

The official due date for all assignments will be 12 noon on a Wednesday. However, a continuous scale of bonus and penalty marks means that your mark degrades gradually down to 35% at 5pm on Thursday. Thus there is only a slight penalty if you miss out on submission by a few minutes.

If you submit your assignment early, you will gain a bonus mark of 0.4% of the earned mark for each "laboratory hour" early, with a maximum bonus of 6.4%. If you submit your assignment late, but before 5pm on Thursday of that week, you will be penalised by 5% of the earned mark for each "laboratory hour" late. A "laboratory hour" is defined as an hour between 9a.m. and 5p.m. For example

Time	Multiplication factor
Monday 9am	106.4%
Monday 12 noon	106.4%
Monday 5pm	104.4%
Tuesday 9am	104.4%
Tuesday 12 noon	103.2%
Tuesday 5pm	101.2%
Wednesday 9am	101.2%
Wednesday 10am	100.8%
Wednesday 11am	100.4%
Wednesday 12 noon	100.0%
Wednesday 1pm	95.0%
Wednesday 2pm	90.0%
Wednesday 3pm	85.0%
Wednesday 4pm	80.0%
Wednesday 5pm	75.0%
Thursday 9am	75.0%
Thursday 10am	70.0%
Thursday 11am	65.0%
Thursday 12 noon	60.0%
Thursday 1pm	55.0%
Thursday 2pm	50.0%
Thursday 3pm	45.0%
Thursday 4pm	40.0%
Thursday 5pm	35.0%
Thursday 5:01pm	Can Not Submit



Assignment Dates

See the calendar at the end of this document.

All Assignments must Compile/Assemble/Run

You will gain no marks for assignments that do not compile/assemble.

If you develop your program in stages, make sure that later changes do not cause previously tested inputs to stop working.

Copying of Assignments

We will run a program to detect copying of assignments. This program is very successful in picking up copied assignments. Both the person doing the copying and the person permitting the copying will lose all marks for the assignment, even if only a portion of the assignment is copied. Note that this also applies to identical twins, married couples, people in relationships, friends, and people just "working together". Also be aware that people selling assignments usually sell them to about 10 people, and all such students lose marks. Even if you are never picked up for copying, the best way to guarantee you fail is to not do assignments. It should be pointed out that students picked up twice in connection with copying end up failing the practical component, and hence failing, the whole course. This applies even to "A" grade students who permit other students to copy their assignments.

For our policy regarding cheating in assignments see

http://www.cs.auckland.ac.nz/CheatingPolicy.html.

Linefeeds and Carriage Returns

A shell script convertFile.bash will be provided in the ~/bin directory to convert standard input text to UNIX, Mac, and PC format. Shell scripts toUNIX.bash, toMac.bash, toPC.bash, taking a list of files, will convert the individual files.

Test and Examination

You will find that it is absolutely essential that you do the assignments. Otherwise, you will not pass.

There is a one and a half hour test and a two hour final examination.

Test Date

See the calendar at the end of this document.

Mark Breakdown (Subject to Change)

Assignment 1	7.0%
Assignment 2	13.0%
Test	20.0%
Examination	60.0%
Total	**100.0%**

Bruce Hutton

Dates

Week	Sun	Mon	Tue	Wed	Thu	Fri	Sat	
1	25-Feb	26-Feb	27-Feb	28-Feb	01-Mar	02-Mar	03-Mar	
2	04-Mar	05-Mar	06-Mar	07-Mar	08-Mar	09-Mar	10-Mar	
3	11-Mar	12-Mar	13-Mar	14-Mar	15-Mar	16-Mar	17-Mar	
4	18-Mar	19-Mar	20-Mar	21-Mar	22-Mar	23-Mar	24-Mar	
5	25-Mar	26-Mar	27-Mar	**28-Mar** **Assgn 1**	29-Mar	30-Mar	31-Mar	**Assgn 1 12 noon Wed** **28-Mar-06**
6	01-Apr	02-Apr	03-Apr	04-Apr	**05-Apr** **Test**	06-Apr Easter	07-Apr	**Test 6:30pm-8pm Thu** **05-Apr-06**
	08-Apr	09-Apr Easter	10-Apr	11-Apr	12-Apr	13-Apr	14-Apr	**Mid Semester Break**
	15-Apr	16-Apr	17-Apr	18-Apr	19-Apr	20-Apr	21-Apr	**Mid Semester Break**
7	22-Apr	23-Apr	24-Apr	25-Apr ANZAC	26-Apr	27-Apr	28-Apr	
8	29-Apr	30-Apr	01-May	02-May	03-May	04-May	05-May	
9	06-May	07-May	08-May	09-May	10-May	11-May	12-May	
10	13-May	14-May	15-May	**16-May** **Assgn 2**	17-May	18-May	19-May	**Assgn 2 12 noon Wed** **16-May-06**
11	20-May	21-May	22-May	23-May	24-May	25-May	26-May	
12	27-May	28-May	29-May	30-May	31-May	01-Jun	02-Jun	
	03-Jun	04-Jun Queen	05-Jun	06-Jun	07-Jun	08-Jun	09-Jun	**Examinations**
	10-Jun	11-Jun	12-Jun	13-Jun	14-Jun	15-Jun	16-Jun	
	17-Jun	18-Jun	19-Jun	20-Jun	21-Jun	22-Jun	23-Jun	
	24-Jun	25-Jun	26-Jun	27-Jun	28-Jun	29-Jun	30-Jun	
Week	**Sun**	**Mon**	**Tue**	**Wed**	**Thu**	**Fri**	**Sat**	