## Learning outcomes

<table>
<thead>
<tr>
<th>Learning outcome</th>
<th>Instructional methods</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employ static and dynamic analysis tools to detect faults in a given piece of software.</td>
<td>Lecture-based instruction</td>
<td>LEX</td>
</tr>
<tr>
<td>Employ profiling tools to identify performance issues (both time and memory) in a given piece of software.</td>
<td>Lab-based hands-on exercises, both individual and group</td>
<td>EXP LPR</td>
</tr>
<tr>
<td>Employ testing frameworks to write tests that fail in the presence of software faults, and pass otherwise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Employ a structured, methodical approach to detecting, testing, identifying and correcting software faults.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Learning activities

- (LEX) Ten weekly lab-based exercises, due in the lab session, done in weeks 1 through 10.

- (PRE)/(PST) A “process” team project, done twice, once as a pre-assessment in weeks 1, 2, and 3 of the semester, and a second time as a post-assessment in weeks 10, 11 and 12. Students are required to document their development/debugging process.

- (EXP) Four two-week exploratory projects, done in weeks 4-5, 6-7, 8-9 and 13-14. These projects ask students to apply the tools and techniques they have been taught up to that point in the course to open-source projects found at repositories such as SourceForge, GitHub and BitBucket. Students are required to document their use of the tools and the results they obtained.

- (LPR) A two-part in-lab practical exam, in weeks 13 and 14. Part 1 will cover basic skills, part 2 will cover process.
# Grading

<table>
<thead>
<tr>
<th>INDIVIDUAL</th>
<th>TEAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEX 1000 (28%)</td>
<td>PRE 200 (6%)</td>
</tr>
<tr>
<td>EXP 800 (22%)</td>
<td>PST 400 (11%)</td>
</tr>
<tr>
<td>LPR 1200 (33%)</td>
<td></td>
</tr>
</tbody>
</table>
Activity log (.csv file)

- design discussions (DD)
- coding progress (CP)
- testing done (TD)
- bugs found and fixed (BFF)
- tools used (editor, compiler, etc) (TU)

DATE, TIME, DURATION, UBIT, ACTIVITY, TOOLS, NOTES

20170130, 10:15 AM, 10 m, alphonce, CP, compiler, write main function
EXP 01

- Apply the tools and techniques they have been taught up to that point in the course (compilers w/options, gdb, cunit) to open-source projects found at repositories such as SourceForge, GitHub and BitBucket.

- Attempt to track down a bug. Pick a C language project that has open bug reports.

- Students are required to document their use of the tools and the results they obtained.
Due date:
Tuesday, March 7

- Work alone (i.e. each person must make their own submission), but discussion of tool usage is OK.

- Use your PRIVATE SOLO repo.

- Keep a log of your activities as you attempt to track down a bug.

DATE, TIME, DURATION, UBIT, ACTIVITY, TOOLS, NOTES
20170130, 10:15 AM, 10 m, alphonce, CP, compiler, write main function
Basic approach to assessing early work

PRE and LEX: If you have engaged with the task, you will receive credit. (Show ability to learn and be introspective about development.)

EXP, LPR, POST: Are you demonstrating an ability to use the tools/techniques covered in class?
For recitation

Think of good tests for a simple expression evaluator:

The expression evaluator provides one function:
```
struct primitiveOption * evaluate(char * input);
```

This function accepts a string representing an arithmetic expression as input, and returns a NONE option if the input cannot be evaluated as a valid arithmetic expression, or SOME(int) otherwise, where the option wraps the value of the expression.

Arithmetic expressions are defined recursively:

- **Base case:** integer constants (such as 37, 0 and -143) are arithmetic expressions.
- **Recursive cases:** if alpha and beta are arithmetic expressions, so are
  
  - alpha + beta, representing the integer sum of the values of alpha and beta
  - alpha - beta, representing the integer difference of the values of alpha and beta
  - alpha * beta, representing the integer product of the values of alpha and beta
  - alpha / beta, representing the integer quotient of the values of alpha and beta
    ( alpha ), whose value is the same as the value of alpha

The usual evaluation rules hold:

- All operators are left associative
  
  - e.g. x - y - z evaluates as (x - y) - z

- Parenthesized expressions have higher precedence than non-parenthesized expressions
  
  - e.g. (x + y) * z evaluates the sum first, then computes the product

- * and / have higher precedence than + and -
  
  - e.g. x - y / z evaluates as x - (y / z)

Whitespace (space (' '), tab ('	'), and newline ('
')) may appear anywhere in the expression, except inside a number.
Survey insights from 13 responses (hopefully more to come)

- We will use languages other than C
- make/build tools
- code coverage tools
- short C tutorial
Experienced Lisp programmers follow bottom-up design—changing the language to suit the problem. You don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish I had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together.

In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

It's worth emphasizing that bottom-up design doesn't mean just writing the same program in a different order. When you work bottom-up, you usually end up with a different program. Instead of a single, monolithic program, you will get a larger language with more abstract operators, and a smaller program written in it.
Experienced programmers change the language to suit the problem.
Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called bottom-up design—changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

It's worth emphasizing that bottom-up design doesn't mean just writing the same program in a different order. When you work bottom-up, you usually end up with a different program. Instead of a single, monolithic program, you will get a larger language with more abstract operators, and a smaller program written in it.
Paul Graham

http://paulgraham.com/progbot.html
Types of types

- primitive (values are atomic)
- (discriminated/disjoint) union \((A + B)\)
- Cartesian product \((A \times B)\)
- mapping \((D \rightarrow R)\)
- recursive (base case, recursive case)
Build good abstractions

- The type system is there for you: use it!
Primitives

- boolean, int, float, double, char, etc.
Cartesian product

- struct (C), class (Java), record (ML)
Disjoint union

union (C), variant record (Pascal), interface/class hierarchy (Java), datatype (ML)
Mappings

- explicit representations
  - a map<D,R>: HashMap<D,R>, array<int,
- implicit representations
  - functions
- hybrid representations
  - memoized functions, dynamic programming

  1.) Top-Down: Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as Memoization.

  2.) Bottom-Up: Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as Dynamic Programming.

Note that divide and conquer is slightly a different technique. In that, we divide the problem in to non-overlapping subproblems and solve them independently, like in mergesort and quick sort.

[https://www.codechef.com/wiki/tutorial-dynamic-programming]